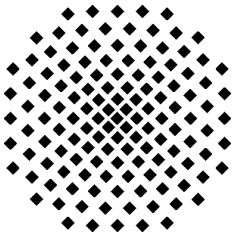

Universität Stuttgart
Fakultät Informatik

Studiengang:	Informatik
Prüfer:	Prof. Dr. Mitschang
Betreuer:	Thomas Schwarz
begonnen am:	01.11.2002
beendet am:	14.05.2003
CR-Klassifikation:	H.2.8, H.3.3, H.3.5, H.4.3

Studienarbeit Nr. 1880

Standard-Verzeichnisdienste als räumliches Verzeichnis für Informationsdienste in Nexus

Carsten Neumann

Institut für Parallele und
Verteilte Systeme (IPVS)
Abteilung Anwendersoftware
Universitätsstr. 38
D-70569 Stuttgart

Zusammenfassung

Die Forschungsgruppe NEXUS beschäftigt sich mit der Entwicklung einer offenen Plattform für ortsbasierte Dienste, auf die Benutzer einfach zugreifen können. Hierfür werden räumliche Verzeichnisse, sog. Area Service Register benötigt, die die Speicherung der Informationen übernehmen.

In dieser Arbeit wird untersucht, ob sich Standard-Verzeichnisdienste, die sich in anderen Bereichen bewährt haben, für einen einfachen Aufbau dieser Area Service Register eignen oder welche Erweiterungen evtl. dafür notwendig sind.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

In der Forschungsgruppe NEXUS wird eine offene Plattform für Location Based Services entwickelt. Diese Plattform ermöglicht es, ortsbezogene Informationsangebote bereitzustellen und zwar auf ähnlich einfache Weise, wie mit dem Aufstellen eines WWW-Servers. Einzelne Angebote auf einem Server sind typischerweise auf bestimmte Kategorien (z.B. Restaurants) und Gebiete (z.B. Stuttgart) beschränkt. Mit Hilfe dieser Plattform können dann Anwendungen einfach auf eine Vielzahl von ortsbezogenen Informationen zugreifen.

Das Area Service Register (ASR) ist der Teil der Plattform, der zu einer Anfrage einer Anwendung nach bestimmten Informationen in einem bestimmten Gebiet eine Liste von Servern zurückliefert, welche die entsprechenden Informationen anbieten und für das Anfragegebiet zuständig sind.

Aufgabenstellung

Ziel dieser Arbeit ist es, auf der Basis von Standard-Verzeichnisdiensten einen Dienst zu entwickeln, der zu jedem Informationsangebot eine Reihe von Meta-Daten (Name, Gebiet, gespeicherte Informationen, Detailgrad) speichert und räumliche Anfragen sowie die Änderung der Meta-Daten ermöglicht.

Dazu sollen sollen verschiedene Standard-Verzeichnisdienste sowie ihre Eignung zur Implementierung des räumlichen Verzeichnisses untersucht und ihre Stärken und Schwächen miteinander verglichen werden. Dabei soll zuerst auf Basis der Spezifikationen die theoretische Umsetzbarkeit beurteilt werden. Anschließend sind Produkte zu suchen, die die benötigte Funktionalität entweder alleine oder mittels zusätzlicher Erweiterungen bereitstellen können. Kann das gewünschte räumliche Verzeichnis auf dieser Basis nicht implementiert werden, ist eine geeignete Erweiterung zu einem bestehenden Produkt zu erstellen. In diesem Fall ist ein Produkt mit frei verfügbaren Quelltext zu bevorzugen.

Anschließend soll die Performanz des räumlichen Verzeichnisses beurteilt und mit dem bereits in einer anderen Diplomarbeit entwickelten, Datenbank-basierten Ansatz verglichen werden.

Aufbau des Dokuments

Zuerst wird im 2. Kapitel nach einer Untersuchung verschiedener Standard-Verzeichnisdienste begründet, warum die Implementierung des räumlichen Verzeichnisses auf Basis des OpenLDAP-Systems erfolgt. In folgenden Kapitel wird dann erläutert, welche Komponenten am Gesamtsystem beteiligt sind und welche Erweiterungen notwendig sind. Der genaue Aufbau und die Umsetzung dieser neuen Komponenten erfolgt dann im Kapitel 4. Anschließend wird das erstellte System bewertet und auf Alternativen zur vorgestellten Implementierung eingegangen. Die bei

der Umsetzung gewonnenen Erkenntnisse werden dann im letzten Kapitel noch einmal zusammengefasst.

Im Anhang wird auf die Installation und Konfiguration der verwendeten Programme erklärt und es werden einige wichtige Dateien aufgelistet.

2. Standard-Verzeichnisdienste

Vor der Entscheidung für einen bestimmten Verzeichnisdienst zur Umsetzung der Aufgabe muss zuerst untersucht werden, in wieweit sich die verscheidenen Standard-Verzeichnisdienste für den Aufbau eines räumlichen Verzeichnisses im Rahmen des Nexus-Projektes eignen. Dazu werden zuerst die Anforderungen beschrieben, die an das räumliche Verzeichnis gestellt werden und dann verschiedene Standard-Verzeichnisdienste daraufhin untersucht, ob sie diese Anforderungen erfüllen können oder sich entsprechend erweitern lassen. Auch ist zu prüfen, ob verfügbare Implementierungen dieser Standards evtl. bereits die benötigten Erweiterung enthalten.

2.1 Anforderungen

Das Area Service Register (ASR) im Rahmen des Nexus-Systems (vgl. [25] und [26]) hat die Aufgabe, Informationen über Augmented Areas zu speichern und auf Anfrage bereit zu stellen.

2.1.1 Zu speichernde Daten

Für jeden Verzeichniseintrag (Augmented Area) sind folgende Informationen zu speichern:

- Name des zuständigen Spatial Model Servers
- Gebiet der Augmented Area (z.B. Gebäude als POLYGON ((10,15),(20,40)...)
- Schema, also die Nexus-Typen der enthaltenen Objekte inklusive Angaben zum Detaillierungsgrad. Diese Typen sind in einer erweiterbaren Hierarchie festgelegt

2.1.2 Zu unterstützende Anfragen

Aufgabe des Area Service Registers (ASR) ist es, bei einer Anfrage an das Nexus-System eine Liste der Spatial Model Server zu liefern, deren Gebiet sich mit dem Anfragegebiet überschneidet und deren Schema den geforderten Nexus-Typ oder einen seiner Sub-Typen enthält. Durch geeignete Wahl der Suchparameter ist es auch möglich, alle verfügbaren Dienste für ein Gebiet (durch Wahl des Wurzel-Knotens der Typhierarchie) oder alle Dienste eines bestimmten Typs und seiner Subtypen (durch entsprechend großes Suchgebiet) zu finden.

Aus den oben aufgeführten Datentypen und Suchanfragen ergeben sich folgende Anforderungen an das räumliche Verzeichnis, die dann als Grundlage für die Prüfung der Standard-Verzeichnisdienste dienen:

(1) Das Verzeichnis muss eine Typhierarchie verwalten können: Anfragen auf einen bestimmten Typ können auch durch Sub-Typen erfüllt werden.

(2) Das Verzeichnis muss geometrische Attribute speichern und darauf Vergleichsoperationen (Überlappung) durchführen können.

(3) Wenn möglich, sollte das Verzeichnis räumliche Anfragen durch geeignete Indexstrukturen o.ä. unterstützen können, um die zeitraubende Überprüfung aller in Frage kommenden Einträge auf geometrische Überlappung mit dem Suchgebiet zu vermeiden oder zu beschleunigen.

2.2 Standard-Verzeichnisdienste

Verzeichnisse sind meist baumförmig organisierte Sammlungen von Daten, die eher statischer Natur sind. Die Einträge können dabei unterschiedlichen Typen angehören, besitzen einen eindeutigen Namen und weitere Attribute. Jedes Verzeichnis unterstützt dabei die Suche nach Einträgen über ihren Namen, manche zusätzlich auch die Suche nach Einträgen mit bestimmten Eigenschaften (Attributwerten).

Zu diesen Verzeichnissen gehören dann Verzeichnisdienste, die den Zugriff auf derartige Verzeichnisse erlauben, um Einträge zu suchen, hinzuzufügen oder zu ändern. Unterschieden werden Verzeichnisse durch das Protokoll, nach dem der Zugriff abläuft, denn nur dieser ist durch Standards definiert. Wie die Speicherung und Verwaltung der Daten erfolgt, ist dagegen nicht festgelegt (vgl. [22]).

Bei der Bewertung der Standard-Verzeichnisdienste werden nur die im vorigen Abschnitt aufgeführten Anforderungen herangezogen. Auf weitere Eigenschaften wie Verteilung, Replikation und Sicherheitsmechanismen wird nicht näher eingangen.

2.2.1 Domain Name Services (DNS)

Der Domain-Name-Service (DNS) (vgl. [20] und [21]) ist ein Protokoll, das speziell auf die Anforderung zugeschnitten ist, zu einem Domain-Namen die passende IP-Adresse zu finden. Dazu werden die Namen in einem hierarchischen Namensraum organisiert, zu jedem Eintrag sind außerdem noch weitere Informationen speicherbar. Diese zusätzlichen Informationen stammen aus einer Menge von bereits vordefinierten Typen, die sich erweitern lässt.

Allerdings ist im DNS-Protokoll nur eine Art von Anfragen vorgesehen: die Suche nach dem Eintrag mit einem bestimmten Namen. Eine Anfrage nach Einträgen, deren weitere Attribute bestimmte Eigenschaften besitzen (wie sie für ein ASR nötig wäre), ist nicht vorgesehen.

Vergleicht man nun diese Eigenschaften mit den oben genannten Anforderungen, wird deutlich, dass die benötigten Eigenschaften bis auf die hierarchische Organisation des Namensraums nicht vorhanden sind. Eine entsprechende Erweiterung ist nicht vorgesehen ist und würde daher – wenn auch prinzipiell möglich – erheblichen Aufwand erfordern.

2.2.2 X.500 Directory Service

Der X.500-Standard[1] definiert dagegen ein weitaus flexibleres Verzeichnis: die Einträge, die jeweils mindestens einer Objektklasse angehören müssen, werden in einer hierarchisch organisierten Struktur, dem Directory Information Tree (DIT) gespeichert. Jeder Eintrag besitzt dabei einen verzeichnisweit eindeutigen Bezeichner, dem distinguished name (DN), der sich jeweils aus dem DN des übergeordneten Knoten und dem ausgezeichneten Wert (distinguished value) eines speziellen Attributes zusammensetzt. Zusätzlich besitzt jeder Eintrag eine beliebige Anzahl von Attributen, die jeweils einen, aber auch mehrere Werte aufnehmen können. Dabei ist durch die Objektklasse des Eintrags festgelegt, welche Attribute ein Eintrag besitzen darf und welcher Syntax diese folgen müssen. Mit dieser Attributsyntax ist gleichzeitig auch festgelegt, welche Vergleichsoperationen auf diesen Attributen zulässig sind.

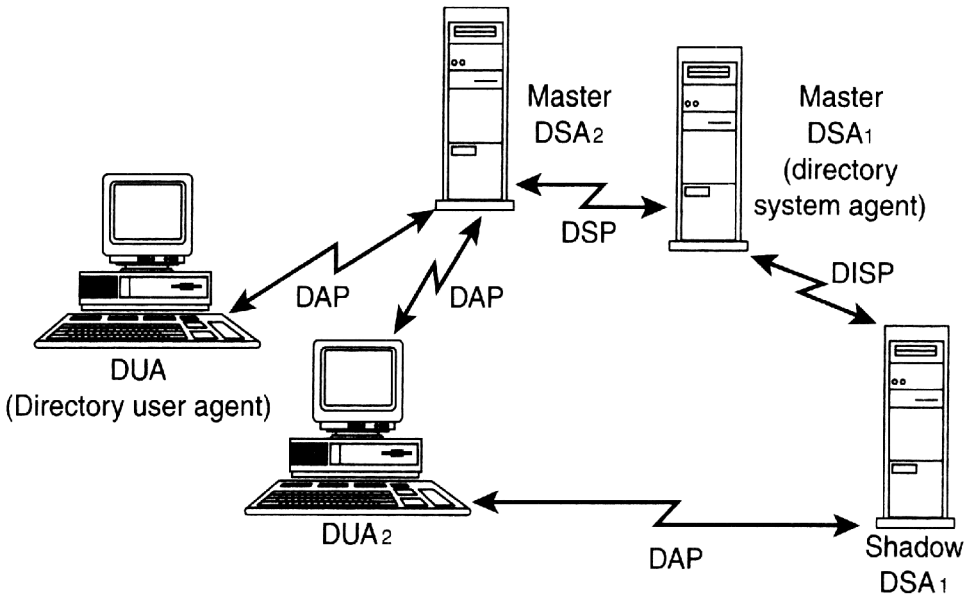

Abb. 1: Aufbau eines X.500-Systems (aus [23])

Durch das Directory Access Protocol (DAP), das einen Teil des X.500-Standards bildet, werden eine Anzahl erlaubter Operationen zwischen dem Directory User Agent (DUA) und dem Directory Service Agent (DSA) festgelegt (vgl. Abb. 1). Durch diese Operationen kann der Client Anfragen an das Verzeichnis stellen oder den Inhalt des Verzeichnisses modifizieren. Zur Verteilung und Replikation eines Verzeichnisses zwischen mehreren Servern wird ein anderes Protokoll, das Directory Service Protocol (DSP) verwendet.

1) Die Bezeichnung X.500 wird hier stellvertretend für eine Sammlung von Empfehlungen verwendet, die von der International Telecommunication Union (ITU) erstmals im Jahre 1998 herausgegeben und seitdem mehrfach aktualisiert und erweitert wurde (vgl. [1]-[10]).

Im Gegensatz zum Domain Name Service erlaubt der X.500-Standard auch die Suche nach Einträge mit bestimmten Eigenschaften durch Definition von Suchfiltern auf den zugehörigen Attributwerten. Auch können Suchanfragen durch Angabe eines Wurzelknotens für die Suche auf einen Teilbaum beschränkt werden. Dadurch ließe sich die Suche nach Augmented Areas eines bestimmten Nexus-Typs und seiner Subtypen leicht umsetzten.

Allerdings unterstützt der X.500-Standard keine geometrischen Attribute. Die vordefinierten Attributtypen sind auf den Haupteinsatzzweck, Adressverzeichnisse, Rechteverwaltungen o.ä. aufzubauen, ausgerichtet und definieren daher Typen zur Speicherung von Namen, Adressen, Telefonnummern usw. Um die für das Area Service Register benötigten geometrischen Information zu speichern, müssten also geeignete Attributtypen und zugehörige Vergleichsoperationen (geometrische Überlappung) eingeführt werden. Derartige benutzerdefinierte Erweiterungen sind im Protokoll vorgesehen, erfordern aber die entsprechende Modifikation des Directory Service Agents.

Im Standard sind zwar auch geographische Vergleichsoperationen vorgesehen (*zonal match*), diese sind allerdings nicht durch Vergleich von geometrischen Objekten bzw. Attributen realisiert, vielmehr ist ein geographisches Verzeichnis (*gazetteer*) nötig, das Orts- und Gebietsnamen hierarchisch verwaltet. Dort wird dann z.B. gespeichert, welche Orte zu einem bestimmten Gebiet gehören, sodass dann nach Verzeichniseinträgen gesucht werden kann, die in diesem Gebiet liegen, indem geprüft wird, ob ihre Ortsbezeichnung mit einem der entsprechenden Orte übereinstimmt. Dieses Verfahren unterstützt so allerdings nicht die für das Area Service Register nötigen Anfragen, bei denen geometrische Angaben verglichen werden sollen.

2.2.3 Lightweight Directory Access Protocol (LDAP)

Das Lightweight Directory Access Protocol (LDAP) entstand als Reaktion auf X.500-Systeme, deren Einsatz – insbesondere durch die notwendige vollständige Implementierung des ISO/OSI-Protocol-Stacks – als zu aufwändig empfunden wurde. Ziel der Entwicklung war es, die Funktionalität eines X.500-Systems ohne einige selten benötigte Funktionen mit einem einfacheren System zu erreichen, das auf bestehende Kommunikationsprotokolle wie TCP/IP aufsetzt. Ursprünglich war LDAP dabei nur als Vermittler zwischen dem Client und dem X.500-System gedacht (vgl. Abb. 2). Der LDAP-Client sollte mittels des einfacheren Protokolls auf einen LDAP-Translator zugreifen, der die Anfragen dann auf das komplexere Directory Access Protocol (DAP) des X.500 Standards umsetzte. Inzwischen wird LDAP in der Regel als Stand-Alone-System eingesetzt und liegt mittlerweile in seiner 3. Version (LDAPv3) vor (vgl. [11]-[19]).

Seit dieser 3. Version ist nun auch in LDAP die Erweiterung um eigene Suchfilter (*extensible match filter*) explizit vorgesehen. Damit besitzt LDAP in Bezug auf die für das Projekt benötigten Anforderungen die gleichen Eigenschaften wie X.500-Verzeichnisse. Die Organisation der Einträge folgt dem gleichen Schema, auch bei LDAP sind neue Objektklassen und Attributtypen definierbar. Auch die Beschränkung

der Suche auf einen Teilbaum ist möglich. Dabei besitzt allerdings auch LDAP keine vordefinierten Typen oder Operationen für geometrische Informationen.

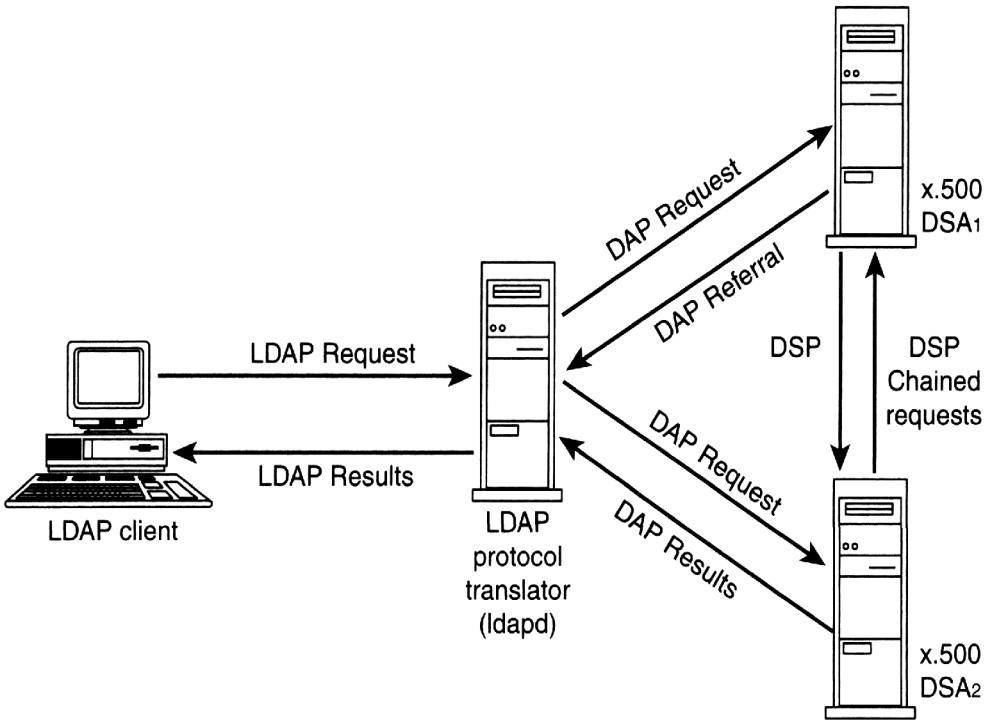

Abb. 2: LDAP als Gateway für ein X.500-Directory (aus [23])

2.2.4 Sonstige Verzeichnisdienste

Neben DNS, X.500 und LDAP gibt es noch weitere Standard-Verzeichnisdienste, wie z.B. Finger, Whois und Whois++ (vgl. [22]). Diese Verzeichnisdienste bieten allerdings alle nur eine sehr eingeschränkte, auf ihren Einsatzzweck zugeschnittene Funktionalität und sehen außerdem als read-only Dienste keine Modifikation der Directory-Einträge über das jeweilige Protokoll vor. Außerdem bieten sie normalerweise keine Suche nach Einträgen mit bestimmten Attributeinträgen. Daher bilden sie auch keine geeignete Basis für die Realisierung des gesuchten räumlichen Verzeichnisses.

2.3 Implementierungen

Basierend auf dien am ehesten geeigneten Standards (X.500 bzw. LDAP) gibt es verschiedene Implementierungen. X.500-Systeme sind durch ihre ausgefeilten Verfahren für Replikation, Server-Server-Kommunikation und Datenmanagement besonders gut geeignet für große, massiv skalierbare Anwendung in stark verteilten Umgebungen. LDAP-Systeme sehen allerdings auch Verfahren für Replikation und sicheren Zugriff vor, für dieses Projekt reichen die Funktionalitäten daher aus.

2.3.1 Kommerzielle Systeme

Für den LDAP-Standard gibt es eine Vielzahl von kommerziellen Implementierungen, eine Übersicht über einige wichtige Vertreter findet sich z.B. in [28]:

- Novell Directory Services (NDS) – eDirectory
- Microsoft Active Directory Services (ADS)
- iPlanet (Sun/Netscape)
- IBM SecureWay Directory

Bis auf das Produkt von Microsoft, das bisher nur für Windows-Betriebssysteme verfügbar ist und teilweise proprietäre Eigenschaften besitzt, bieten all diese Systeme eine vollständige Umsetzung des aktuellen LDAPv3-Standards und sind auf einer Vielzahl von Plattformen verfügbar.

Alle Systeme sehen Möglichkeiten vor, das LDAP System entsprechend der im Standard vorgesehenen Optionen um eigene Attributtypen und benutzerdefinierte Vergleichsoperationen zu ergänzen. So ist z.B. bei NDS eine Plug-In Schnittstelle enthalten, mit der sich der LDAP-Server erweitern lässt (vgl. [29]). Damit läßt sich neben der Anforderung (1), die bereits grundsätzlich durch den hierarchischen Aufbau des LDAP-Verzeichnesses umsetzbar ist, auch die die Anforderung (2) erfüllen: durch *Pre-operation* Plug-Ins lässt sich das Format der benutzerdefinierten Datentypen überprüfen, durch *Matching-rule* Plug-Ins können server-seitig die neuen räumlichen Vergleichsoperationen eingeführt werden.

Problematisch erscheint allerdings die Umsetzung der Anforderung (3), da im LDAP-Standard keine geometrischen Attribute und daher auch keine Unterstützung einer räumlichen Suche vorgesehen ist. Normalerweise erfolgt die Suche nach passenden Einträgen durch die Index-Funktion der vom LDAP-Server benutzten Datenbank. Dabei kann zwar eine eigene Index-Funktion für benutzerdefinierte Datentypen aufgerufen werden, allerdings sind vom System nur eindimensionale Index-Werte vorgesehen. Daher erscheint eine Erweiterung des LDAP-Servers um eine mehrdimensionale Suchfunktionalität ohne Verfügbarkeit des Quelltextes ausgeschlossen.

2.3.2 OpenLDAP

Neben den genannten kommerziellen Systemen existiert auch eine freie LDAP-Implementierung (OpenLDAP)[2] eines LDAP Stand-Alone-Servers. Die zur Zeit aktuelle Version 2.1.8 erfüllt auch den LDAPv3-Standard und bietet daher grundsätzlich die gleiche Funktionalität wie die kommerziellen Systeme.

2) http://www.openldap.org

Durch die Verfügbarkeit des Quelltextes können auch evtl. notwendige tiefergreifende Anpassungen oder Erweiterungen des LDAP-Servers erfolgen. Gleichzeitig auch besteht die Möglichkeit, den OpenLDAP-Server durch dynamisch ladbare Module zu erweitern, um dadurch neue Syntax-Typen und Vergleichsoperationen zu definieren ohne dazu direkt den Code zu verändern.

2.4 Entscheidung

Die Realisierung des räumlichen Verzeichnisses erfolgt auf Basis des OpenLDAP-Systems, das am Besten den beschriebenen Anforderungen entspricht. Sowohl X.500 als auch LDAP-Standard bieten in Bezug auf die genannten Anforderungen ähnliche Funktionalität, reichen ohne Erweiterungen aber nicht zur Umsetzung. Daher bietet sich das OpenLDAP-System an, das hierbei durch die freie Verfügbarkeit des Quelltextes den größten Spielraum bietet und gemäß Aufgabenstellung zu bevorzugen ist.

3. Analyse und Grobentwurf

Das in diesem Projekt verwendete LDAP-System besteht aus mehreren Schichten (vgl. Abb. 3). Der Benutzter ruft bei einer Abfrage Funktionen des LDAP-Clients auf. Dieser setzt diese Anfrage nun auf das LDAP-Protokoll um, das zur standardisierten Kommunikation zwischen LDAP-Client und LDAP-Server verwendet wird und übermittelt diese via TCP/IP an den LDAP-Server. Der LDAP-Server speichert und verwaltet die Daten aber nicht selber, sondern wandelt die Anfrage in Zugriffe auf die Backend-Datenbank um.

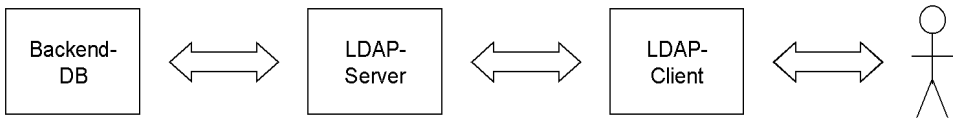

Abb. 3: Aufbau eines LDAP-Systems

Die erhaltenen Ergebnisse werden dann in das vom LDAP-Protokoll definierte Format umgesetzt und dem Client zurückgeliefert. Dieser muss dann abschließend die Informationen in die von der Anwendung bzw. dem Benutzer erwartete Darstellung übersetzen.

Zur Implementierung des räumlichen Verzeichnisses muss nun eine geeignete Organisation der zu speichernden Daten gewählt werden, die die bereits vorhandenen Eigenschaften des LDAP-Systems möglich gut ausnützt.

3.1 Aufbau des Directory Information Trees (DIT)

Wie bereits im vorigen Kapitel kurz erläutert, sind die Basis eines LDAP-Verzeichnis Einträge. Einträge sind Sammlungen von Attributen und werden als Knoten in einem Baum, dem sog. Directory Information Tree (DIT) gespeichert. Die Position jedes Knotens innerhalb der Baumstruktur ergibt sich durch den für jeden Knoten eindeutigen *distinguished name (DN)*. Dieser ergibt sich aus dem DN des Vaters und dem *relative distinguished name (RDN)* des jeweiligen Knotens, der sich aus Bezeichner und Wert eines ausgewählten Attributs ergibt[3].

Um die hierarchische Struktur des LDAP-Verzeichnis bei der späteren Suche nach passenden Einträgen zu nutzen, werden alle Nexus-Typen entsprechend ihrer

3) Im Beispiel der Typhierarchie (vgl. Abb. 2) setzt sich so z.B. der DN des „Restaurant-Knotens“ (nt=restaurant, nt=root, o=nexus) aus dem DN des Vater-Knotens (nt=root, o=nexus) und dem RDN (nt=Restaurant) zusammen.

Klassenhierarchie als *NexusTypeNode*[4] *(ntn)* unterhalb des LDAP-internen Wurzelknotens „o=nexus“ in das Verzeichnis eingetragen. Alle Augmented Areas werden dann als *NexusAreaNode (nan)* als direkte Nachfolger des entsprechenden *NexusTypeNode* gespeichert. Für jede Nexus-Typhierarchie (z.B. Abb. 4) kann nach diesem Schema ein entsprechender LDAP-Verzeichnisbaum (vgl. Abb. 5) aufgebaut werden.

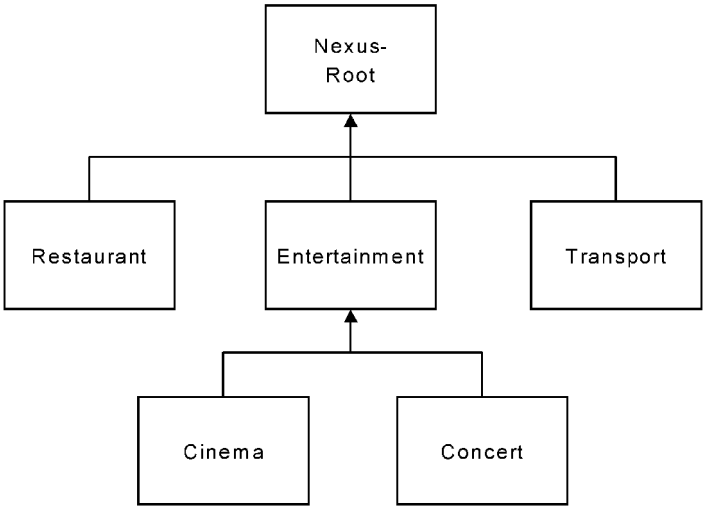

Abb. 4: Nexus-Typhierarchie

Durch diesen Verzeichnisaufbau können dann Suchanfragen auf einen bestimmten Teilbaum beschränkt werden, denn alle Augmented Areas des gesuchten Nexus-Typs oder der möglichen Subtypen sind unterhalb des entsprechendes *NexusTypeNode* zu finden (vgl. Abb. 6). Der Aufbau und die Bedeutung der einzelnen Attribute wird später (vgl. Kap. 4.1) erläutert.

Da der Nexus-Typ für jeden *NexusAreaNode* wegen der Organisation des Verzeichnisbaums eindeutig ist, muss eine Augmented Area, die mehrere Einträge für den Nexus-Typ besitzt, durch mehrere Knoten im Verzeichnisbaum vertreten werden. Dazu muss für jeden Nexus-Typ der Augmented Area ein *NexusAreaNode* an der entsprechenden Stelle des Verzeichnisbaums eingefügt werden (in diesem Bsp. die Augmented Area mit „nal=nexus://moviecenter.de“). Dabei müssen alle Attribute des Eintrags, auch die Flächendefinition der zugeordneten *NexusArea,* wiederholt werden. Da die Adresse des zuständigen Nexus-Servers, der *NexusAreaLocator,* für jede Augmented Area eindeutig ist, können durch eine entsprechende Anfrage leicht alle Einträge für eine Augmented Area gefunden werden.

Eine Gliederung der Einträge nach ihrer geographischen Anordnung erfolgt nicht. Daher kann z.B. auch eine Anfrage nach allen Augmented Areas in der Umgebung eines

4) Alle Bezeichner für Objektklassen, Attribute usw., die für das LDAP-Verzeichnis benötigt werden, erhalten als Präfix „*Nexus*“, um ihre Eindeutigkeit sicherzustellen und die Zugehörigkeit zu dieser Erweiterung zu dokumentieren.

Anwenders auch nicht zur Beschleunigung auf einen Teilbaum des Verzeichnisses beschränkt werden. Dies verhindert auch eine geographische Aufteilung des Verzeichnisses durch Partitionierung und Replikation in einzelne Regionen, da die Zuständigkeitsbereiche jedes LDAP-Servers immer durch einen Namensraum, also einen Teilbaum des Gesamtverzeichnisses festgelegt werden.

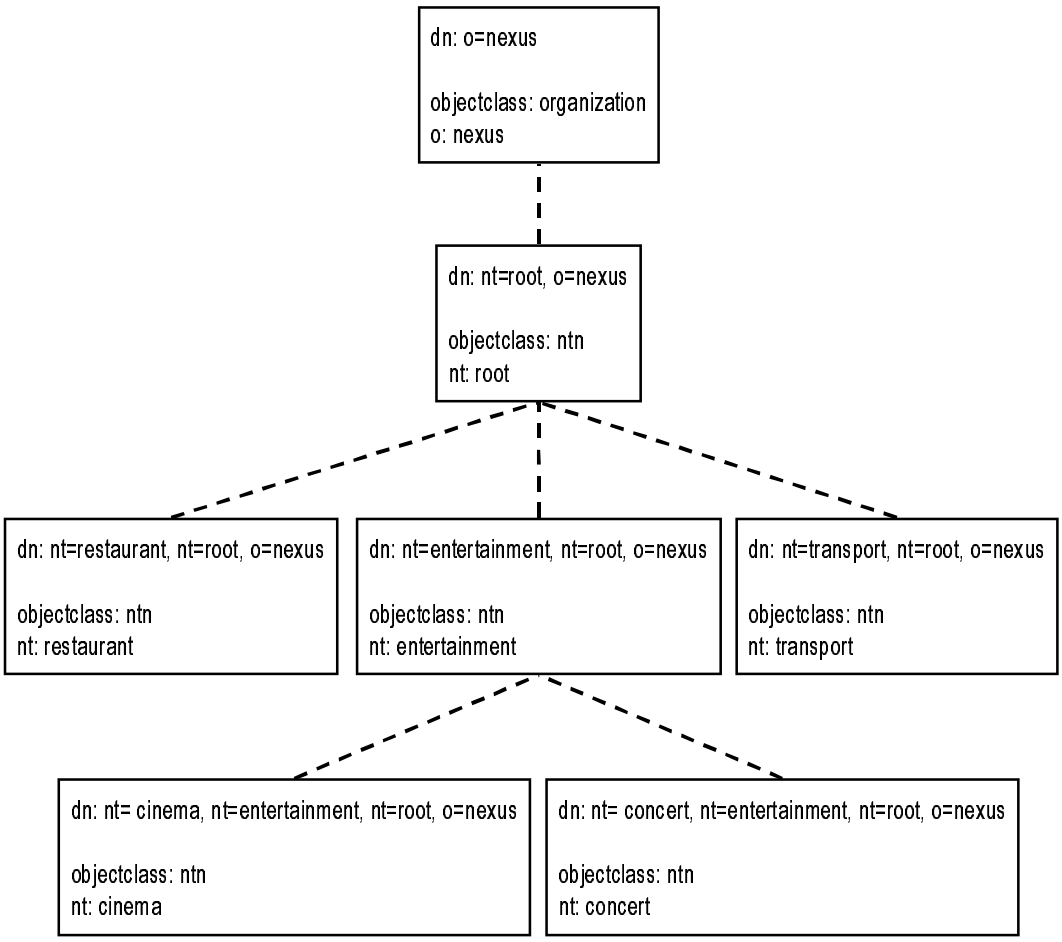

Abb. 5: Umsetzung einer Typhierarchie in einen LDAP-Verzeichnisbaum

Wird nun also eine Anfrage nach allen Augmented Areas gestellt, die Dienste eines bestimmten Nexus-Typs (oder seiner Subtypen) bieten und sich mit einer bestimmten (durch ein Polygon definierten) Fläche überlappen, so läuft die Bearbeitung in 2 Schritten ab. Zuerst wird der *NexusTypeNode* für diesen Nexus-Typ ermittelt. Diese Operation wird vom System unterstützt, da es die Definition von Indizes auf Attributen erlaubt. Der distinguished name (DN) dieses Knotens wird dann als Einstiegspunkt für die den 2. Teil der Anfrage an den LDAP-Server verwendet. Da das LDAP-System allerdings nur eindimensionale Indizes vorsieht, müssen bei dieser Anfrage nun alle *NexusAreaNodes* in einer Filterfunktion (*nexusAreaOverlaps*, s.u.) einzeln überprüft werden.

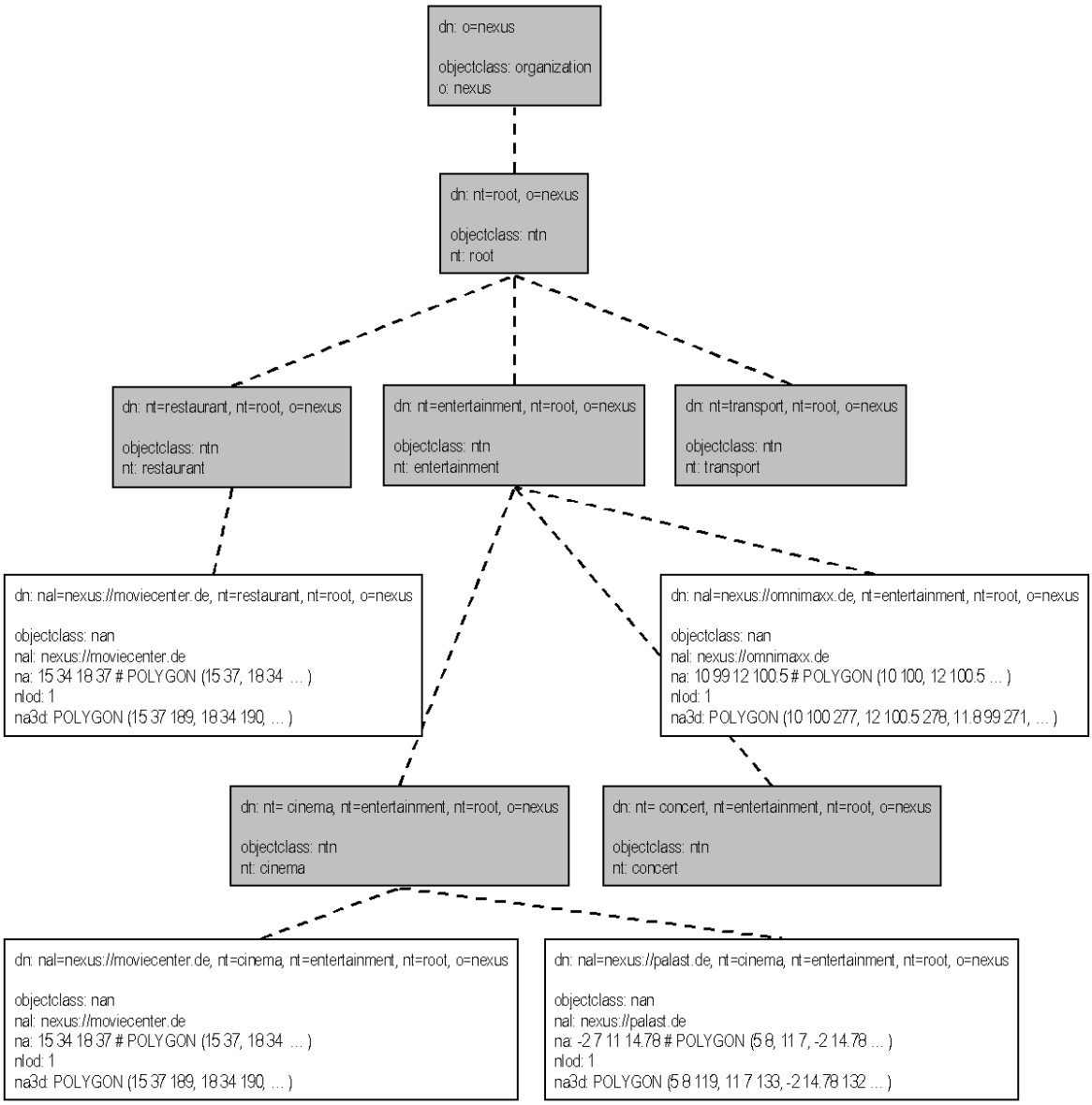

Abb. 6: Aufbau des Directory Information Tree (DIT)

3.2 LDAP-Server

Zur Realisierung des räumlichen Verzeichnisses auf der Basis eines LDAP-Systems müssen zwei Probleme gelöst werden. Zuerst muss der LDAP-Server, der besonders für die Verwaltung von textorientierten Verzeichnissen wie z.B. Telefonverzeichnissen, Benutzerverwaltungen, Inventarlisten usw. ausgelegt ist, um die Fähigkeit zur Verarbeitung von räumlichen (geometrischen) Informationen erweitert werden.

Dazu muss eine neue Syntax *(NexusAreaSyntax)* für die Verarbeitung von Polygonen eingeführt werden, die dann für das Flächenattribut der *NexusAreaNodes* verwendet

wird. Hinzu kommt eine Vergleichsoperation *(nexusAreaOverlaps)*, die die Überlappung der gespeicherten Fläche mit dem Anfrage-Polygon überprüft. Zur Performanzsteigerung des Systems soll dabei eine optimierte Darstellung der Polygone verwendet werden, die einen möglichst einfachen Test auf mögliche Überlappung erlaubt.

Die Erweiterung um eine neue Syntax erfolgt nicht durch eine Veränderung des Quelltextes, sondern durch ein Modul, das beim Start des LDAP-Servers geladen wird und die Registrierung der neuen Syntax und der darauf definierten Operationen (*Matching Rules*) übernimmt. Auf dieser Basis können dann die neuen Attributtypen und Objektklassen zur Speicherung der Einträge *(NexusTypeNode* bzw. *NexusAreaNode)* angelegt werden.

3.3 LDAP-Client

In einem zweiten Schritt müssen für den LDAP-Client geeignete Funktionen zum einfachen Zugriff auf die Daten entwickelt werden, die gleichzeitig die im Kapitel 3.2 beschriebene Organisationsstruktur des Verzeichnisses sicherstellen, da ein LDAP-System keine Regeln definiert, an welcher Stelle Knoten eingefügt werden dürfen[5].

Dies sind – in Anlehnung an [31] – Funktionen zur zur Verwaltung der Typhierarchie:

- **createType** - fügt einen neuen Nexus-Typ in die Hierarchie ein.
- **dropType** - löscht einen Eintrag aus der Typhierarchie.
- **listTypes** - erzeugt eine Liste aller Einträge der Typhierarchie.

und Funktionen zur Verwaltung und Abfrage der Augmented Areas:

- **insertArea** - fügt eine neue Augmented Area in das Verzeichnis ein.
- **updateArea** - ändert die Attributwerte für einen Eintrag.
- **deleteArea** - löscht einen Eintrag aus dem Verzeichnis.
- **query** - liefert eine Liste von Einträgen, die den übergebenen Merkmalen entsprechen.

Vergleichsoperationen bei Anfragen an das LDAP-System erfolgen durch Filterfunktionen, die nur den gespeicherten Attributwert und den bei der Anfrage angegebenen Vergleichswert erhalten. Insbesondere ist innerhalb dieser Filterfunktionen kein Zugriff auf andere Attribute eines Eintrags möglich, die z.B. eine optimierte Darstellung enthalten könnten. Daher muss LDAP-intern eine Darstellung der Flächen gewählt werden, die gleichzeitig den im vorigen Kapitel geforderten einfachen Test und die genaue Prüfung auf Überlappung erlaubt.

Damit diese Darstellung nicht bei jedem Zugriff auf das Objekt erneut erzeugt werden muss, erfolgt diese Umsetzung bereits im LDAP-Client. (Funktion *optimizeAreaString*)

5) Bei einem neuen Knoten muss nur der Vaterknoten existieren und der relative dinstinguished name (RDN) eindeutig sein.

4. Feinentwurf und Realisierung

Der grobe Systemüberblick (vgl. Abb. 3) lässt sich weiter detaillieren. Dabei zeigen sich 4 logisch voneinander getrennte Schichten (vgl. Abb. 7). Da der LDAP-Server direkt über Bibliotheksfunktionen auf die Datenbank zugreift, müssen der LDAP-Server und die Backend-DB auf dem selben Rechner laufen. Dies gilt auch für die Verbindung zwischen Nexus-Anwendung und Nexus-LDAP-Client. Deshalb ist bei der gewählten Implementierung des Systems nur zwischen LDAP-Client und LDAP-Server eine Rechnergrenze möglich, da zwischen diesen beiden Komponenten die Kommunikation per LDAP-Kommandos über TCP/IP stattfindet.

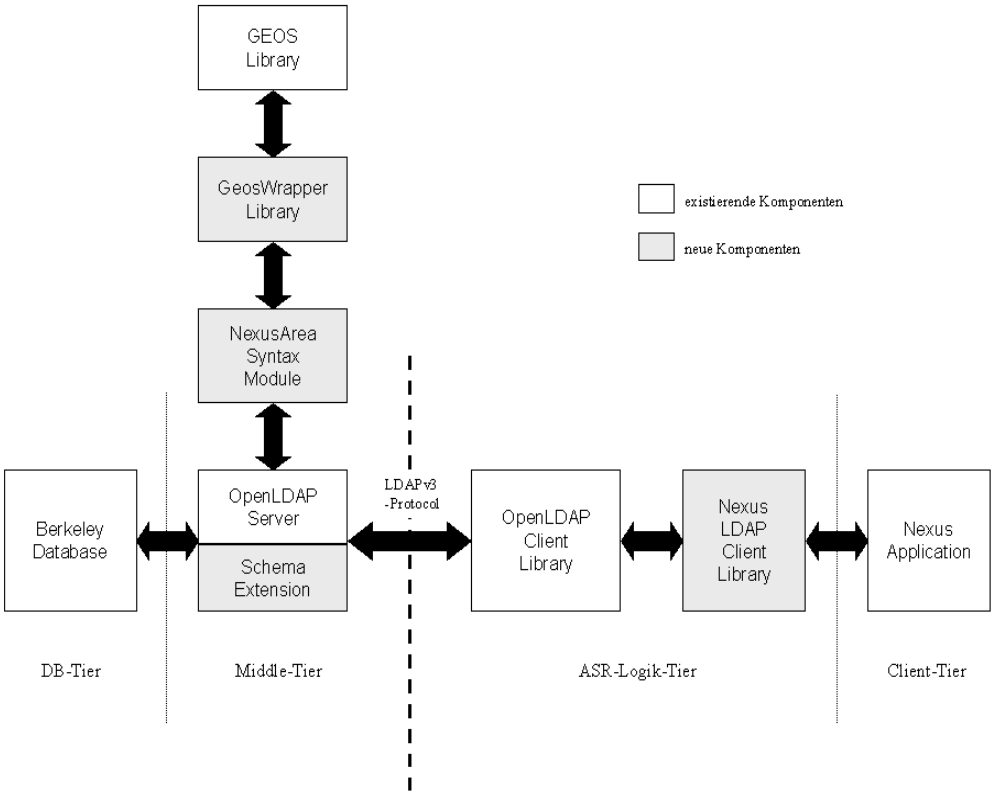

Abb. 7: Detaillierte Darstellung des Systemaufbaus

Wie in der Darstellung erkennbar, müssen 4 Komponenten im Rahmen dieses Projektes entwickelt werden, deren Aufbau in den folgenden Abschnitten erläutert wird.

4.1 Schema-Erweiterung des LDAP-Servers

Im LDAP-Standard ist eine Anzahl von Klassen und Attribute angegeben, die zur Speicherung häufig verwendeter Informationen (Adressen, Telefonnummern usw.) verwendet werden können. Da dies zum Aufbau des räumlichen Verzeichnisses aber nicht ausreicht, muss das Schema, in dem der Aufbau dieser Standardklassen für den LDAP-Server beschrieben wird, erweitert werden. Dies erfolgt durch Ergänzung der Konfigurationsdatei, die beim Start der LDAP-Servers angegeben wird.

Jeder Eintrag in das Verzeichnis folgt einem Aufbau, der durch die Definition einer Objektklasse vorgeben ist. In dieser Objektklasse wird festgelegt, welche Attribute ein Eintrag enthält und ob diese Attribute optional oder obligatorisch sind. Die Festlegung erfolgt durch die Aufzählung von Attributtypen. Diese Attributtypen entsprechen einer Kombination von Variablennamen und Typbezeichern: sie beschreiben nicht nur, unter welchem Bezeichner auf den Attributwert zugegriffen werden kann, sondern legen gleichzeitig das genaue Format des Attributs fest, indem sie die verwendete Syntax (die in etwa dem Typ in einer Programmiersprache entspricht) und die auf dieses Attribut anwendbaren Operationen angeben. Aus diesem Grund darf ein Attributtyp auch nicht mehrfach in einer Objektklassendefinition auftreten. Sollen also z.B. Vor- und Nachname einer Person gespeichert werden, müssen dafür 2 Attributtypen definiert werden, auch wenn sie genau dem gleichen Format folgen.

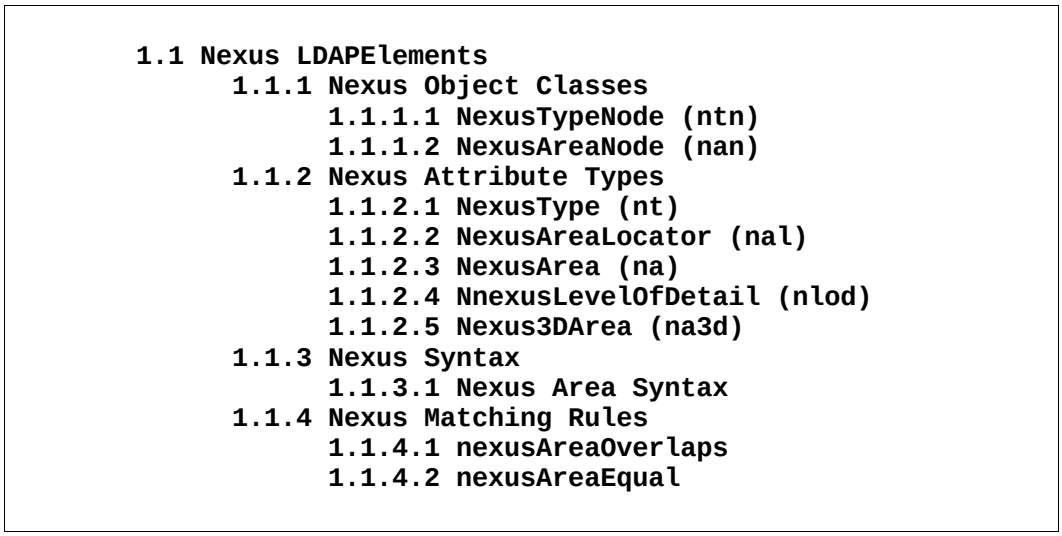

Abb. 8: Überblick über das Nummerierungsschema

Allerdings erlaubt es der LDAP-Standard, mehrere Werte für ein Attribut anzugeben und gleichberechtigt zu speichern. So können z.B. mehrere eMail-Adressen für eine Person in nur einem Attribut gespeichert werden. Bei allen für dieses Projekt deklarierten Attributtypen wird aber durch die Angabe der Kennzeichnung **`SINGLE-VALUE`** festgelegt, dass jeder Eintrag nur einen Attributwert besitzt.

Muss allerdings eine neue Syntax definiert werden, da die vordefinierten Typen nicht ausreichen (so sieht der LDAP-Standard nicht einmal einen Typ für reelle Zahlen vor), ist ein tieferer Eingriff in das System notwendig (vgl. Kap. 4.2).

Alle neu definierten Elemente (Objektklassen, Attributtypen, Syntaxerweiterungen, Vergleichsoperationen) des LDAP-Servers benötigen bei ihrer Deklaration einen *Object Identifier* (*OID*) zur eindeutigen Identifizierung. Diese Notwendigkeit ergibt sich aus der engen Verwandschaft von LDAP zu X.500. Der LDAP-Standard erlaubt zusätzlich noch die Definition eines kurzen Namens, der vom LDAP-Client für den Server-Zugriff verwendet werden kann. Die Verwendung und Kenntnis dieser Nummern ist also nur für die interne Konfiguration des LDAP-Servers notwendig.

In dieser Studienarbeit wird dazu ein Unterbaum des für private Experimente ausdrücklich freigegebenen Teilbaums „1.1“ verwendet (vgl. Abb. 8). Bei einem tatsächlichen Einsatz sollten diese Nummern dann durch registrierte Werte ersetzt werden[6] [29]. In dieser Übersicht sind gleichzeitig neben den ausführlichen Bezeichnern auch in Klammern die Kurznamen[7] (z.b. *nlod* für *NexusLevelOfDetail*) angegeben.

4.1.1 Neue Objektklassen

Für das Speichern der Knoten muss das LDAP-Schema um 2 neue Objektklassen erweitert werden. Eine Anleitung zur Vorgehensweise findet sich z.B. in [29] oder [30]. Die Beschreibung der Objektklassen folgt der Syntax, wie sie im LDAPv3-Standard beschrieben ist [2]. Die so definierten Objektklassen werden dann beim Start des LDAP-Servers als Schemaerweiterung durch die Konfigurationsdatei des LDAP-Servers geladen (vgl. Anhang A.2.2).

NexusTypeNode

Die Knoten, in die die Nexus-Typen eingetragen werden, besitzen außer ihrem Klassenbezeichner *NexusType* keine weiteren Attribute.

NexusTypeNode	
NexusType	: String

Abb. 9: Klassenschema - NexusTypeNode

6) Eine Registierung kann z.B. entweder bei der Internet Assigned Numbers Authoriy (IANA) oder einer nationalen Behörde erfolgen.

7) Die Verwendung derartig kurzer Bezeichner wurde aus Effizienzgründen gewählt, da jeder Attributwert durch Vorstellen des zugehörigen Attributtyps gekennzeichnet wird. Insbesondere würden so auch die entstehenden *distiniguished names* überproportional länger, da auch in diesem Fall für jede Ebene des Verzeichnisbaums der Attributtyp des *relative distinguished names* als Präfix übernommen wird.

Aus diesem Klassendiagramm leitet sich die entsprechende Deklaration der Objektklasse für den LDAP-Server ab:
(Ausschnitt aus der Datei: **conf/nexus_slapd.schema**)

```
objectclass (
      1.1.1.1
      NAME 'ntn'
      DESC 'NexusTypeNode'
      MUST nt)
```

Durch dieses Deklaration wird die neue Objektklasse mit dem Kurznamen „ntn“ und dem OID „1.1.1.1“ eingeführt. Das einzige Attribut *NexusType* (kurz: *nt*) wird gleichzeitig von den Verwaltungsoperationen auch als *relative distinguished name (RND)* für den Typknoten verwendet.[8] Dieses Attribut ist als Pflichtattribut (**MUST nt**) definiert, daher wird die Existenz bei Anlage oder Änderung eines Eintrags durch das System überprüft. Der dabei verwendete Typ (String) ergibt sich sich durch die Definition des Attributtyps (s.u.).

NexusAreaNode

Die Knoten, in die die einzelnen Augmented Areas eingetragen werden, besitzen dagegen weitere Attribute, deren genauer Aufbau und Bedeutung im nachfolgenden Abschnitt (Kap. 4.1.2) erläutert wird.

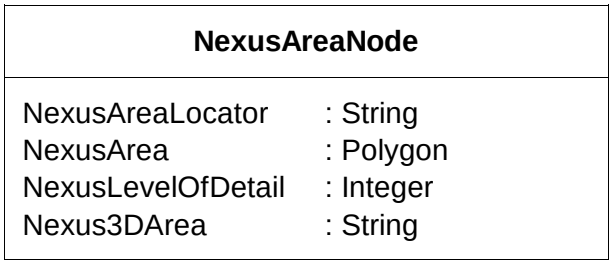

Abb. 10: Klassenschema - NexusAreaNode

Dies entspricht folgender Schemadefinition:
(Ausschnitt aus der Datei: **conf/nexus_slapd.schema**)

```
objectclass (
      1.1.1.2
      NAME 'nan'
      DESC 'NexusAreaNode'
      MUST ( nal $ na $ nlod $ na3d ) )
```

Alle Attribute sind als Pflichtattribute definiert. Für den *relative distinguished name* (RDN) eines *NexusAreaNodes* wird der *NexusAreaLocator* (kurz: *nal*) verwendet, da dieser als einziges Attribut für alle *NexusAreaNodes* unterhalb eines Typknotens eindeutig ist. Diese Entscheidung kann aber bei der Definition der Objektklasse nicht

8) Dabei wird jeweils die vollständige Bezeichnung des Nexus-Typs in den DN übernommen, es findet keine Umsetzung in eine kürzere, codierte Form (z.B. „restaurant“-> T001, ...) statt.

angegeben werden und wird vom LDAP-Server auch nicht erzwungen[9]. Diese Festlegung wird vielmehr vom NexusLDAPClient (vgl. Kap. 4.4) beim Einfügen neuer Einträge in das Verzeichnis umgesetzt.

Da sich die Suchoperationen auf diese Festlegung und den damit definierten Verzeichnisbaumaufbau verlassen, darf auch nicht mit anderen LDAP-fähigen Anwendungen schreibend auf das Verzeichnis zugegriffen werden, da sonst Fehler auftreten können.

4.1.2 Neue Attributtypen

Die Neudefinition von Attributtypen läuft entsprechend ab und wird mit den Objektklassen beim Start des LDAP-Servers geladen.

NexusType

Der *NexusType* (kurz: *nt)* ist die verzeichnisweit eindeutige Bezeichnung des Nexus-Typs innerhalb der Typhierarchie. Die Speicherung erfolgt als ASCII-String (IA5String), dies erfolgt durch die Zeile „SYNTAX 1.3.6.1.4.1.1466.115.121.1.26“, wobei diese komplexe OID dem Typ ASCII-String entspricht. An dieser Stelle ist eine Verwendung eines klareren Syntax-Namens leider nicht erlaubt. Eine Angabe oder Beschränkung der Länge ist beim OpenLDAP-System nicht notwendig. Die Deklaration erfolgt durch folgenden Eintrag: (Ausschnitt aus der Datei: **`conf/nexus_slapd.schema`**)

```
attributetype (
      1.1.2.1
      NAME 'nt'
      DESC 'NexusType'
      SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
      EQUALITY caseIgnoreMatch
      SUBSTR caseIgnoreSubstringsMatch
      SINGE-VALUE )
```

Dabei ist festgelegt, dass Vergleichsoperationen ohne Beachtung der Groß- und Kleinschreibung durchgeführt werden (**`caseIgnoreMatch)`**. Außerdem darf ein Verzeichniseintrag nur einen Wert besitzen (**`SINGLE-VALUE`**), denn prinzipiell unterstützen LDAP-Verzeichnisse auch mehrere Werte für ein Attribut. Da jedoch der Aufbau des Verzeichnisbaums nach dem Nexus-Typ erfolgen soll, muss dieser für jeden Eintrag eindeutig sein.

NexusAreaLocator

Der *NexusAreaLocator* (kurz: *nal*) ist die Adresse des zuständigen Spatial Model Servers. Format und Aufbau der Adresse werden allerdings nicht überprüft, sie wird wie der *NexusType* als String ohne Längenfestlegung gespeichert.
(Ausschnitt aus der Datei: **`conf/nexus_slapd.schema`**)

9) Der RDN eines Eintrags muss nur – und dies wird vom LDAP-Server überprüft – aus einem Attribut des Eintrags gebildet werden und für alle direkten Nachfolgeknoten eines Eintrags eindeutig sein.

```
attributetype (
      1.1.2.2
      NAME 'nal'
      DESC 'NexusAreaLocator'
      SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
      EQUALITY caseIgnoreMatch
      SUBSTR caseIgnoreSubstringsMatch
      SINGLE-VALUE )
```

NexusArea

Die *NexusArea* (kurz: *na*) ist das Gebiet, für das der Service definiert ist. Dieses Gebiet ist dabei als Polygon wird im Format der neuen *NexusAreaSyntax* gespeichert (vgl. Kap. 4.2), dies erfolgt durch die Angabe der OID „1.1.3.1“ für die Syntax.
(Ausschnitt aus der Datei: **`conf/nexus_slapd.schema`**)

```
attributetype (
      1.1.2.3
      NAME 'na'
      DESC 'NexusArea'
      SYNTAX 1.1.3.1
      EQUALITY nexusAreaEqual
      SINGLE-VALUE )
```

In diesem Attribut wird die optimierte Darstellung der Fläche gespeichert, die für den Test auf Überlappung mit dem Anfragegebiet verwendet wird.

NexusLevelOfDetail

Der *NexusLevelOfDetail* (kurz: *nlod*) gibt den Detaillierungsgrad des angebotenen Services an und wird als Integerwert[10] (**`1.3.6.1.4.1.1466.115.121.1.27`**) gespeichert. (Ausschnitt aus der Datei: **`conf/nexus_slapd.schema`**)

```
attributetype (
      1.1.2.4
      NAME 'nlod'
      DESC 'NexusLevelOfDetail'
      SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
      EQUALITY integerMatch
      ORDERING integerOrderingMatch
      SINGLE-VALUE )
```

Wichtig ist hier auch die Angabe der *MatchingRule* **`integerOrderingMatch`**, da sonst kein größer- bzw. kleiner-Vergleich auf diesem Attribut durchgeführt werden kann.

Nexus3DArea

In der *Nexus3DArea* (kurz: *na3d*) wird die ursprüngliche und nicht optimierte Darstellung der *NexusArea* gespeichert. Dies ist notwendig, da sich die 3-dimensionale Darstellung nicht wieder aus der optimierten Darstellung gewinnen lässt. Bei Anfragen an das räumliche Verzeichnis wird dann dieses Attribut zurückgeliefert. Da aber auf

10) Die interne Speicherung und Verarbeitung von Integerwerten im OpenLDAP-Server erfolgt dabei allerdings auch als Strings.

diesem Attribut keinerlei geometrische Vergleichsoperationen durchgeführt werden, wird es, genauso wie z.B. der *NexusAreaLocator*, als String definiert. Es wird auch keine Überprüfung der Korrektheit des WKT-Strings durchgeführt:
(Ausschnitt aus der Datei: **`conf/nexus_slapd.schema`**)

```
attributetype (
      1.1.2.5
      NAME 'na3d'
      DESC 'Nexus3DArea'
      EQUALITY caseIgnoreMatch
      SUBSTR caseIgnoreSubstringsMatch
      SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
      SINGLE-VALUE )
```

4.2 Syntaxerweiterung

Zur Speicherung und Verarbeitung der NexusArea muss eine neue Syntax im LDAP-Server definiert werden. Diese Erweiterung ist nicht wie die Einführung neuer Objektklassen und Attributtypen durch deren Angabe in der Konfigurationsdatei möglich. Dies ist nur durch Eingriff in den Code des LDAP-Servers oder durch Erstellen eines dynamisch ladbaren Moduls möglich, das beim Start des Systems geladen wird und die neue Syntax und darauf definierte Operationen registriert.

Wird durch die Direktive „**`moduleload`**“ in der Konfigurationsdatei ein zu ladendes Modul definiert, so sucht das System nach der Funktion **`init_module`** (erweitert um den Namen des Moduls und **`_LTX_`**, in diesem Fall also nach der Funktion **`nexusAreaSyntax_LTX_init_module`**) und führt diese aus. In dieser Funktion wird dann durch Aufruf der LDAP-Server-Funktionen **`register_syntax`** bzw. **`register_matching_rule`** die eigentliche Registrierung durchgeführt.

4.2.1 Neue Syntax

NexusAreaSyntax

Eigentlich ist die NexusArea als Polygon in 3-dimensionalen Koordinaten (Längengrad, Breitengrad und Höhe über Geoid) gemäß dem World Geodetic System 1984 (WGS 84)[11] definiert. Zur Beschleunigung des Systems wird aber eine optimierte Darstellung gewählt. Die Umrechnung erfolgt nicht im LDAP-Server, sondern muss bereits vom LDAP-Client durchgeführt werden (vgl. Kap. 3.3).

Die optimierte Darstellung enthält nur noch 2-dimensionale Koordinaten (Längen- und Breitengrad), die direkt der GEOS-Library[12] zur Verarbeitung übergeben werden können. Außerdem stehen vor der Well-Known-Text (WKT) Darstellung des Polygons die minimalen und maximalen Koordinaten des Polygons, sodass vor der genauen und

11) http://www.wgs84.com

12) Die GEOS-Library unterstützt nur 2-dimensionale WKT-Strings.

aufwändigen exakten Prüfung der Polygone auf Überlappung durch einfachen Vergleich der Bounding-Boxes überprüft werden kann, ob eine Überlappung überhaupt möglich ist.

Daher ist der gesamte Eintrag wie folgt aufgebaut – die Darstellung des Polygon folgt dabei einem Ausschnitt der OGIS[13] Well-Known Text (WKT) Spezifikation [32]:

```
<NexusArea> := <BoundingBox> # <Polygon Tagged Text>

  <BoundingBox> := <min x> <min y> <max x> <max y>

    <min x> := double precision literal
    <min y> := double precision literal
    <max x> := double precision literal
    <max y> := double precision literal

  <Polygon Tagged Text> := POLYGON <Polygon Text>

    <Polygon Text> := EMPTY
      | ( <LineString Text> { , <LineString Text> }* )

    <LineString Text> := EMPTY | ( <Point> { , <Point> }* )

    <Point> := <x> <y>

    <x> := double precision literal
    <y> := double precision literal
```

Ein einfaches Polygon ohne Löcher könnte in optimierter Darstellung somit wie folgt aussehen[14]:

```
-1.28 -23.2 10.5 20.2 # POLYGON ( ( 10.5 20.2, 5 -23.2,
-1.28 2, 10.5 20.2 ) )
```

Diese Syntax-Darstellung kann allerdings nicht dem LDAP-System übergeben werden, die Prüfung des korrekten Aufbaus erfolgt durch die Funktion **`nexusAreaSyntaxValidate`**, die bei der Registrierung der neuen Syntax angeben wird. Diese Funktion überprüft, ob die <BoundingBox>[15] der obigen Darstellung entspricht und der <Polygon Tagged Text> von der GEOS-Bibliothek erfolgreich verarbeitet werden kann, also eine korrekte Polygondarstellung enthält. Dazu wird die Funktion **`gw_testPolygonString`** der *GeosWrapper*-Library aufgerufen.

Die Registrierung der neuen Syntax erfolgt durch die Übergabe der folgenden Parameter an die LDAP-Serverfunktion **`register_syntax(..)`**:
(vgl. Datei „**`src/nexusAreaSyntax/nexusAreaSyntax.h`**“)

```
{"( 1.1.3.1 DESC 'Nexus Area' )" , 0,
nexusAreaSyntaxValidate, NULL, NULL}
```

13) http://www.opengis.org

14) Der letzte Punkt eines Polygons muss immer gleich dem 1. Punkt sein.

15) Es wird aber nicht geprüft, ob die BoundingBox geometrisch korrekt ist, also tatsächlich das Polygon umschließt.

Dadurch wird eine neue Syntax mit der OID „1.1.3.1“ eingeführt, deren korrekte Darstellung durch die C-Funktion **`nexusAreaSyntaxValidate`** kontrolliert werden soll. Hier wird auch deutlich, dass neben der Beschreibung „Nexus Area“ kein Namen für die Syntax angegeben werden kann, der dann bei der Definition von Attributtypen statt der OID verwendet werde könnte.

In den durch „**`NULL`**“ belegten Feldern könnten noch Funktionen zur Normalisierung oder zur Umwandlung in eine (vom Menschen) lesbare Darstellung angegeben werden, dies ist allerdings bei dieser Syntax nicht notwendig.

4.2.2 Neue Vergleichsregeln (Matching Rules)

Um auf dieser neu eingeführten Syntax auch Operationen durchführen zu können, müssen nach der Registrierung der Syntax noch *Matching Rules* beim LDAP-Server registriert werden.

Normalerweise sind nach dem LDAP-Protokoll nur eine kleine Menge von Vergleichsoperationen definiert und bei der Anfrage anzugeben (exakter Vergleich „**`=`**“, Teilstring „**`= ..*..`**“, ähnlich „**`~=`**“, kleiner gleich „**`<=`**“, größer gleich“**`>=`**“, existiert „**`=*`**“ sowie die boolschen Operatoren AND, OR und NOT). Zwar wäre es möglich, durch „Missbrauch“ einer dieser Operatoren den benötigten Überlappungstest einzuführen, seit der Version 3 des LDAP-Standards (LDAPv3) von 1997 ist allerdings auch eine neue Operation *extensibleMatch* definiert, die durch beliebige Funktionen definiert werden kann. Voraussetzung für die Registrierung einer *extensibleMatch*-Funktion ist allerdings die Existenz einer Standard-Vergleichsoperation für die entsprechende Syntax.

Daher wird neben der für das Projekt notwendigen Funktion **`nexusAreaOverlaps`** auch noch die Funktion **`nexusAreaEqual`** eingeführt, die sonst nicht verwendet wird.

nexusAreaOverlaps

Bei der Regristierung der neuen *Matching Rule* wird die C-Funktion **`nexusMRAreaOverlaps`** angeben, die vom LDAP-System zur Durchführung eines Vergleichs aufgerufen wird. Bei einer Anfrage an den LDAP-Server wird die Erfüllung eines Suchfilters nicht von der Datenbank überprüft. Diese liefert vielmehr aufgrund der Baumstruktur mögliche Verzeichniseinträge an den LDAP-Server. Diese übergibt dann den Attributwert des zu untersuchenden Verzeichniseintrages und den Vergleichswert, der bei der Anfrage angegeben wurde, an eine Filterfunktion (in diesem Fall **`nexusMRAreaOverlaps`**). Diese führt die eigentliche Vergleichsoperation durch und meldet dem LDAP-Server das Ergebnis des Vergleichs zurück.

In dieser Funktion **`nexusMRAreaOverlaps`** wird der Test auf die geometrische Überlappung der NexusArea eines Verzeichniseintrages mit dem Anfragepolygon durchgeführt. Dazu werden zunächst nur die Minimal und Maximal-Koordinaten der beiden Flächen (Bounding Box) eingelesen und überprüft, ob eine Überlappung überhaupt möglich ist. Ist dies der Fall, so werden die beiden Strings mit der

Polygondarstellung an die Funktion *gw_testPolygonOverlap* der *GeosWrapper*-Library übergeben, die den exakten Vergleich durchführt.

Die Registrierung erfolgt somit durch die Übergabe der folgenden Parameter an die LDAP-Serverfunktion **register_matching_rule(..)**:
(vgl. Datei „**src/nexusAreaSyntax/nexusAreaSyntax.h**“)

```
{"( 1.1.4.1 NAME 'nexusAreaOverlaps' SYNTAX 1.1.3.1 )",
      SLAP_MR_EXT, NULL, NULL, NULL,
      nexusMRAreaOverlaps, NULL, NULL, NULL}
```

Auf diese Weise wird die neue Operation mit der OID „1.1.4.1“ eingeführt, die auf der Syntax „Nexus Area Syntax“ (OID 1.1.3.1) definiert ist und durch Aufruf der C-Funktion **nexusMRAreaOverlaps** ausgeführt wird. In diesem Fall ist – im Gegensatz zur Syntaxregistrierung – auch wieder ein Name „nexusAreaOverlaps“ angegeben, der vom LDAP-Client statt der unübersichtlicheren OID verwendet werden kann.

nexusAreaEqual

Diese Funktion führt einen Vergleich zwischen Verzeichniseintrag und Anfragewert durch byteweisen Vergleich der Strings durch. Es wird also nicht auf eine topologische Übereinstimmung geprüft, sondern ob die Darstellung der Polygone exakt übereinstimmt. Die Durchführung über nimmt die C-Funktion **nexusMRAreaEqual**. Die bestehende String-Vergleichsregel **caseIgnoreMatch,** wie sie z.B. beim Attribut NexusType benutzt wird, kann hier nicht verwendet werden, da ihre Anwendung auf eine neue Syntax nicht möglich ist.

Die Registrierung erfolgt wieder durch Übergabe der folgenden Parameter an die LDAP-Serverfunktion **register_matching_rule(..)**:
(vgl. Datei „**src/nexusAreaSyntax/nexusAreaSyntax.h**“)

```
{"( 1.1.4.2 NAME 'nexusAreaEqual' SYNTAX 1.1.3.1 )",
      SLAP_MR_EQUALITY, NULL, NULL, NULL,
      nexusMRAreaEqual, NULL, NULL, NULL}
```

4.3 Bibliothek zur Kapselung der GEOS-Library

Diese *GeosWrapper*-Library ist nur aus technischen Gründen notwendig, da der OpenLDAP-Server in C, die GEOS-Library aber in C++ implementiert ist und gleichzeitig keine C-API bereitstellt. Wegen der unterschiedlichen Symbolcodierung von C und C++-Compilern darf das NexusAreaSyntax-Modul nicht in C++ implementiert werden, da von diesem Modul auch Aufrufe auf die Registrierungsfunktionen des LDAP-Servers notwendig sind, die vom Linker nicht erfolgreich aufgelöst werden können.

Daher wird, da nur wenige Funktionen der GEOS-Library benötigt werden, mittels dieser Wrapper-Bibliothek ein C-Zugriff auf die GEOS-Library realisiert. Diese Bibliothek muss, um auf die GEOS-Library zuzugreifen, in C++ geschrieben werden, exportiert ihre Symbole aber in der C-Aufrufsyntax, sodass vom in C geschriebenen NexusAreaSyntax-Modul darauf zugegriffen werden kann.

Zu beachten ist, dass die GEOS-Library mit ebenen kartesischen 2-dimensionalen Koordinaten arbeitet, die NexusArea allerdings in geodätischen Koordinaten (Längen- und Breitengrad) definiert ist. Daher müssten die geodätischen Koordinaten eigentlich in eine entsprechende ebene Darstellung, z.B. Gauss-Krüger-Koordinaten, transformiert werden. Allerdings kann jede Projektion von geodätischen auf ebene Koordinaten nur bestimmte Eigenschaften der Informationen erhalten, z.B. Flächen- oder Winkeltreue.

Da bei diesem Projekt nur Polygone auf Überlappung getestet werden, die außerdem von der Ausdehnung eher klein sind (z.B. Gebäude) und keine Informationen über die tatsächliche Entfernung auf der Erde benötigt werden, wird auf eine komplizierte Projektion der Koordinaten verzichtet. Der Längengrad wird als x-Wert, der Breitengrad wird ohne weitere Umrechnungen als y-Wert verwendet. Unter der Voraussetzung, dass die betrachteten Flächen eher klein sind und in normalen Breiten (z.B. Deutschland) liegen, sollten dadurch keine Fehler auftreten. Eine gesonderte Behandlung von Pol-nahen Koordinaten oder Flächen, die den 180°-Meridian schneiden, findet nicht statt.

Die *GeosWrapper*-Library stellt nur 2 Funktionen zur Verfügung:

int **gw_testPolygonString**(const char *wkt)

Diese Funktion prüft, ob der übergebene WKT-String erfolgreich in ein Polygon umgewandelt werden kann. Als Ergebnis liefert die Funktion **`1`** (true) für die erfolgreiche Durchführung oder **`0`** (false), wenn bei der Umwandlung Fehler aufgetreten sind.

int **gw_testPolygonOverlap**(const char *wktA, const char *wktB)

Diese Funktion prüft, ob 2 durch WKT-Strings definierte Polygone sich überlappen. Sie verwendet dazu die Funktion **`intersects`** der Klasse **`polygon`**. Diese testet, ob die Flächen zweier Polygone nicht disjunkt sind, wobei sich sich die Ränder nicht notwendigerweise schneiden. Die eigentlich naheliegende Funktion **`overlaps`** würde dagegen auf Überdeckung prüfen.

Als Ergebnis liefert die Funktion **`1`** (true), wenn sich die beiden Polygon überlappen. bzw. **`0`** (false), wenn dies nicht der Fall ist oder bei der Überlappungsprüfung Fehler aufgetreten sind.

4.4 Funktionen zur Abfrage des LDAP-Servers

Damit eine Anwendung einfach auf die im LDAP-Server gespeicherten Informationen zugreifen kann, werden mit dieser Bibliothek, dem *nexusLDAPClient* eine Anzahl von Funktionen bereitgestellt, die die notwendigen Operationen durchführen und dabei gewährleisten, dass der beschriebene Aufbau des räumlichen Verzeichnisses erhalten bleibt.

Denn prinzipiell kann mit allen Programmen, die das LDAPv3-Protokoll beherrschen, so z.B. die Kommandozeilen-Werkzeuge **`ldapadd, ldapsearch, ldapmodify`** .., auf das Verzeichnis zugegriffen werden. Während dies bei Abfrage-Operationen noch unkritisch ist, kann durch Änderungsoperationen der in Kap. 3.1 beschriebene Aufbau des

Verzeichnisbaums verletzt werden, da der LDAP-Server nicht die Einhaltung der Regeln überwachen kann. Allerdings ist in der Konfiguration der Zugriffsrechte für den LDAP-Server festgelegt, dass nur autorisierte Benutzer Änderungsoperationen durchführen dürfen, während für den lesenden Zugriff kein Passwort notwendig ist.

Daher muss auch bei allen Änderungsoperationen ein Benutzername **`user`** und das zugeordnete Passwort **`passwd`** angegeben werden. Diese Informationen werden vom LDAP-Server verwaltet und geprüft.[16]

4.4.1 Funktionen zur Verwaltung der Typ-Hierarchie

int **createType**(const char *user, const char *passwd, const char *newType, const char* superType)

Mit dieser Funktion kann ein neuer Nexus-Typ in die Hierarchie eingefügt werden. Die Bezeichnung für den neuen Typ wird in **`newType`**, die Bezeichnung für den direkt übergeordneten Typ in **`superType`** übergeben. Dabei wird überprüft, ob der neue Typ nicht bereits im Verzeichnis vorhanden ist und der übergeordnete Typ existiert. Nur dann wird die Operation durchgeführt und der neue Nexus-Typ als neuer *NexusTypeNode* direkt unterhalb des zuvor ermittelten übergeordneten Typs in den Verzeichnisbaums eingefügt. Besitzt ein Nexus-Typ keinen Super-Typ, da er an der Wurzel des Verzeichnisbaum eingefügt werden soll, so ist für **`superType`** ein leerer String (aber nicht **`NULL`**) anzugeben. Dadurch sind auch mehrere Nexus-Root-Typen möglich.[17]

Da die Bezeichnung des Nexus-Typs vollständig in den *distinguished name* (DN) des Knotens übernommen wird und auch nicht in eine kürzere interne Darstellung umgewandelt wird, darf diese Bezeichnung gemäß der LDAPv3-Spezifikation [11-16] einige Sonderzeichen[18] nicht enthalten.

Als Ergebnis der Operation liefert die Funktion einen Wert von **`1`** (true) für die erfolgreiche Durchführung oder **`0`** (false) im Fehlerfall. Ist ein Fehler aufgetreten, wird außerdem die Fehlervariable **`errno`** auf einen der Fehlercodes gesetzt, die in der Headerdatei **`nexusLDAPClient.h`** der Bibliothek aufgeführt sind. Eine Liste der Fehlercodes und ihrer Bedeutung findet sich im Kap. 4.4.4.

int **dropType**(const char *user, const char *passwd, const char *oldType)

Durch diese Funktion wird der Nexus-Typ mit der Bezeichnung **`oldType`** aus dem Verzeichnis gelöscht. Der diesem Typ zugeordnete *NexusTypeNode* darf muss dabei ein

16) Die Benutzerverwaltung wird im Anhang A.2.4 erläutert.

17) Für das gesamte LDAP-Verzeichnis wird ein interner Wurzelknoten mit dem DN „o=nexus“ erstellt. Alle Nexus-Root-Typen werden dann als direkte Nachfolger dieses internen Wurzelknotens in den Verzeichnisbaum eingefügt.

18) Die nicht erlaubten Sonderzeichen sind: **`, + " \ < > ;`**
Außerdem darf der Typbezeichner nicht mit einem **#** beginnen oder enden.

Blatt-Knoten des Verzeichnisbaums sein. Er darf keine untergeordneten Typ-Knoten besitzen, auch dürfen diesem Typ keine Augmented Areas zugeordnet sein.

Auch diese Funktion liefert einen Wert von **`1`** (true) für die erfolgreiche Durchführung oder **`0`** (false) im Fehlerfall und setzt in diesem Fall die Fehlervariable **`errno`** auf den entsprechenden Fehlercode.

TypeEntry ***listTypes**()

Diese Funktion erzeugt eine Liste mit allen Einträgen der Typhierarchie. Als Ergebnis liefert die Funktion einen Zeiger auf ein Null-terminiertes Array von Einträgen des Typs **`TypeEntry`**. Diese Struktur besteht jeweils aus einem Zeiger auf den Bezeichner des übergeordneten Typs (**`superTypeName`**) und des aktuellen Typs (**`typeName`**). Dabei können keinerlei Aussagen über die Reihenfolge der Einträge gemacht werden. Ein Knoten an der Wurzel des Verzeichnisses erhält als superTypeName einen leeren String.

```
typedef struct {
  char     *superTypeName;
  char     *typeName;
} TypeEntry.
```

Diese Liste muss später vom aufrufenden Programm wieder gelöscht werden, damit der verbrauchte Speicher wieder freigegeben wird. Dies sollte wegen der verschachtelten Struktur der Liste durch die Bibliotheksfunktion **`freeTypeList`** erfolgen, da sonst Speicherverluste auftreten können.

Das in Abb. 5 gezeigte Beispiel einer einfachen Typhierarchie würde folgende Typliste liefern (Abb. 11):

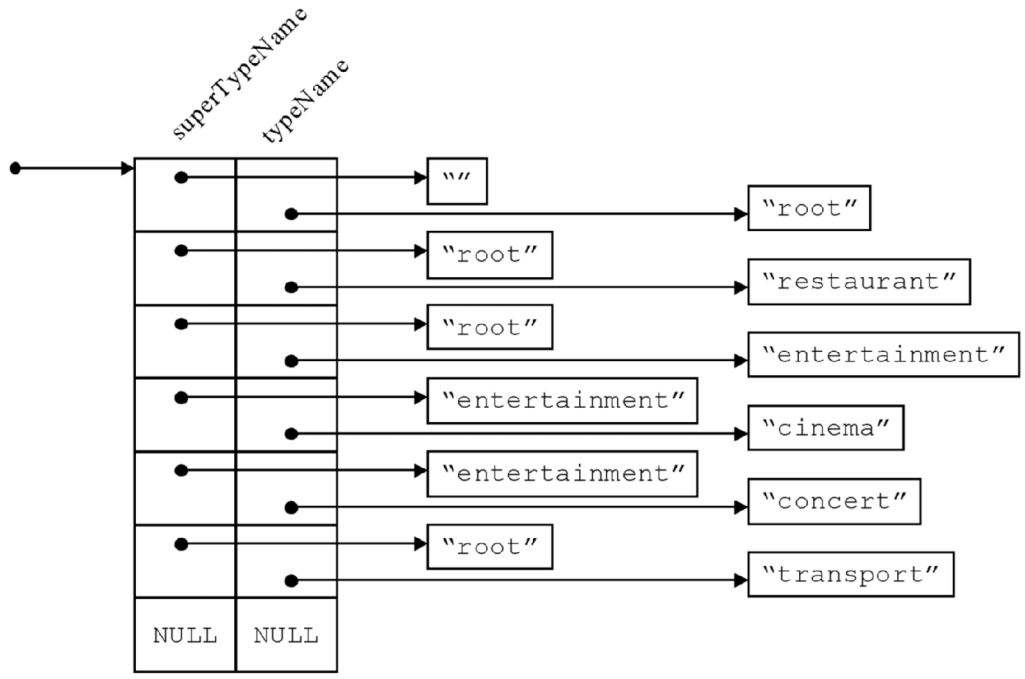

Abb. 11: Bsp. für Typliste

int **freeTypeList**(TypeEntry *typeList)

Mit dieser Funktion wird der für eine Typliste verwendete Speicherplatz wieder freigeben. Dazu werden zuerst die einzelnden Einträge, dann das Array gelöscht.

4.4.2 Funktionen zur Verwaltung des AreaServiceRegisters

Zur übersichtlicheren Verarbeitung der Angaben zu einer Augmented Area, die jeweils durch einen *NexusAreaNode* repräsentiert wird, wird in der Headerdatei **`nexusLDAPClient.h`** folgende Datenstruktur definiert:

```
typedef struct {
  char              *nexusAreaLocator;
  char              *nexusType;
  char              *nexusArea;
  int               nexusLevelOfDetail;
} AugmentedArea;
```

Diese enthält alle Angaben die beim Anlegen, Ändern oder Abfragen einer Augmented Area notwendig sind. Die Bedeutung dieser Variablen wurde bereits in Kap. 4.1.2 beschrieben.

int **insertArea**(const char *user, const char *passwd, AugmentedArea *newArea)

Diese Funktion fügt einen neuen *NexusAreaNode* in das Verzeichnis ein. Die Eigenschaften des neuen Knotens werden in der durch **`newArea`** übergebenen Struktur festgelegt. Die Position innerhalb des Verzeichnisbaums ergibt sich durch den Nexus-Typ, dem die Augmented Area zugeordnet ist, da jede Augmented Area immer ein direkter Nachfolgeknoten des entsprechenden *NexusTypeNodes* ist.

Daher wird zuerst geprüft, ob dieser Typ-Knoten existiert und der DN dieses Knotens ermittelt. Aus dem **`nexusAreaLocator`** wird der RDN des neuen Area-Knotens erstellt[19]. Dieser ergibt zusammen mit den DN des Typknotens den kompletten DN des neuen *NexusAreaNodes:*

```
dn: nt=restaurant, nt=all
dn: nal=nexus://example.de, nt=restaurant, nt=all
```

Wegen der systembedingten Eindeutigkeit des RDNs wird zuvor allerdings überprüft, ob bereits eine Augmented Area mit gleichem **`nexusAreaLocator`** direkt unterhalb des Typ-Knotens eingetragen ist. Dies ist nicht erlaubt, allerdings dürfen mehrere Augmented Areas mit gleichem **`nexusAreaLocator`** existieren, wenn sie dabei verschiedene Services (durch den nexusType definiert) anbieten, sich also an verschiedenen Positionen des Verzeichnisbaums befinden. In diesem Fall, also wenn einen Augmented Area mit mehreren Typen verknüpft ist, muss die Funktion **`insertArea`** für jeden einzelnen Nexus-Typ erneut aufgerufen werden.

19) Aus diesem Grund darf auch der nexusAreaLocator die zuvor beschriebenen Sonderzeichen nicht enthalten. (vgl. S. 34)

Auch diese Funktion liefert einen Wert von **`1`** (true) für die erfolgreiche Durchführung oder **`0`** (false) im Fehlerfall und setzt in diesem Fall die Fehlervariable **`errno`** auf den entsprechenden Fehlercode.

int **updateArea**(const char *user, const char *passwd, AugmentedArea *newArea)

Mit dieser Funktion können die Attribute eines *NexusAreaNodes* geändert werden. Dabei ist allerdings nur eine Änderung von **`nexusArea`** oder **`nexusLevelOfDetail`** möglich. Eine Änderung der anderen Attribute würde gleichzeitig auch eine Änderung der Position des Knotens innerhalb des Verzeichnisbaums erfordern und wird daher nicht unterstützt. Ist eine derartige Änderung dennoch notwendig, ist sie durch Löschen und erneutes Einfügen der Augmented Area durchzuführen.

Durch **`nexusType`** und **`nexusAreaLocator`** ist genau ein Knoten festgelegt, für den die Operation wirksam ist. Dieser wird durch Konstruktion eines Suchfilters erstellt, der dann dem LDAP-Server übergeben wird:

```
newArea->nexusType = "restaurant"
newArea->nexusAreaLocator = "nexus://example.de"

filter = (&(nt:dn:=restaurant)(nal=nexus://example.de))
```

Durch diesen Suchfilter wird definiert, dass der gesuchte Verzeichniseintrag entweder den Wert "restaurant" für das Attribut **`nt`** besitzt oder "nt=restaurant" ein Teil seines DNs ist. Außerdem (&) muss er das Attribut **`nal`** mit dem Wert "nexus://example.de" besitzen.

Anschließend werden die Attributwerte des so ermittelten Verzeichniseintrags durch die neuen Werte ersetzt.

int **deleteArea**(const char *user, const char *passwd, const char* nal)

Durch den Aufruf dieser Funktion werden alle[20] Augmented Areas gelöscht, die als *NexusAreaLocator* den durch **`nal`** übergebenen Wert besitzen. Da *NexusAreaNodes* immer Blatt-Knoten des Verzeichnisbaums sind, ist dies in jedem Fall möglich.

AugmentedArea ***query**(const char *type, const char *area, int lod_min, int lod_max)

Mit dieser Funktion können Verzeichniseinträge ermittelt werden, die einem bestimmten Profil entsprechen.

Mit dem Parameter **`type`** wird der gesuchte Nexus-Typ festgelegt. Alle Augmented Areas, die als Ergebnis geliefert werden, unterstützen Services dieses Typs oder einer seiner Untertypen gemäß der Nexus-Typhierarchie. Wegen des Aufbaus des Verzeichnisbaums definiert der DN des Typknotens, der sich aus diesem Parameter

20) Es können ja mehrere Augmented Areas mit gleichem *NexusAreaLocator* aber unterschiedlichem *NexusType* im Verzeichnis enthalten sein.

bestimmen lässt, gleichzeitig auch den Einstiegspunkt für die Suchoperation. Dadurch kann die Suche auf einen Teil des gesamten Verzeichnisses eingeschränkt und somit deutlich beschleunigt werden. Ist keine derartige Einschränkung gewünscht, so muss als Wert für **type** der String **"*"** übergeben werden, dann wird das komplette Verzeichnis durchsucht.

Mit dem Parameter **area** kann ein Suchbereich angegeben werden. Als Ergebnis werden dann nur Augmented Areas geliefert, deren *NexusArea* sich mit dieser Fläche überlappt. Kann oder soll die Suche nicht auf ein bestimmtes Gebiet beschränkt werden, um z.B. alle Augmented Areas eines bestimmten Typs zu finden, kann als Wert für **area** der String **"*"** angegeben werden, es findet dann keine Überlappungsprüfung statt.

Durch die Parameter **lod_min** bzw. **lod_max** kann ein minimaler bzw. maximaler Wert für das Attribut *NexusLevelOfDetail* festgelegt werden. Ist der Maximalwert kleiner als der Minimalwert, wird die Suche nicht eingeschränkt.

So liefert also die Anfrage:

```
query( "*", "*", 1, 0 )
```

alle Einträge des räumlichen Verzeichnisses.

Nach der Ermittlung des Einstiegspunktes für die Suche wird aus den übrigen Parametern ein Suchfilter erzeugt, der dem LDAP-Server übermittelt wird:

```
area = "POLYGON Z ( ( 10.5 20.2 600, 5 -23.2 580, -1.28 2 585,
10.5 20.2 600 ) ) "
lod_min = 1
lod_max = 4

filter
 = (&
     (na=-1.28 -23.2 10.5 20.2 # POLYGON \28 \28 10.5 20.2, 5
         -23.2 , -1.28 2 , 10.5 20.2 \29 \29)
     (nlod>=1)
     (nlod<=4)
    )
```

Dabei muss zuerst das Anfragegebiet durch die interne Funktion **optimizeAreaString** in die optimierte Darstellung umgewandelt werden. Da außerdem mehrere Sonderzeichen[21] laut LDAPv3-Spezifikation nicht in Suchfiltern enthalten sein dürfen, müssen sie vor der Übermittlung an den LDAP-Server durch entsprechende Escape-Sequenzen ersetzt werden. Dies übernimmt die interne Funktion **translateFilterString**.

Als Ergebnis liefert die Funktion **query** einen Zeiger auf eine Liste mit allen Einträgen des Verzeichnisses, die diesen Suchfilter erfüllen. Die Liste ist ein Null-terminiertes Array von Einträgen des Typs **Augmented Area** (s.o.). Diese Liste muss später vom aufrufenden Programm wieder gelöscht werden, damit der verbrauchte Speicher wieder freigegeben wird. Auch hier sollte dies wegen der verschachtelten Struktur der Liste durch die Bibliotheksfunktion **freeAreaList** erfolgen.

21) Die nicht erlaubten Sonderzeichen sind: *** ( ) \ NUL**

Das in Abb. 6 dargestellte Beispiel eines kleinen Verzeichnisbaums würde ohne einschränkende Suchparameter folgende Area-Liste liefern (Abb. 12)

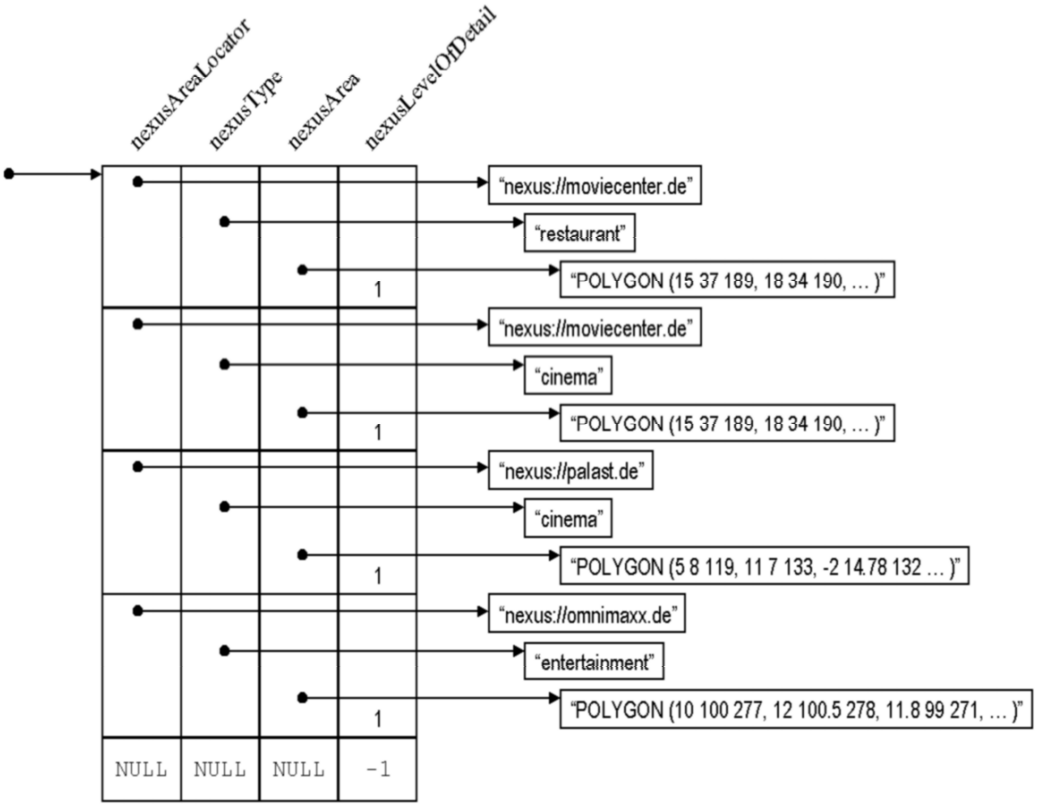

Abb. 12: Bsp. für NexusArea-Liste

Tritt bei der Erzeugung der Typliste ein Fehler auf, so liefert die Funktion **`NULL`** zurück und setzt die Fehlervariable **`errno`** auf den entsprechenden Fehlercode.

int **freeAreaList**(AugmentedArea *areaList)

Mit dieser Funktion wird der für eine NexusArea-Liste verwendete Speicherplatz wieder freigeben. Dazu werden zuerst die einzelnden Einträge, dann das Array gelöscht.

4.4.3 Interne Funktionen

Neben den bisher beschriebenen Funktionen, mit denen eine Anwendung auf das räumliche Verzeichnis zugreifen kann, gibt es noch einige interne Funktionen, die nur innerhalb der Bibliothek benötigt werden und daher auch nicht in der Header-Datei der Bibliothek deklariert werden.

int **testDNComponent**(const char *testString)

Die Funktion überprüft, ob der übergebene **`testString`** nur Zeichen enthält, die gemäß der LDAPv3-Spezifikation innerhalb eines *distinguished names* (DN) erlaubt sind. Dazu

wird getestet, ob keines der Zeichen `, + " \ < > ;` auftritt. Ist dies der Fall, so liefert die Funktion **`1`** (true), ansonsten **`0`** (false) zurück.

Diese Prüfung wird für alle Attributwerte, die auch im DN verwendet werden (also *NexusType* und *NexusAreaLocator*) durchgeführt.

Character	Value (Decimal)	Escape Sequence
Space at the beginning or end of a DN or RDN	32	\<space>
Octothorpe (#) at the beginning of a DN or RDN	35	\#
Comma (,)	44	\,
Plus sign (+)	43	\+
Double-quote (")	34	\"
Backslash (\)	92	\\
Less-than-symbol (<)	60	\<
Greater-than-symbol (<)	62	\>
Semicolon (;)	59	\;

Abb. 13: Unerlaubte Zeichen in distinguished names (aus [23])

Kann auf die Verwendung dieser Sonderzeichen nicht verzichtet werden, so müssten diese entsprechend der Tabelle (Abb. 14) vor der Aufnahme in einen DN durch Escape-Sequenzen ersetzt werden. Diese Umsetzung ist bisher allerdings nicht implementiert.

char ***translateFilterString**(const char *filterString)

Da auch in Suchfiltern einige Sonderzeichen nicht erlaubt sind, gleichzeitig z.B. aber die Verwendung von Klammern für die Definition von WKT-Strings notwendig ist, führt diese Funktion eine Umsetzung der verbotenen Zeichen in entsprechende Escape-Sequenzen durch. Diese Umsetzung erfolgt nach folgender Tabelle (Abb. 14).

Character	Value (Decimal)	Value (Hex)	Escape Sequence
*	42	0x2A	\2A
(	40	0x28	\28
)	41	0x29	\29
\	92	0x5C	\5C
NUL	0	0x00	\00

Abb. 14: Unerlaubte Zeichen in Suchfiltern (aus [23])

Als Ergebnis liefert die Funktion einen neuen String, der nach Verwendung wieder freigegeben werden muss. Ist bei der Umwandlung ein Fehler aufgetreten, liefert die Funktion den Wert **NULL**.

char ***optimizeAreaString**(const char *areaString)

Diese Funktion führt die Umwandlung der nach außen sichtbaren 3D-Darstellung der *NexusAreas* in eine für die Verarbeitung im LDAP-Server optimierte interne 2D-Darstellung durch (vgl. Kap. 4.2.1). Alle Flächen, die den Funktionen **insertArea** und **updateArea** übergeben werden, müssen als Polygon entsprechend der für 3-dimensionale Objekte erweiterten WKT-Spezifikation angegeben werden [33]. Die x, y und z-Werte sind dabei 3-dimensionale Koordinaten (Längengrad, Breitengrad und Höhe über Geoid) gemäß dem World Geodetic System 1984 (WGS 84)[22]. Dies gilt auch für die Augmented Areas, die als Ergebnis der Suchfunktion **query** geliefert werden.

Zunächst wird in der Funktion **optimizeAreaString** die BoundingBox des Polygons bestimmt, indem die minimalen bzw. maximalen Koordinaten ermittelt werden. Außerdem wird die 3. Koordinate ausgefiltert. Eine Überprüfung auf Korrektheit des WKT-Strings erfolgt hier nicht. Erst durch die Funktion **nexusAreaSyntaxValidate** wird beim Einfügen eines neuen Eintrags in das LDAP-Verzeichnis die Syntax überprüft (vgl. Kap.4.2.1).

So wird z.B. die Darstellung:

```
POLYGON Z ( ( 10.5 20.2 600, 5 -23.2 580, -1.28 2 585, 10.5
20.2 600 ) )
```

in folgende vereinfachte Darstellung überführt.

```
-1.28 -23.2 10.5 20.2 # POLYGON ( ( 10.5 20.2, 5 -23.2,
-1.28 2, 10.5 20.2 ) )
```

Da diese Umwandlung nicht bijektiv ist, kann bei Anfragen an das räumliches Verzeichnis die ursprüngliche 3D-Darstellung aus der optimierten Form nicht zurückgewonnen werden. Statt als 3D-Darstellung eine ähnliche Form zu erzeugen (z.B. mit 0 als z-Wert), wird die nicht optimierte Darstellung im Attribut *Nexus3DArea* mit abgespeichert und bei Anfragen als Ergebnis zurückgeliefert.

Für das Ergebnis wird eine neuer String erzeugt, der nach Verwendung freizugeben ist. Ist bei der Umwandlung ein Fehler aufgetreten, liefert die Funktion **NULL** zurück.

4.4.4 Fehlercodes

Die Funktionen der Bibliothek *nexusLDAPClient* setzen bei Auftreten von Fehlern auch die Fehlervariable **errno**[23] auf einen der in der Headerdatei **nexusLDAPClient.h** definierten Werte:

0 **NEXUSLDAPCLIENT_SUCCESS**

22) http://www.wgs84.com

23) Die Fehlervariable **errno** ist eine globale Variable und daher nicht thread-sicher!

Keine Fehler bei der Ausführung aufgetreten.

Die folgenden Fehlercodes (1-6) werden zurückgeliefert, wenn bei der tatsächlichen Ausführung einer LDAP-Funktion ein Fehler auftritt.

1 **NEXUSLDAPCLIENT_LDAP_INIT_FAILED**

Die Funktion ldap_init ist fehlgeschlagen. Hierbei wird allerdings noch keine Verbindung zum LDAP-Server aufgenommen. Der Fehler kann also nur beim Anlegen der LDAP-Struktur aufgetreten sein.

2 **NEXUSLDAPCLIENT_LDAP_BIND_FAILED**

Die Funktion ldap_bind ist fehlgeschlagen. Dies ist zum Beispiel der Fall, wenn der LDAP-Server nicht erreichbar ist oder noch nicht gestartet wurde. Dieser Fehler tritt auch auf, wenn das übergebene Passwort nicht zum übergebenen Benutzernamen passt.

3 **NEXUSLDAPCLIENT_LDAP_SEARCH_FAILED**

Die Funktion ldap_search ist fehlgeschlagen. Dies ist normalerweise nur der Fall, wenn der Suchfilter nicht der LDAPv3-Syntax entspricht.

4 **NEXUSLDAPCLIENT_LDAP_LDAP_ADD_FAILED**

5 **NEXUSLDAPCLIENT_LDAP_MODIFY_FAILED**

Die Funktion ldap_add bzw. ldap_modify ist fehlgeschlagen. Dies kann z.B. an unzureichenden Rechten liegen (Kombination aus Benutzernamen und Passwort nicht akzeptiert). Dieser Fehler tritt auch auf, wenn bei der Überprüfung des *NexusArea*-Attributs auf Korrektheit der Syntax ein Fehler gefunden wird.

6 **NEXUSLDAPCLIENT_LDAP_DELETE_FAILED**

Die Funktion ldap_delete ist fehlgeschlagen. Dies liegt meist an unzureichenden Zugriffsrechten. Auch scheitert diese Operation, wenn noch untergeordnete Knoten existieren, dies sollte allerdings zuvor erfolgreich geprüft worden sein und zum Fehlercode 11 geführt haben.

Diese Fehlercodes (7-13) treten auf, wenn bereits bei der Überprüfung der Parameter ein Fehler gefunden wurde.

7 **NEXUSLDAPCLIENT_PARAMETER_UNDEFINED**

Einer oder mehrere Funktionsparameter sind nicht angegeben.

8 **NEXUSLDAPCLIENT_ILLEGAL_CHARACTER**

Die Funktion **testDNComponent** hat bei der Überprüfung der Parameter in den Bezeichnern für den Nexus-Typ oder im *NexusAreaLocator* ein unerlaubtes Zeichen gefunden.

9 **NEXUSLDAPCLIENT_OUT_OF_MEMORY**

Beim Reservieren von Speicher ist ein Fehler aufgetreten.

10 **NEXUSLDAPCLIENT_SUPERNODE_MISSING**

Ein neuer Knoten soll in das Verzeichnis eingefügt werden, aber der übergeordnete Typ-Knoten existiert nicht.

11 **NEXUSLDAPCLIENT_NODE_HAS_SUBNODES**

Ein Knoten soll aus Typ-Hierarchie gelöscht werden besitzt, aber untergeordnete Knoten. Dies können sowohl *NexusTypeNodes* also auch *NexusAreaNodes* sein.

12 **NEXUSLDAPCLIENT_NODE_EXISTS**

Ein neuer Knoten soll in das Verzeichnis eingefügt werden, es existiert allerdings bereits ein gleicher Eintrag.

13 **NEXUSLDAPCLIENT_NODE_DOES_NOT_EXIST**

Der Knoten, der gelöscht werden soll existiert nicht.

5. Erkenntnisse

Bei der gewählten Implementierung des räumlichen Verzeichnisses auf Basis eines LDAP-Dienstes gibt es einige grundsätzliche Schwierigkeiten. Durch den mehrschichtigen Systemaufbau (vgl. Abb. 7) ist für jede Anfrage an das System eine mehrfache Umsetzung und Kommunikation der Informationen zwischen den beteiligten Komponenten erforderlich. Neben dieser prinzipiellen Komplexität ergeben sich noch Probleme durch den räumlichen Charakter der zu verarbeitenden Daten.

Zusätzlich zu einigen Messergebnissen sollen in diesem Kapitel insbesondere auch die Erkenntnisse dokumentiert werden, die bei der Umsetzung des Systems gewonnen wurden, aber nicht in die Realisierung eingeflossen sind. Dazu gehören einerseits die erkannten Schwächen des Systems wie aber auch Ideen und Vorschläge, wie diese Probleme gelöst bzw. umgangen werden könnten.

5.1 Messergebnisse

Wegen eines Fehlers in der GEOS-Library[24] konnte das vorgestellte System bis zum Ende der Studienarbeit nicht in vollem Umfang auf der gewünschten Zielplattform (assun98: Sun Blade 1000, 2xUltraSPARC III@750 MHz, 4 GB RAM, SunOS 5.8, Solaris 8) zum Laufen gebracht werden. Diese Einschränkung betrifft den exakten Polygonvergleich, der eigentlich von der GEOS-Library durchgeführt werden soll (Funktion *nexusAreaOverlaps*, vgl. Kap. 4.2.2). Sollte sich dieser Fehler nicht lösen lassen, müsste entweder eine andere Bibliothek mit ähnlichem Funktionsumfang (d.h. Verarbeitung von WKT-Strings und Polygon-Polygon-Vergleich) eingesetzt werden oder die notwendige Funktionalität durch eigenen Code ersetzt werden.

Daher wurden die Messungen nicht nur auf der Zielplattform (mit eingeschränkter Funktion, d.h. Polygonvergleich nur durch die BoundingBoxes, also ohne Test der exakten Geometrien), sondern zum Vergleich auch auf einem anderen System durchgeführt (SuSE Linux 8.1, AMD Athlon 1 GHz, 256 MB RAM).

Die Tests wurden mit einem einfachen Testprogramm (nexusTest) durchgeführt. Dieses erzeugt zunächst die gewünschte Zahl an Typknoten. Dabei erhält jeder neue Typknoten zufällig einen der schon angelegten Typknoten als Supertyp. Dann wird entsprechend die gewünschte Anzahl von Augmented Areas eingefügt. Jede Augmented Area wird zufällig einem der Typknoten zugeordnet und erhält als Fläche ein regelmäßiges Vieleck mit

24) Auf dem Solaris-System führen alle nicht-trivialen Operationen der GEOS-Library zu einem *segmentation fault*. Dies liegt vermutlich an Unsauberkeiten bei der internen Speichernutzung der Library, die von der Solaris-Umgebung erkannt werden, während die gleichen Programme auf einem SuSE-Linux 8.1-System problemlos laufen. Diese Fehler treten nicht nur bei den für dieses Projekt verwendeten Programmen auf, sondern lassen sich auch durch die mit der Library gelieferten Test-Routinen reproduzieren.

wählbarer Eckenzahl. Zur Generierung von Anfragen wird dann eine beliebige Augmented Area herausgegriffen und ihre Attribute für eine Anfrage verwendet: es wird nach allen Einträgen gesucht, die dem gleichen Typ (oder einem der Untertypen) angehören, sich mit der Augmented Area überlappen und den gleichen NexusLevelOfDetail besitzen.

Auch wenn auf diese Weise keine detaillierte Steuerung des Verzeichnisaufbaus möglich ist, lassen sich doch Aussagen über das grundsätzliche Systemverhalten machen und für Tests eine beliebig große Last erzeugen.

5.1.1 Resultate für Zielplattform

Alle Messungen wurden 3 mal durchgeführt (mit unterschiedlich initialisiertem Zufallszahlengenerator), dann gemittelt und auf ganze Sekunden gerundet. Auffällige Ausreißer in einer 3er-Gruppe wurden dabei ignoriert, da sie vermutlich auf andere Systemeinflüsse zurückzuführen sind. Kleine Werte (insb. 0 oder 1 s) sind wegen der Messungenauigkeit nur als Näherungswerte zu sehen.

	10 Augmented Areas			100 Augmented Areas			1000 Augmented Areas		
	T	A	Q	T	A	Q	T	A	Q
10 Nexus Types	0	0	40	0	1	57	0	19	229
100 Nexus Types	1	0	42	1	1	45	2	32	93
1000 Nexus Types	14	0	43	13	1	56	14	21	90

T = Typknoten anlegen
A = Augmented Areas einfügen
Q = 10000 Abfragen (query) durchführen

Alle Werte in Sekunden

Abb. 15: Messergebnis für Solaris-System

Erkennbar ist in der Tabelle (vgl. Abb. 15), dass der Zeitbedarf für für das Einfügen neuer Knoten in den Verzeichnisbaum mit wachsender Verzeichnisgröße überproportional zunimmt. Dies ist wohl darauf zurückzuführen, dass zunächst der richtige Einfügepunkt gefunden werden muss, was in einem größeren Verzeichnisbaum mehr Zeit erfordert.

Interessant sind außerdem die kleineren Werte für die Laufzeit bei der Suche mit 1000 Augmented Areas und mehr als 10 Typknoten. Hier zeigt sich der Aufbau des Verzeichnisbaums nach der Typhierarchie. Unterhalb des zufällig ermittelten Einstiegspunktes für die Suche innerhalb des Baumes sind durch die detailliertere Typhierarchie weniger Einträge, die geprüft werden müssen. Die Suche kann also auf einen kleineren Teil des Verzeichnisses eingeschränkt werden.

Dies ist denn auch ein Schwachpunkt des Systems: Anfragen, bei denen nur ein sehr grober Typ gegeben ist, führen dazu, dass ein sehr großer Teil des gesamten Verzeichnisses durchsucht und einzeln überprüft werden muss.

5.1.2 Resultate für alternative Plattform

Für die alternative Plattform (SuSE Linux 8.1, AMD Athlon 1 GHz, 256 MB RAM) wurden genau die gleichen Tests durchgeführt, allerdings mit und ohne Verwendung der GEOS-Library für einen exakten Polygonvergleich.

	10 Augmented Areas			100 Augmented Areas			1000 Augmented Areas		
	T	A	Q	T	A	Q	T	A	Q
10 Nexus Types	1	1	26/48	1	5	35/66	1	69	172/190
100 Nexus Types	4	1	28/50	5	4	30/60	4	78	51/80
1000 Nexus Types	73	1	37/55	72	10	33/70	75	113	44/60

T = Typknoten anlegen
A = Augmented Areas einfügen
Q = 10000 Abfragen (query) durchführen, mit / ohne GEOS-Library

Alle Werte in Sekunden

Abb. 16: Messergebnis für Linux-System

Insgesamt ergibt sich dabei das gleiche Systemverhalten wie auf der Solaris-Plattform, wenn auch die Zeiten für das Anlegen der Typknoten und Augemented Areas größer, für die Abfragen allerdings kleiner sind.

Aber auch bei dieser Messung (vgl. Abb. 16) zeigt sich der überproportionale Zeitbedarf für die Generierung größerer Verzeichnisbäume, insbesondere beim Übergang von 100 zu 1000 Augmented Areas bzw. Typknoten, und die bessere Leistung bei Anfragen mit höherer Typselektivität.

Wie zu erwarten, sind die Ergebnisse mit und ohne Verwendung der GEOS-Library für das Anlegen des Verzeichnisses im Rahmen der Messungenauigkeit gleich, da hier die Library nur zur Überprüfung des WKT-Strings beim Einfügen einer Augmented Area verwendet wird.

Anders ist dies bei der Durchführung der Abfragen. Hier muss, für alle Einträge, die nicht bereits durch den Vergleich der BoundingBoxes ausgeschlossen werden können, ein genauer und aufwändiger Vergleich der Polygone ausgeführt werden. Wie sich allerdings gerade in den Werten für 1000 Augmented Areas und nur 10 Nexus Typen zeigt, ist diese exakte Prüfung nur die Ausnahme, daher weichen die beiden Messwerte nicht so deutlich voneinander ab. Ohne eine derartige schnelle Aussortierung

offensichtlich unpassender Einträge würde sich also die Laufzeit erheblich verschlechtern.

5.2 Einschränkungen der Systemleistung

Bei der Beurteilung der Leistungsfähigkeit des Systems beim Zugriff auf räumliche Daten zeigen sich 3 große Hemmnisse: die Auswahl von Einträgen erfolgt nicht in der Datenbank, die vom System unterstützten Indizes erlauben keine mehrdimensionalen (räumlichen) Strukturen und die Suche kann nicht auf Grund des Anfragegebiets auf einen Teil des Verzeichnisbaums beschränkt werden.

5.2.1 Auswahl durch Filterfunktionen

Die Auswahl geeigneter Einträge bei einer Anfrage an das Area Service Register geschieht nicht durch die Datenbank, vielmehr muss die Prüfung potentieller Kandidaten durch den OpenLDAP-Server erfolgen. Dies gilt nicht nur für räumliche Prädikate, die die Datenbank ohne räumliche Erweiterungen nicht versteht, sondern für alle Attribute[25]. Falls nicht durch einen Index auf passende Einträge zugegriffen werden kann, erfolgt der Vergleich zwischen Attributwert eines Eintrags und dem Anfragewert immer durch eine Filterfunktion im OpenLDAP-Server. Dieser Mechanismus erlaubt einerseits die große Flexibilität und Erweiterbarkeit des LDAP-Systems, führt natürlich aber auch zu einem großen Overhead.

Dazu liefert die Datenbank z.B. die Werte des *NexusArea*-Attributs von Einträgen, die auf Grund ihrer Position im Verzeichnisbaum in Frage kommen[26], an den OpenLDAP-Server. Dieser testet dann mit Hilfe der Filterfunktion *nexusAreaOverlaps* die Übereinstimmung von Eintrags- und Anfrageattributwert. Durch dieses Verfahren ist eine große Zahl von Nachrichten erforderlich, die zwischen Datenbank und OpenLDAP-Server ausgetauscht und verarbeitet werden müssen.

Der Test auf geometrische Überlappung ist außerdem – falls die Überlappung nicht bereits durch den Vergleich der BoundingBoxes ausgeschlossen werden kann – sehr aufwändig, da bei jedem Vergleich die WKT-String-Darstellung der Flächen erneut geparst werden muss.

5.2.2 Nutzbare Indexstrukturen

Das OpenLDAP-System sieht auch vor – ähnlich wie durch die Einführung neuer MatchingRules zum Vergleich von Attributwerten – Indexfunktionen für eine neue benutzerdefinierte Syntax zu definieren. Damit können für jeden Eintrag, z.B. durch eine

25) Dies gilt z.B. auch für die Integer-Attribute, die für die Speicherung des *LevelOfDetail* verwendet werden. Hier wird kein Suchprädikat an die Datenbank übergeben, der Vergleich von Attributwert mit dem Anfragefilter erfolgt vielmehr auch in diesem Fall durch die für die Integersyntax zuständige Filterfunktion.

26) Es kommen alle Augmented Areas in Frage, die unterhalb des bei der Anfrage angegeben Nexus-Typs im Verzeichnisbaum eingetragen sind.

geeignete Hash-Funktion, ein oder mehrere Schlüsselwerte[27] erzeugt werden. Bei einer Anfrage an das Verzeichnis können dann gezielt Einträge mit bestimmten Schlüsselwerten aus der Datenbank abgerufen werden.

Komplexe mehrdimensionale Indexstrukturen, wie sie z.B. für R-Trees notwendig sind, lassen sich mit solchen einfachen Schlüsselwerten nicht umsetzen. Da allerdings für einen Eintrag auch mehrere Indexwerte generiert werden können, sind Ansätze möglich, die geometrische Informationen durch raumfüllende Kurven (z.B. Z-Kurven, Hilbert-Kurven usw.) auf einfache Indexwerte abbilden. Die Speicherung mehrerer Indexwerte für eine Augmented Area ist notwendig, da Flächen bei diesen Verfahren durch mehrere Indexwerte repräsentiert werden müssen: jeder Indexwert repräsentiert ein kleines Gebiet, dessen Lage durch die verwendete Kurve festgelegt ist; zur Beschreibung einer Fläche müssen dann die Indexwerte aller Gebiete, die sich mit der Fläche überlappen, gespeichert werden. Für jeden Verzeichniseintrag muss also eine beliebige Anzahl von Indexwerten gespeichert werden (vgl. Abb. 17).

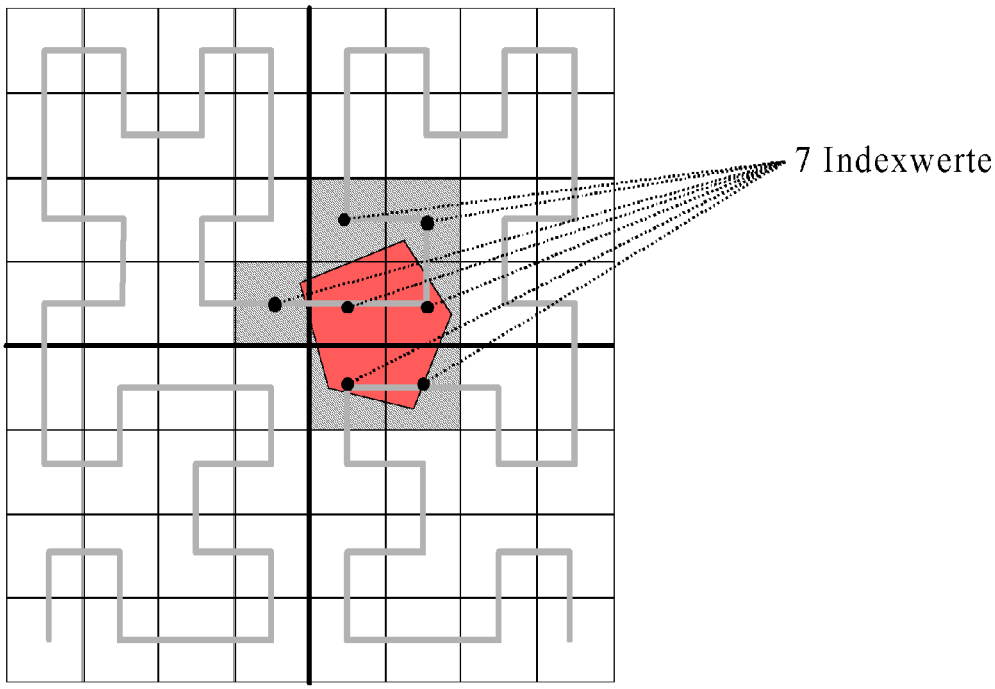

Abb. 17: Indexierung von Flächen durch Hilbert-Kurve

Für das Anfragegebiet wird nach der gleichen Methode eine Menge von Indexwerten bestimmt. Bei der Suche müssen nun Einträge gefunden werden, die Indexwerte besitzen, die gleich einem der Indexwerte des Anfragegebiets sind. Kritisch für die Performanz dieses Verfahrens ist dabei die Größe der Indexgebiete. Bei einer zu groben

27) Jeder Schlüssel hat normalerweise eine Länge von 32 Bit (gespeichert als 4 Characters bzw. Bytewerte mit je 8 Bit). Mehrere Schlüsselwerte werden z.B. dazu verwendet, Indexwerte für alle Substrings eines Attributwerts anzulegen und dadurch auch gezielt nach Strings zu suchen, die einen Teilstring enthalten.

Annäherung an die Augmented Areas werden womöglich viele eigentlich unnötige Einträge geliefert, bei einer zu feinen Unterteilung des Suchraums steigt der Verwaltungs- und Vergleichsaufwand für jeden Eintrag stark an.

Versucht man den Verwaltungsaufwand zu reduzieren und speichert für jeden Eintrag nur einen Indexwert ab (für das kleinste Teilgebiet, das die Fläche vollständig überdeckt), so ist nur eine sehr ungenaue Annäherung an den Eintrag möglich, es müssen also später viele Einträge unnötig geprüft werden. Besonders kritisch ist es hierbei, wenn die Fläche zentrale Grenzlinien überlappt: im oben gezeigten Beispiel überlappt kein Teilgebiet des Suchraums die Fläche vollständig, es muss daher der gesamte Suchraum als Indexgebiet gespeichert werden (vgl. Abb. 18). Ein weiteres Problem ist, dass nun eine Überlappung von Indexgebieten nicht nur bei Schlüsselgleichheit auftritt, die Indexwerte für Flächen unterschiedlicher Größe können nicht direkt verglichen werden. Dies verhindert insbesondere auch den Abruf passender Einträge über den Indexschlüssel aus der Datenbank.

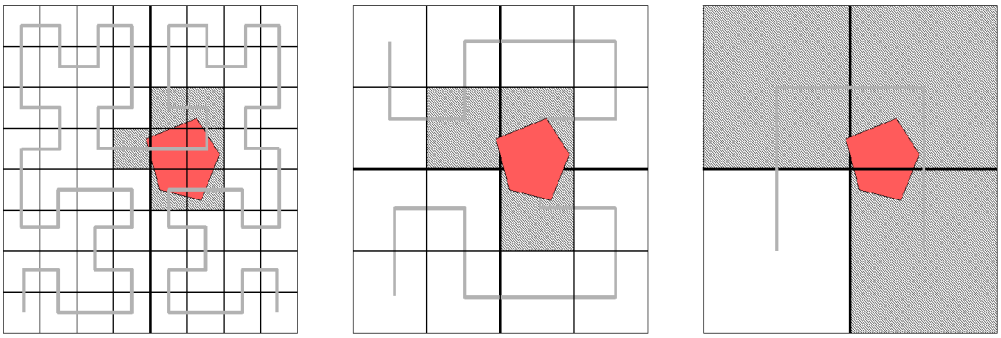

Abb. 18: Probleme bei der Repräsentation von Flächen durch einen Indexwert

Ein anderer Ansatz wäre es, auf eine flexible und dafür möglichst exakte Indexierung der Einträge zu verzichten und den gesamten Suchraum in relativ grobe Planquadrate gleicher Größe einzuteilen, in die normalerweise jeder Eintrag hineinpasst[28]. Falls typischerweise nur relativ kleine Augmented Areas (z.B. Gebäude) im Area Service Register gespeichert werden, könnte evtl. auch schon durch diese sehr grobe, dafür aber einfache Indexunterstützung die Leistung des Systems deutlich gesteigert werden. Für jeden Verzeichniseintrag verwendet nun man die Nummer des passenden Planquadrats als Indexwert. Über den oder die entsprechend aus dem Anfragegebiet erzeugten Indexwerte können dann in Frage kommene Einträge in der Datenbank gesucht werden. Auch bei einer groben Untergliederung können zumindest viele Einträge ausgeschlossen werden, die sonst erst an den OpenLDAP-Server übertragen und dort durch die Filterfunktion getestet werden müssten. Wie andere hier vorgeschlagene einfache Verfahren reagiert aber auch dieses Verfahren empfindlich auf eine stark unterschiedliche Größe der Augemented Areas.

28) Größere Flächen oder Flächen, die auf Grenzen liegen, erfordern eine Sonderbehandlung, z.B. durch mehrfachen Eintrag oder Auftrennung entlang der Grenzen.

Die verwendete Backend-Datenbank, die Berkeley-Database, bietet selbst keine geometrischen Attribute und keine mehrdimensionalen Indexverfahren. Jedoch auch die Verwendung einer Datenbank mit räumlichen Erweiterungen, z.B. IBM's DB2 mit Spatial Extender (vgl. [34]), würde dieses Problem nicht lösen, da die gesamte Datenbank-Anbindung des OpenLDAP-Servers auf die Verarbeitung von Zeichenketten als Attributwerte ausgerichtet ist. So hat auch bei der in diesem Projekt durchgeführten Erweiterung weder der OpenLDAP-Server noch die Datenbank Informationen über den besonderen räumlichen Charakter der *NexusArea*-Syntax. Vielmehr werden die geometrischen Daten als Zeichenketten gespeichert und nur bei Bedarf in eine für geometrische Operationen geeignete Darstellung umgewandelt.[29]

Die direkte Anbindung an eine räumliche Datenbank zur Beschleunigung der Suchanfragen würde also erhebliche Änderung bei der Verarbeitung der Attribute im OpenLDAP-Server und der Anbindung an die Backend-Datenbank erfordern. Der OpenLDAP-Server müsste um neue räumliche Datentypen erweitert werden, die - anders als bei der vorgestellten einfache Syntaxerweiterung – auch in der Datenbank als räumliche Daten verarbeitet werden. Dies betrifft insbesondere die Datenbank-Anbindung, die bisher noch keine räumliche Datenbank unterstützt. Gelingt es allerdings, auf diese Weise den größten Teil der Anfrageverarbeitung auf die Datenbank zu verlagern, stellt sich die Frage nach der Notwendigkeit des LDAP-Systems als zusätzliche Systemschicht. Es wäre auf jeden Fall genau zu prüfen, ob die Alternative, eine Datenbank als Basis für das Area Service Register zu verwenden (vgl. [31]) nicht eine bessere Systemleistung ermöglicht.

Alle Änderungen wären außerdem bei jeder Weiterentwicklung des OpenLDAP-Systems zu testen und evtl. anzupassen, wohingegen die vorgestellte Erweiterung ohne Eingriff in den Quelltext möglich und daher auch auf andere LDAP-Systeme übertragbar ist (vgl. Kap. 5.4).

5.2.3 Gliederung des Verzeichnisbaums

Das vielleicht größte Problem ergibt sich allerdings aus dem gewählten Aufbau des LDAP-Verzeichnisbaums: dieser beruht auf der Hierarchie der Nexus-Typen und ordnet alle Augmented Areas entsprechend der Dienst-Typen, die sie bieten, in das Verzeichnis ein (vgl. Kap. 3.1).

Auf diese Weise lassen sich Anfragen, die sich auf einen speziellen Nexus-Typ beziehen, auch bei nur ungenauer Ortsangabe[30] einfach auf einen sehr kleinen Teil des gesamten Verzeichnisses eingrenzen und somit rasch beantworten. Bei Anfragen mit hoher räumlicher Selektivität (d.h. kleinem Anfragegebiet) und geringer Typselektivität[31] (d.h.

29) Dazu übergibt die Filterfunktion *nexusAreaOverlaps* die Stringdarstellung an die GEOS-Library. Diese baut aus dem WKT-String ein Polygon-Objekt auf, für das dann die Überlappung getestet wird. Die GEOS-interne Darstellung wird nicht gespeichert, sondern jedesmal bei Bedarf neu generiert.

30) z.B.: „Wann hat die Messe geöffnet?“

31) z.B.: „Was ist heute Abend in der Nähe los?“ (Also die allgemeine Suche nach Veranstaltungen in der engeren Umgebung.)

sehr allgemeiner oder gar ohne NexusTyp) ist dies dagegen nicht möglich. In diesem Fall muss der Verzeichnisbaum fast vollständig durchsucht und viele Einträge einzeln und aufwändig auf Übereinstimmung mit dem Anfragegebiet (vgl. Kap. 5.2.1) getestet werden.

Die Gliederung des Verzeichnisbaums nach der Typhierarchie verhindert auch die Aufteilung des Area Service Registers (ASR) auf mehrere Rechner nach räumlichen Kriterien, da der LDAP-Standard und das OpenLDAP-System nur die Verteilung eines Gesamtverzeichnisses auf Teilbäume vorsieht. Jedes Teilverzeichnis ist dann für alle Einträge mit einem bestimmten Suffix im distinguished name (DN) zuständig. So könnte z.B. beim Beispiel-Verzeichnis (vgl. Abb. 5/6) Server A für die Einträge zuständig sein, deren *DN* mit dem Suffix "**`nt=entertainment, o=nexus`**" endet. Server B übernimmt dann alle Einträge für "**`nt=transport, o=nexus`**", usw. Die Verteilung kann also nur nach den Nexus-Typen, nicht aber regional erfolgen.

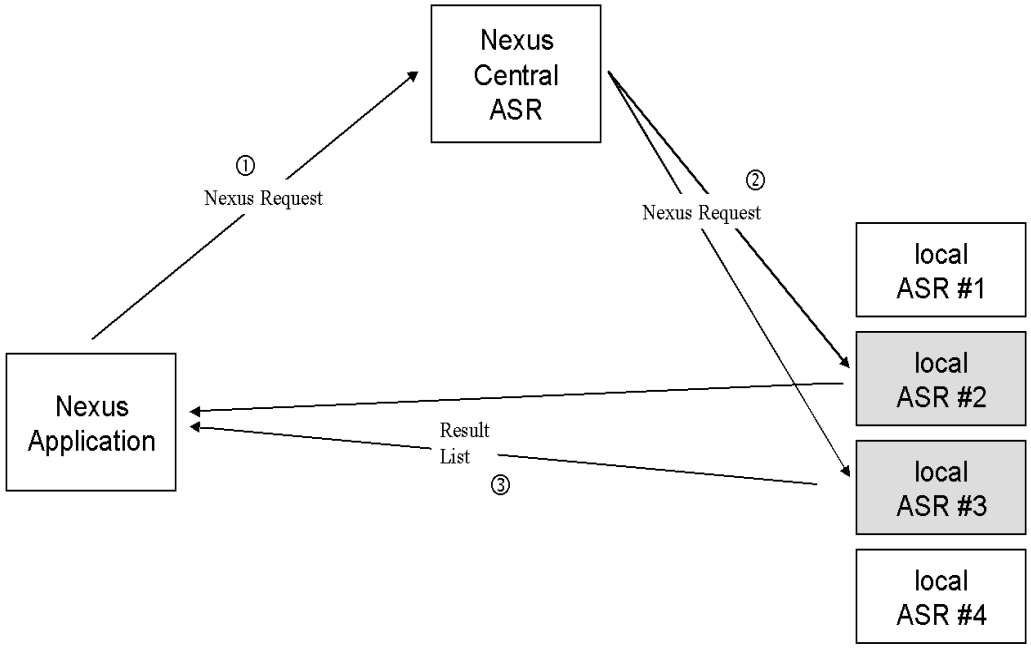

Abb. 19: Variante mit regionalen ASRs und Zentralregister

Will man dennoch das Gesamtverzeichnis in voneinander unabhängige[32] regionale Area Service Register auftrennen, ist ein zusätzliches Gesamtregister notwendig. Dieses bestimmt dann für jede Anfrage die[33] regionalen Area Service Register, die für das Anfragebiet zuständig sind und leitet die Anfrage an diese weiter (vgl. Abb. 19). Jedes dieser zuständigen Area Service Register liefert dann eine Liste aller zutreffenden Einträge an die Nexus-Anwendung zurück.

32) Jedes regionale Area Service Register enthält nur Einträge für das jeweilige Zuständigkeitsgebiet, das Zentrale Register enthält keine Augmented Areas, sondern nur ein Verzeichnis der regionalen ASRs und ihrer Zuständigkeitsgebiete.

33) Wegen evtl. Überlappung ist nicht immer eine eindeutige Zuordnung möglich.

Ein Problem ist dabei die Behandlung von Augmented Areas, die auf den Grenzen der Zuständigkeitsgebiete liegen, also nicht eindeutig einem regionalen Area Service Register zugeordnet werden können. Diese Augmented Areas müssen dann entweder entlang der Grenzen in mehrere Teilflächen aufgeteilt werden, die vom jeweils zuständigen Area Service Register gespeichert werden, oder die komplette Augmented Area wird von allen in Frage kommenden Area Service Registern gespeichert. Durch die redundante Speicherung der Augmented Area (komplett oder in Teilen) können solche Einträge auch mehrfach in der Ergebnisliste auftreten. Dies muss bei der Zusammenführung der Teilergebnisse berücksichtigt werden.

Will man die mehrfache Speicherung von Einträgen vermeiden, kann dies auch durch eine Anpassung der Zuständigkeitsgebiete erfolgen, die sonst fest und überlappungsfrei sind. Lässt sich nun ein neuer Eintrag, da er auf der Grenze mehrerer Zuständigkeitsgebiete liegt, nicht eindeutig einem der regionalen Area Service Register zuordnen, wird nun eines der in Frage kommenden Area Service Register zur Speicherung ausgewählt[34]. Das Zuständigkeitsgebiet dieses Area Service Register wird dazu so erweitert, dass es danach auch den neuen Eintrag vollständig abdeckt. Bei der Erweiterung des Gebiets besteht natürlich ein Konflikt zwischen der möglichst engen Ausdehnung des Gebiets und einer möglichst einfachen Geometrie, die die schnelle Durchführung von Tests erlaubt. Zu beachten ist, dass nun ein regionales Area Service Register nicht mehr unbedingt alle Augmented Areas verwaltet, die innerhalb seines Zuständigkeitsbereichs liegen. Bei der Suche nach den für eine Anfrage zuständigen Area Service Registern müssen daher auf jeden Fall alle Zentralregistereinträge überprüft werden und die Anfrage an alle zuständigen regionalen Area Service Register übermittelt werden, während dies sonst nur notwendig ist, wenn keines der Area Service Register das Anfragegebiet vollständig überdeckt.

Insgesamt ist aber bei der Einführung eines Zentralregisters zu untersuchen, ob bei tatsächlichen Daten und realistischem Nutzungsverhalten der erzielte Leistungsgewinn durch die regionale Verteilung den zusätzlichen Kommunikationsaufwand rechtfertigt. Kann man aber unterstellen, dass Benutzer hauptsächlich auf kleinräumige, lokale Informationen zugreifen, die sich somit meist auf nur ein regionales Verzeichnis beschränken lassen, könnte durch Zwischenspeicherung des lokal zuständigen Area Service Registers der Zugriff auf das Zentralregister und der damit verbundene Systemengpass oft vermieden werden.

Voraussetzung für eine Leistungsverbesserung durch diese Variante ist es jedoch, dass die Aufteilung der Area Service Register gut zur Größe des typischen Anfragegebiets passt: jede Anfrage sollte von nur einem (oder wenigen) Area Service Registern beantwortet werden können. Je kleiner die lokalen Zuständigkeitsgebiete sind, desto weniger Einträge muss jedes Area Service Register überprüfen, werden die Zuständigkeitsgebiete (z.B. in Ballungsräumen) aber zu klein, so besteht die zunehmende Gefahr von Überlappungen, so dass – ähnlich wie bei Mehrprozessorsystemen – die Systemleistung trotz zusätzlicher Knoten nicht weiter zunimmt.

34) Mögliche Auswahlstrategien wären z.B.: größte Überlappung mit dem neuen Eintrag oder wenigste Einträge ...

Ist eine optimale Anpassung der regionalen Aufteilung nicht möglich (z.B. weil die Größe der Anfragegebiete stark schwankt), sind die zu erwartenden Ergebnisse dieser Variante vermutlich schlechter als die der vorgestellten Implementierung.

5.3 Alternative Gliederung des Verzeichnisbaums

Nach der im vorigen Kapitel beschriebenen Notwendigkeit eines räumlichen Indexes und den Problemen bei einer direkten Implementierung (vgl. Kap. 5.2.2) stellt sich die Frage, ob man die erhoffte Beschleunigung nicht auch durch eine andere Struktur des Verzeichnisbaums erreichen könnte: die Einträge werden nach ihrer räumlichen Anordnung geordnet und ein räumlicher Index durch den Aufbau des Verzeichnisbaums nachgeahmt[35]. Eine gezielte Suche nach Einträgen mit einem bestimmten Nexus-Typ kann über einen Index auf dem Typ-Attribut erfolgen, der bereits vom System unterstützt wird. Eine Gliederung des Verzeichnisses nach der Typhierarchie ist also nicht unbedingt notwendig, allerdings müssen dann auf andere Weise bei einer Anfrage alle zu einem gesuchten Nexus-Typ passenden Subtypen gefunden werden.

Die Speicherung der Typhierarchie könnte z.B. durch einen weiteren Verzeichnisbaum erfolgen, der nur die Typhierarchie verwaltet. Eine andere Alternative wären die bereits in Kap. 5.2.2 erwähnten mehrfachen Indexwerte, die durch eine benutzerdefinierte Funktion generiert werden können. Legt man zu jedem Verzeichniseintrag nicht nur einen Schlüsselwert für den Nexus-Typ der Augmented Area ab, sondern auch die Schlüsselwerte für alle direkten Supertypen in der Hierarchie[36], so können bei einer Anfrage an das Verzeichnis über den Schlüsselwert des Anfragetyps gleichzeitig auch alle untergeordneten Einträge abgerufen werden.

Bei einer räumlichen Gliederung des Verzeichnisses ist außerdem zu unterscheiden, ob die Logik zur Ausnutzung der räumlichen Gliederung im OpenLDAP-Server oder auf der Anwendungsseite (also im NexusLDAPClient) liegen soll, da sich aus dieser Festlegung Konsequenzen für die Art des Indexes ergeben.

Soll ohne eine Änderung des OpenLDAP-Servers ausgekommen werden, sind nur statische Indexstrukturen (z.B. Quad-Trees) möglich. Dynamische Indexstrukturen (z.B. R-Trees), bei denen nicht ohne Untersuchung der Indexknoten entschieden werden kann, an welcher Position des Verzeichnisbaums ein Eintrag zu suchen ist, erfordern dagegen eine Erweiterung des OpenLDAP-Servers.

5.3.1 Statische Indexstruktur

Baut man den LDAP-Verzeichnisbaum nach dem Vorbild eines Quad-Trees auf, so werden zunächst alle Indexknoten für einen Baum festgelegter Tiefe angelegt. Alle diese Knoten werden dabei nach einem festen Schema benannt, so dass später der

35) Ein Überblick über verschiedene mehrdimensionale Index-Verfahren findet sich z.B. in [35].

36) In dem einfachen Verzeichnisbaum (vgl. Abb. 6) würden also z.B. für den Eintrag mit dem "dn: nal=nexus://palast.de, nt=cinema .." nicht nur der Typindex für "cinema", sondern auch je ein Indexwert für die übergeordneten Nexus-Typen "entertainment" und "root" abgespeichert.

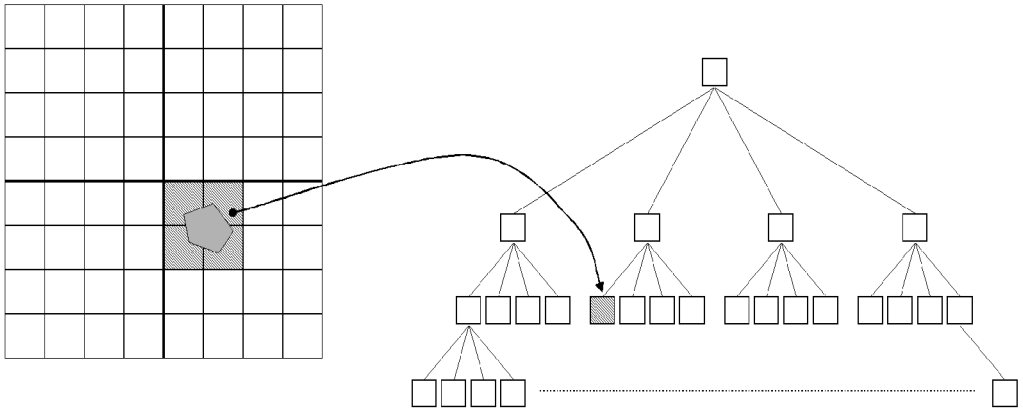

Abb. 20: Räumlicher Index mit statischer Struktur.

distinguished name (DN) jedes Indexknotens auch ohne Zugriffe auf das Verzeichnis bestimmt werden kann. Jedem Knoten ist dann ein Planquadrat des gesamten Suchraums zugeordnet, dessen Größe von der Indexebene des Knotens abhängig ist (vgl. Abb. 20).

Nun müssen alle Augmented Areas so in diesen Indexbaum einsortiert werden, dass eine gezielte Suche nach passenden Einträgen erfolgen kann. Dafür muss es möglich sein, aus dem Anfragegebiet einen (oder einige wenige) Einstiegsknoten für die Suche im Verzeichnisbaum zu ermitteln, für die die Anzahl unnötig getesteter Einträge so gering wie möglich ist.

Damit auch bei einem tiefen Sucheinstieg in den Verzeichnisbaum alle in Frage kommenden Einträge gefunden werden, müssen die Augmented Areas unterhalb der letzten Indexebene, den Blattknoten des Indexbaum eingetragen werden. Je nach Tiefe des Indexbaums (und damit verbundener Größe der kleinsten Indexgebiete) kann sich eine Augmented Area auch auf mehrere Indexgebiete erstrecken. Um eine evtl. häufige mehrfache Speicherung der Augmented Areas zu vermeiden, kann es sinnvoll sein, hier nicht die Augmented Areas selbst, sondern nur Verweise auf eine entsprechende Liste zu speichern. Auch ist bei der Verarbeitung der Ergebnisse das mehrfache Auftreten der selben Augmented Area zu bedenken.

Für die Ermittlung des Einstiegsknotens aus dem Anfragegebiet gibt es nun verschiedene Möglichkeiten. Soll nur eine einzige Anfrage an den OpenLDAP-Server gestellt werden, so wird aus der Lage und Größe des Anfragegebiets der kleinstmögliche Indexknoten ermitteln, dessen zugeordnetes Planquadrat diese Fläche vollständig überdeckt. Alle in Frage kommenden Einträge sind dann unterhalb dieses Indexknotens zu finden und müssen dann auf Typübereinstimmung, passenden *LevelOfDetail* und durch genauen Geometrievergleich getestet werden. Je nachdem, wie vollständig das Anfragegebiet das Planquadrat des Indexknotens ausfüllt, kann es hierbei jedoch zu einer großen Anzahl unnötig geprüfter Einträge kommen.

Der entgegengesetzte Ansatz wäre es, aus dem Anfragegebiet (wie beim Eintragen der Augmented Areas) alle diejenigen kleinsten Indexgebiete zu ermitteln, die sich mit dem Anfragegebiet überlappen. Für jeden der entsprechenden Indexknoten wird dann eine

Anfrage an den OpenLDAP-Server gestellt. Zwischen diesen beiden extremen Ansätzen sind auch Zwischenlösungen möglich: überdeckt das Anfragegebiet ein größeres Indexfeld einer höheren Ebene bereits vollständig, müssen alle untergeordneten Indexknoten natürlich nicht mehr ermittelt werden und der höhere Indexknoten kann für die Suchanfrage verwendet werden. Auf diese Weise kann die Anzahl der sonst nötigen Anfragen an den OpenLDAP-Server reduziert werden, ohne dass Leistungseinbußen durch unnötig getestete Verzeichniseinträge auftreten.

Wird beim Eintragen der Augmented Areas und bei der Ermittlung der Einstiegsknoten auch geprüft, ob die jeweilige Fläche sich nicht nur mit einem Indexgebiet überlappt, sondern dieses vollständig überdeckt, kann diese Information später für einen vereinfachten Vergleich verwendet werden, sodass auf den aufwändigen exakten Vergleich der Flächengeometrien in manchen Fällen verzichtet werden kann.

Problematisch erweist sich dieser statische Ansatz bei ungleichmäßiger Verteilung der Einträge, wodurch die Anzahl der Einträge pro Indexknoten sehr unterschiedlich werden kann. Dies könnte durch eine Flexibilisierung mit lokal unterschiedlicher Indextiefe korrigiert werden. Damit steigt allerdings auch der Verwaltungsaufwand, da die Existenz von Indexknoten einer bestimmten Stufe immer zuerst geprüft werden muss.

5.3.2 Dynamische Indexstruktur

Erheblich aufwändiger gestaltet sich die Umsetzung eines dynamischen Indexverfahrens (z.B. R-Trees), bei denen keine festen Indexgebiete existieren. Auf jeder Ebene des räumlichen Indexes muss vielmehr geprüft werden, in welchen der Nachfolgeknoten das gesuchte Gebiet hineinpasst. Will man auch diese Lösung ohne Eingriff in den OpenLDAP-Server umsetzen (also nur durch Erweiterung des Clients), so müssen auf jeder Stufe der Suche alle Nachfolgeknoten des Indexknotens mit einer neuen Anfrage an den OpenLDAP-Server ermittelt werden. Die Folge von Anfragen endet, sobald das Ende des räumlichen Indexes (also die detaillierteste Stufe) erreicht wird oder die weitere Suche nicht mehr auf einen Nachfolgeknoten beschränkt werden kann.

Durch die mehrstufige Abwicklung einer Suche entsteht ein derartig hoher Kommunikationsaufwand zwischen LDAP-Server und LDAP-Client, dass dynamische Indexverfahren nur bei einer internen Umsetzung, also durch Erweiterung des OpenLDAP-Servers sinnvoll wären. Damit dieser aber die räumliche Gliederung des Verzeichnisbaums durch ein derartiges Suchverfahren ausnutzen kann, müsste die gesamte Suchfunktionalität weitgehend neu implementiert werden.

Auch bei dieser nicht implementierten Variante, die Hierarchie des LDAP-Verzeichnisbaums zur räumlichen Gliederung des Area Service Registers zu nutzen stellen sich also viele Probleme. Der mögliche Leistungsgewinn erfordert in jedem Fall z.T. erheblichen zusätzlichen Aufwand und ist stark vom späteren Nutzungsverhalten abhängig.

5.4 Übertragbarkeit auf andere LDAP-Systeme

Die Entscheidung für OpenLDAP zur Umsetzung des räumlichen Verzeichnisses wurde in Kap. 2.4 auch damit begründet, Erweiterungen oder Änderungen des Systems zur Verbesserung des Zugriffsverhaltens (vgl. Kap. 5.2 bzw. 5.3) zu ermöglichen. Bei der genaueren Untersuchung des Quelltextes des OpenLDAP-Systems zeigt sich allerdings, das hierfür erheblicher Aufwand erforderlich ist. Vor jeder Veränderung des Systemverhaltens ist auf jeden Fall eine detaillierte Einarbeitung in die umfangreichen und kaum dokumentierten Quellen notwendig. Außerdem wäre jede Erweiterung dieser Art nur auf dem OpenLDAP-System einsetzbar und bei jeder Weiterentwicklung des OpenLDAP-Systems anzupassen.

Verzichtet man aber auf solche Erweiterungen, die einen direkten Einriff in den LDAP-Server erfordern, so ist eine Umsetzung des in dieser Studienarbeit vorgestellten Ansatzes auch auf andere kommerzielle LDAP-Systeme (z.B. Novell Directory Service NDS, ..) möglich. Die Auswahl des verwendeten LDAP-Systems kann dann frei nach anderen Kriterien wie Verfügbarkeit, Robustheit, Skalierbarkeit usw. erfolgen.

Die für diese Produkte bereitgestellten Erweiterungsschnittstellen (vgl. Kap. 2.3.1) reichen für die Einführung einer neuen Syntax mit zugehörigen Vergleichsregeln aus, die Verfügbarkeit des Quelltextes ist nicht erforderlich.

6. Resümee und Ausblick

In dieser Studienarbeit wird gezeigt, dass die Umsetzung eines räumlichen Verzeichnisses für Informationsdienste in Nexus auf der Basis von Standard-Verzeichnisdiensten möglich ist. Die vorgestellte Implementierung wurde auf der Basis des OpenLDAP-Systems durchgeführt.

Der Aufbau des LDAP-Verzeichnisbaums folgt dabei der Nexus-Typhierarchie. Alle Augmented Areas werden als Nachfolgeknoten ihres jeweiligen Nexus-Typs eingetragen, dadurch können leicht alle Einträge eines Nexus-Typs und all seiner Subtypen gefunden werden. Zur Verarbeitung der räumlichen Attribute der Augmented Areas wurde - durch Ausnutzung der vorhandenen Erweiterungsschnittstellen - eine neue Syntax eingeführt. Die geometrische Prüfung in Frage kommender Einträge erfolgt durch eine Filterfunktion innerhalb des LDAP-Servers, eine Indexunterstützung bei der Suche nach geeigneten Augmented Areas existiert dabei nicht.

Daher werden, insbesondere wenn der Typ des gewünschten Dienstes bei Anfragen an das System nur ungenau angegeben werden kann, die Vorteile eines Verzeichnisses mit seiner Baumstruktur kaum ausgenutzt. Auch die mögliche Leistung des Systems wird dadurch eingeschränkt, dass hierbei keine frühe Beschränkung auf eine kleine Anzahl geographisch in Frage kommender Objekte durch die Eingrenzung der Suche auf einen Teilbaum des gesamten Verzeichnisses möglich ist. Das vorgestellte System reagiert also empfindlich auf Anfragen mit geringer Typselektivität und hoher räumlicher Selektivität, da in solchen Fällen viele Einträge unnötig geprüft werden müssen.

Reicht die erreichbare Performanz zusammen mit den für LDAP-Verzeichnisdienste vorgesehenen Funktionen zur Replikation für den Anwendungszweck nicht aus, müssen andere Ansätze untersucht werden, um eine Verbesserung des Zugriffsverhaltens zu erzielen.

In Kap. 5 werden einige nicht implementierte Varianten vorgestellt. Am interessantesten – und am leichtesten Umzusetzen - erscheint dabei die Idee, für jeden Verzeichniseintrag zumindest einen groben Indexwert zu erzeugen (vgl. Kap. 5.2.2). Hierdurch wird auf jeden Fall der Kommunikationsaufwand zwischen LDAP-Server und der Backend-Datenbank reduziert, da weniger Einträge zur Überprüfung an den LDAP-Server übertragen werden müssen.

Auch der Ansatz, den Verzeichnisbaum nicht nach der Typhierarchie aufzubauen, sondern räumlich zu gliedern (vgl. Kap. 5.3.1), ist ohne Veränderung des LDAP-Systems durchführbar. Allerdings muss dann die Verwaltung der Typhierarchie auf andere Weise erfolgen, da sie nicht mehr durch den Aufbau des Verzeichnisbaums berücksichtigt ist.

Auch für die Lösung, das Gesamtverzeichnis durch Einführung eines zentralen Registers auf mehrere regional zuständige Rechner zu verteilen (vgl. Kap. 5.2.3), muss der LDAP-Server nicht erweitert werden. Hier ist die zu erwartende Leistung um so besser, je feiner die Aufteilung in Teilverzeichnisse ist, also je weniger Einträge jeder einzelne Knoten

verarbeiten muss. Ein kritische Größe wird allerdings erreicht, wenn sich die Zuständigkeitsgebiete zunehmend überlappen, eine Anfrage also normalerweise nicht mehr von einem (oder sehr wenigen) lokalen Area Service Registern beantwortet werden kann.

Diese 3 „einfachen“ Vorschläge bringen wohl nur bei einer in etwa gleichmäßige Größe der Verzeichniseinträge gute Ergebnisse, denn dann kann die erforderliche statische Aufteilung des Suchraums gut an die typische Größe der Augmented Areas angepasst werden und so ein guter Kompromiss zwischen Verwaltungsaufwand und Genauigkeit der Verfahren gefunden werden.

Alle anderen Varianten erfordern dagegen Änderungen am LDAP-Server und sind daher erheblich aufwändiger zu implementieren, werden wegen ihrer Flexibilität aber auch bei ungleichmäßiger Größe und Verteilung der Einträge nicht so rasch ineffizient wie die statischen Ansätze. Dies gilt sowohl für eine direkte Anbindung an ein räumliches Datenbanksystem (vgl. Kap. 5.2.2) wie auch für die Idee, einen räumlichen Index in den Aufbau des LDAP-Verzeichnisbaums zu integrieren (vgl. Kap. 5.3.2). Dafür ist aber für auch deutliche Verbesserung des Zugriffsverhaltens zu erwarten, wenn das Problem der fehlenden räumlichen Gliederung nicht nur umgangen wird, sondern gezielt nach räumlich passenden Einträgen gesucht werden kann.

Alle hier aufgeführten Ideen – und es sind sicherlich weitere Ansätze denkbar – erzeugen zunächst einmal – neben etwaigen Schwierigkeiten bei der Realisierung – zusätzlichen Verwaltungsaufwand bei der Verarbeitung der Daten. Es ist also genau zu untersuchen, ob dieser Aufwand bei realistischen Daten und Anfragen zu einer tatsächlichen Verbesserung der Systemleistung führt.

Insgesamt zeigt sich, dass die Schwierigkeiten nicht bei der Speicherung und Verwaltung der Verzeichniseinträge, sondern vielmehr beim effizienten Zugriff auf räumlich passende Einträgen liegen. Hierfür bietet das verwendete OpenLDAP-System wie auch die anderen Standardverzeichnisdienste keinerlei Unstützung. Dies muss durch eigene Erweiterungen – in Verbindung mit einem geschickten Aufbau des Verzeichnisbaums – erfolgen.

Anhang A: Verwendete Software

Alle Hinweise beziehen sich auf eine lokale Installation im Unterverzeichnis **`/home/neumancn/nexus`**, sodass hierzu keine Administratorrechte benötigt werden (vgl. Abb. 21). Dabei wurde soweit wie möglich auf absolute Pfadangaben verzichtet. In Fällen, in denen dies nicht möglich ist, wird darauf hingewiesen. Durch die gewählte Konfiguration bei der Installation der einzelnen Pakete kommen alle später benötigten Bibliothek- bzw. Header-Dateien in ein gemeinsames Verzeichnis.

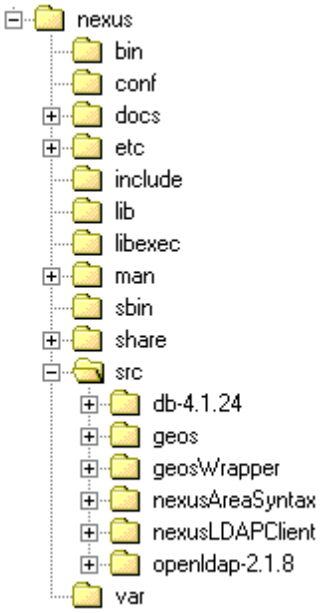

Abb. 21: Verzeichnisstruktur der Projektdateien

Alle Befehle sind für die Ausführung in einer *Bourne Again Shell (bash)* ausgelegt, ggf. ist diese zuvor zu starten. Zuerst sind nacheinander die verschiedenen Softwarepakete zu installieren. Dabei sollte die aufgeführte Reihenfolge eingehalten werden, da gegenseitige Abhängigkeiten existieren. Nach der erfolgreichen Installation aller Komponenten kann dann das Area Service Register mit den im Abschnitt A.2.4 beschriebenen Anweisungen in Betrieb genommen werden.

A.1 Berkeley Database

Als Backend für den OpenLDAP Server wird die Berkeley DB in der Version 4.1.24 von Sleepycat Software[37] verwendet. Dies ist die bevorzugte Datenbank für OpenLDAP, allerdings werden auch andere Backends unterstützt (vgl. [29]).

Auf jedem Fall muss die verwendete Datenbank vor der Installation des OpenLDAP-Systems installiert werden. Eine weitere Konfiguration der DB ist nicht nötig, alle nötigen Einstellungen erfolgen durch die Konfigurationsdatei des LDAP-Servers, auch muss die Datenbank nicht separat gestartet werden.

```
[38]>cd $HOME/nexus/src

>gunzip db-4.1.24.tar.gz
>tar xf db-4.1.24.tar

>cd db-4.1.24/build_unix/

>env CC=gcc ../dist/configure --prefix=$HOME/nexus

>make

>make install

>LD_LIBRARY_PATH=$HOME/nexus/lib; export LD_LIBRARY_PATH
```

A.2 OpenLDAP

Das OpenLDAP-System[39] wird in der Version 2.1.8 verwendet.

A.2.1 Installation

Zur Installation der Software sind folgende Anweisungen nötig:

```
>cd $HOME/nexus/src

>gunzip openldap-2.1.8.tgz
>tar xf openldap-2.1.8.tar

>cd openldap-2.1.8/

>env CC=gcc CPPFLAGS="-I$HOME/nexus/include
  -I/usr/local/share/libtool/libltdl"
  LDFLAGS="-L$HOME/nexus/lib -L/usr/local/lib"
  ./configure --prefix=$HOME/nexus --enable-modules
```

37) http://www.sleepycat.com/download.html

38) Jede vollständige Kommandozeile ist durch einen Prompt „**>**“ gekennzeichnet. Alle folgenden Zeilen ohne Kennzeichnung müssen ohne Trennung durch einen Zeilenumbruch mit eingegeben werden.

39) http://www.openldap.org

```
>make depend

>make

>make install
```

Wichtig dabei ist die Option „**--enable-modules**“, die Voraussetzung für die Verwendung von dynamisch zu ladenden Erweiterungen ist. Deshalb müssen auch die Verweise auf die Header-Datei (**ltdl.h**) und Bibliothek (**libltdl**) zu „**libtool**“ mit angegeben werden.

Der in der Installationsanweisung beschriebene Test („**make test**“) würde erst nach einer Änderung der Test-Skripte funktionieren, da die von der verwendeten Berkeley DB angelegten Dateien lokal liegen müssen und nicht über NFS angeschlossen sein dürfen, wie es für die Benutzerverzeichnisse der Fall ist.

A.2.2 Konfiguration

Alle wichtigen Einstellungen für den Betrieb des OpenLDAP-Servers erfolgen durch eine Konfigurationsdatei (**$HOME/nexus/conf/nexus_sldap.conf**), die beim Start des OpenLDAP-Servers angegeben wird (vgl. Anhang B.1). Ausführliche Hinweise zu den den in dieser Datei möglichen Einstellungen finden sich in [29] und [36] oder den *man*-Pages, die bei der Installation des OpenLDAP-Servers mit installiert werden.

Hier ist das ladbare Modul (**nexusAreaSyntax**) anzugeben. Dann erfolgt die Definition des LDAP-Standard-Schemas und der Nexus-Erweiterung. Die übrigen Einstellungen betreffen die Konfiguration der Backend-Datenbank[40] und die Zugriffsrechte der Benutzer.

A.2.3 Betrieb

Der LDAP-Server sollte mit folgendem Befehl gestartet werden:

```
>$HOME/nexus/libexec/slapd -f
  $HOME/nexus/conf/nexus_slapd.conf -h ldap://localhost:9009
```

Die saubere Beendigung des LDAP-Servers erfolgt durch:

```
>kill -INT $(cat $HOME/nexus/var/slapd.pid)
```

Vor dem Anlegen des Nexus-Verzeichnisses muss zuallererst der interne Wurzelknoten des Verzeichnisbaums angelegt (**"dn: o=nexus"**, vgl. Abb. 6). Dies erfolgt durch die Anweisung:

```
>$HOME/nexus/bin/ldapadd -x -h localhost -p 9009
  -D "cn=admin,o=nexus" -W
  -f ~/nexus/conf/nexus_root-element.ldif
```

40) Das mit der direktive "directory" angegeben Verzeichnis muss vor dem Start des OpenLDAP-Servers existieren und muss lokal auf dem Rechner liegen, auf dem der OpenLDAP-Server (und damit auch die Berkeley-DB läuft). Per NFS angeschlossene Massenspeicher werden nicht unterstützt und verhindern den erfolgreichen Start den OpenLDAP-Servers.

Dabei ist das Passwort aus der OpenLDAP-Konfigurationsdatei anzugeben (**secret**).

A.2.4 Benutzerverwaltung

Die Rechteverwaltung erfolgt durch den OpenLDAP-System selbst. Die Einstellungen in der Konfigurationsdatei wurden dabei so gewählt, dass für den lesenden Zugriff auf das Verzeichnis kein Passwort notwendig ist. Für schreibende Zugriffe, also das Einfügen neuer Einträge oder das Ändern der bestehenden Einträge sind alle Personen berechtigt, die im Verzeichnis eingetragen sind.

Um einen neuen Benutzer anzulegen, muss eine Datei nach dem Schema der Datei (**$HOME/nexus/conf/nexus_add-user.conf**) angelegt werden. Dabei sind als **cn** und **userPassword** die Wörter anzugeben, die später als Argument **user** bzw. **passwd** beim Aufruf der Funktionen des NexusLDAPClient (vgl. Kap. 4.4) benutzt werden. Der distinguished name (DN) in der ersten Zeile muss exakt mit dem gewählten Wert für **cn** übereinstimmen. Im Feld **sn** kann dann ein beliebiger ausführlicher Name verwendet werden, dieser hat sonst keine weitere Bedeutung.

```
1    dn: cn=testuser,o=nexus
2    objectclass: person
3    cn: testuser
4    sn: Mustermann
5    userPassword: xxx
```

Die so angepasste Datei wird dann mit dem Befehl

```
>$HOME/nexus/bin/ldapadd -x -h localhost -p 9009
  -D "cn=admin,o=nexus" -W -f ~/nexus/conf/nexus_add-user.ldif
```

an den OpenLDAP-Server übergeben. Dabei ist dann bei Aufforderung das Passwort des Administrators anzugeben (**secret**). Neben dem Administrator sind allerdings auch alle anderen zuvor angelegten Benutzer berechtigt, neue Benutzer einzuführen.

Die Passwörter der so angelegten Benutzer können dann mit dem OpenLDAP-Kommandozeilen-Werkzeug **ldappasswd** geändert werden.

A.3 GEOS

Zum Test der geometrischen Überlappung von NexusArea und Anfrageregion wird die GEOS[41] (Geometry Engine Open Source) Bibliothek in der Version vom 12.03.2003 verwendet. Dies ist eine C++-Portierung der Java Topology Suite (JTS)[42].

Die aktuellen Quellen sind durch folgende Anweisungen per Anonymous-CVS zu beziehen: (als Passwort ist dabei „**cvs**“anzugeben)

```
>cd $HOME/nexus/src

>cvs -d ":pserver:cvs@geos.refractions.net:/home/cvs/postgis"
  login
```

41) http://geos.refractions.net/

42) http://www.vividsolutions.com/jts/jtshome.htm

```
>cvs -d
  ":pserver:cvs@geos.refractions.net:/home/cvs/postgis"
  checkout geos

>cd geos
```

Wegen eines Fehlers in den Library-Makefiles lässt sich das Paket allerdings nicht ohne Korrekturen installieren. Im Archiv „**`geos.tar.gz`**“sind diese Änderungen enthalten:

1. In der Datei „**`source/headers/Makefile.in`**“ ist der Eintrag „**`geosdir`**“ zu ändern. Nur auf diese Weise werden die Header-Dateien im gewünschten Verzeichnis (**`$HOME/nexus/include`**) installiert.
2. Die Makefiles sind durch Ausführung des Skripts „**`autogen.sh`**“ neu zu erzeugen. Dies funktioniert allerdings nur auf einem System mit aktualisierter Entwicklungsumgebung (Automake[43] Version 1.6.3, Libtool Version 1.4.3), das mit der Unterverzeichnisstruktur der Projektdateien zurechtkommt:

Die restliche Installation kann dann entsprechend der normalen Installationsanweisungen durchgeführt werden:

```
>env CC=gcc ./configure --prefix=$HOME/nexus

>make

>make install
```

A.4 Projektkomponenten

Nach der Fremdsoftware sind noch neu entwickelten Komponenten zu übersetzen und zu installieren. Dies betrifft die Pakete:

- geosWrapper
- nexusAreaSyntax
- nexusLDAPClient

Die Installation verläuft in allen Fällen identisch. Für jede Komponente bestehen neben den Quell-Dateien noch je eine Datei **`"configure.in"`** bzw. **`"Makefile.am"`**, aus denen durch die Werkzeuge *Autoconf* bzw. *Automake* die jeweils spezifischen Make-Dateien erzeugt werden können. Die dabei benötigten Anweisungen mit den genauen Parametern finden sich in der Datei mit der Endung **`"install"`**.

43) http://www.gnu.org

Anhang B: Listings

B.1 Konfigurationsdatei: nexus_slapd.conf

```
1    # ******************************************************************************
2    #
3    # * Universitaet Stuttgart, IPVS
4    # *
5    # * Studienarbeit: Standard-Verzeichnisdienste als raeumliches Verzeichnis
6    # *                fuer Informationsdienste in NEXUS
7    # *
8    # * Autor:         Carsten Neumann
9    # *
10   # * Stand:         12.05.2003
11   # *
12   # * conf/nexus_slapd.conf
13   # * ====================
14   # *
15   # * Konfiguration des OpenLDAP-Servers: Allgemeine Einstellungen
16   # * Diese Datei muss beim Start den OpenLDAP-Server mit angegeben werden
17   # *
18   # ******************************************************************************
19
20   #
21   #  Diese Datei enthaelt absolute Pfadangaben, die evtl. angepasst werden muessen
22   #
23
24   ################################### Allgemeines ###################################
25
26
27   pidfile       /home/neumancn/nexus/var/slapd.pid
28   argsfile      /home/neumancn/nexus/var/slapd.args
29
30   # loglevel      65535
31   loglevel      0
32   schemacheck   on
33
34   # default: 500
35   sizelimit     -1
36
37   modulepath    /home/neumancn/nexus/lib
38   moduleload    nexusAreaSyntax.la
39
40   include       /home/neumancn/nexus/etc/openldap/schema/core.schema
41   include       /home/neumancn/nexus/conf/nexus_slapd.schema
42
43   ################################### Datenbank ###################################
44
45   database      bdb
46
```

```
47    directory     /tmp/openldap-data
48
49    suffix        "o=nexus"
50
51    rootdn        "cn=admin,o=nexus"
52    rootpw        secret
53
54    index  objectClass  eq
55    index  nt           eq
56    index  nal          eq
57
58    ##################################### Rechte ########################################
59
60    access to filter="cn=*" attr=userPassword
61      by self write
62      by * auth
63
64    access to filter="o=*"
65      by dn="cn=(.*),o=nexus" write
66      by * none
67
68    access to filter="nt=*"
69      by dn="cn=(.*),o=nexus" write
70      by * read
71
72    access to filter="nal=*"
73      by dn="cn=(.*),o=nexus" write
74      by * read
```

B.2 Schemaerweiterung: nexus_slapd.schema

```
1    # ******************************************************************************
2    #
3    # * Universitaet Stuttgart, IPVS
4    # *
5    # * Studienarbeit: Standard-Verzeichnisdienste als raeumliches Verzeichnis
6    # *                fuer Informationsdienste in NEXUS
7    # *
8    # * Autor:         Carsten Neumann
9    # *
10   # * Stand:         14.04.2003
11   # *
12   # * conf/nexus_slapd.schema
13   # * =======================
14   # *
15   # * Konfiguration des OpenLDAP-Servers: Erweiterung des Servers um neue
16   # * Attributtypen und Objektklassen.
17   # *
18   # ******************************************************************************
19
20
21   ### neue Attributtypen ###
22
23
24   attributetype (
25     1.1.2.1
26     NAME 'nt'
27     DESC 'NexusType'
28     EQUALITY caseIgnoreMatch
29     SUBSTR caseIgnoreSubstringsMatch
30     SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
31     SINGLE-VALUE)
32
33   attributetype (
34     1.1.2.2
35     NAME 'nal'
36     DESC 'NexusAreaLocator'
37     EQUALITY caseIgnoreMatch
38     SUBSTR caseIgnoreSubstringsMatch
39     SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
40     SINGLE-VALUE)
41
42   attributetype (
43     1.1.2.3
44     NAME 'na'
45     DESC 'NexusArea'
46     EQUALITY nexusAreaEqual
47     SYNTAX 1.1.3.1
48     SINGLE-VALUE)
49
50
51   attributetype (
52     1.1.2.4
53     NAME 'nlod'
```

```
54      DESC 'NexusLevelOfDetail'
55      SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
56      EQUALITY integerMatch
57      ORDERING integerOrderingMatch
58      SINGLE-VALUE)
59
60    attributetype (
61      1.1.2.5
62      NAME 'na3d'
63      DESC 'Nexus3DArea'
64      EQUALITY caseIgnoreMatch
65      SUBSTR caseIgnoreSubstringsMatch
66      SYNTAX 1.3.6.1.4.1.1466.115.121.1.15
67      SINGLE-VALUE)
68
69
70    ### neue Objectklassen ###
71
72
73    objectclass (
74      1.1.1.1
75      NAME 'ntn'
76      DESC 'NexusTypeNode'
77      MUST nt )
78
79    objectclass (
80      1.1.1.2
81      NAME 'nan'
82      DESC 'NexusAreaNode'
83      MUST ( nal $ na $ nlod $ na3d ) )
```

B.3 Header-Datei: nexusAreaSyntax.h

```
1    /*******************************************************************************
2     *
3     * Universitaet Stuttgart, IPVS
4     *
5     * Studienarbeit: Standard-Verzeichnisdienste als raeumliches Verzeichnis
6     *                fuer Informationsdienste in NEXUS
7     *
8     * Autor:         Carsten Neumann
9     *
10    * Stand:         07.04.2003
11    *
12    * src/nexusAreaSyntax/nexusAreaSyntax.h
13    * =====================================
14    *
15    * Header-Datei fuer das ladbare Modul zur Erweiterung des OpenLDAP-Servers
16    * um neue Syntax und zugehoerige Funktionen (Validierung und Vergleich).
17    *
18    *******************************************************************************/
19
20   #ifndef NEXUSAREASYNTAX_H
21   #define NEXUSAREASYNTAX_H
22
23
24   #include "portable.h"
25
26   #include <stdio.h>
27   #include <string.h>
28   #include <ac/errno.h>
29
30   #include "slap.h"
31   #include "ldap_log.h"
32
33   #include "geosWrapper.h"
34
35
36   int nexusAreaSyntax_LTX_init_module ( int argc, char *argv[] );
37
38   static int   nexusAreaSyntaxValidate(
39     Syntax *syntax,
40     struct berval *val
41   );
42
43   static int   nexusAreaMROverlaps(
44     int *matchp,
45     slap_mask_t flags,
46     Syntax *syntax,
47     MatchingRule *mr,
48     struct berval *value,
49     void *assertedValue
50   );
51
52   static int nexusAreaMREqual(
53     int *matchp,
```

```
54      slap_mask_t flags,
55      Syntax *syntax,
56      MatchingRule *mr,
57      struct berval *value,
58      void *assertedValue
59    );
60
61    /*************************************************************************/
62
63    /* aus 'slap.h' */
64
65    /*
66    typedef struct slap_syntax_defs_rec {
67      char *sd_desc;
68      int sd_flags;
69      slap_syntax_validate_func      *sd_validate;
70      slap_syntax_transform_func     *sd_normalize;
71      slap_syntax_transform_func     *sd_pretty;
72    #ifdef SLAPD_BINARY_CONVERSION
73      slap_syntax_transform_func     *sd_ber2str;
74      slap_syntax_transform_func     *sd_str2ber;
75    #endif
76    } slap_syntax_defs_rec;
77    */
78
79    static slap_syntax_defs_rec nexus_syntax_defs[] = {
80
81      // Nexus Syntax Types
82      {"( 1.1.3.1 DESC 'Nexus Area Syntax' )" ,
83        0, nexusAreaSyntaxValidate, NULL, NULL},
84
85      {NULL, 0, NULL, NULL, NULL}
86
87    };
88
89    /*************************************************************************/
90
91    /* aus 'slap.h' */
92
93    /*
94    typedef struct slap_mrule_defs_rec {
95      char *                        mrd_desc;
96      slap_mask_t                   mrd_usage;
97      char **                       mrd_compat_syntaxes;
98      slap_mr_convert_func *        mrd_convert;
99      slap_mr_normalize_func *      mrd_normalize;
100     slap_mr_match_func *          mrd_match;
101     slap_mr_indexer_func *        mrd_indexer;
102     slap_mr_filter_func *         mrd_filter;
103     char *                        mrd_associated;
104   } slap_mrule_defs_rec;
105   */
106
107
108   static slap_mrule_defs_rec nexus_mrule_defs[] = {
109
110     // Nexus Matching Rules
```

```
111
112     {"( 1.1.4.1 NAME 'nexusAreaOverlaps' "
113       "SYNTAX 1.1.3.1 )",
114       SLAP_MR_EXT, NULL,
115       NULL, NULL,
116       nexusAreaMROverlaps, NULL, NULL,
117       NULL},
118
119     {"( 1.1.4.2 NAME 'nexusAreaEqual' "
120       "SYNTAX 1.1.3.1 )",
121       SLAP_MR_EQUALITY, NULL,
122       NULL, NULL,
123       nexusAreaMREqual, NULL, NULL,
124       NULL},
125
126     {NULL, SLAP_MR_NONE, NULL,
127       NULL, NULL, NULL, NULL, NULL,
128       NULL }
129
130   };
131
132   #endif // NEXUSAREASYNTAX_H
```

B.4 Header-Datei: geosWrapper.h

```
1    /*************************************************************************
2     *
3     * Universitaet Stuttgart, IPVS
4     *
5     * Studienarbeit: Standard-Verzeichnisdienste als raeumliches Verzeichnis
6     *                fuer Informationsdienste in NEXUS
7     *
8     * Autor:         Carsten Neumann
9     *
10    * Stand:         07.04.2003
11    *
12    * src/geosWrapper/geosWrapper.h
13    * =============================
14    *
15    * Header-Datei fuer die Library zur Kapselung der GEOS-Library. Diese stellt
16    * C-aufrufbare Funktionen zur Nutzung der in C++ geschriebenen GEOS-Library
17    * bereit.
18    *
19    *************************************************************************/
20
21   #ifndef GEOSWRAPPER_H
22   #define GEOSWRAPPER_H
23
24
25   #if defined(__cplusplus)
26   extern "C" {
27   #endif
28
29     int gw_testPolygonString( const char *wkt );
30
31     int gw_testPolygonOverlap( const char *wktA, const char *wktB );
32
33   #if defined(__cplusplus)
34   }
35   #endif
36
37   #endif // GEOSWRAPPER_H
```

B.5 Header-Datei: nexusLDAPClient.h

```
1    /*************************************************************************
2     *
3     * Universitaet Stuttgart, IPVS
4     *
5     * Studienarbeit: Standard-Verzeichnisdienste als raeumliches Verzeichnis
6     *                fuer Informationsdienste in NEXUS
7     *
8     * Autor:         Carsten Neumann
9     *
10    * Stand:         14.04.2003
11    *
12    * src/nexusLDAPClient/nexusLDAPClient.h
13    * =====================================
14    *
15    * Header-Datei fuer die nexusLDAPClient-Library.
16    * Diese stellt Funktionen zur Verwaltung und Abfrage des raeumlichen Verzeich-
17    * nisses bereit.
18    *
19    *************************************************************************/
20
21   #ifndef NEXUSLDAPCLIENT_H
22   #define NEXUSLDAPCLIENT_H
23
24   #include <stdio.h>
25   #include <stdlib.h>
26   #include <string.h>
27   #include <errno.h>
28   #include <ctype.h>
29
30   #include "ldap.h"
31
32
33   #define NEXUSLDAPCLIENT_SUCCESS                          0
34
35   #define NEXUSLDAPCLIENT_LDAP_INIT_FAILED                 1
36   #define NEXUSLDAPCLIENT_LDAP_BIND_FAILED                 2
37   #define NEXUSLDAPCLIENT_LDAP_SEARCH_FAILED               3
38   #define NEXUSLDAPCLIENT_LDAP_ADD_FAILED                  4
39   #define NEXUSLDAPCLIENT_LDAP_MODIFY_FAILED               5
40   #define NEXUSLDAPCLIENT_LDAP_DELETE_FAILED               6
41
42   #define NEXUSLDAPCLIENT_PARAMETER_UNDEFINED              7
43   #define NEXUSLDAPCLIENT_ILLEGAL_CHARACTER                8
44   #define NEXUSLDAPCLIENT_OUT_OF_MEMORY                    9
45
46   #define NEXUSLDAPCLIENT_SUPERNODE_MISSING               10
47   #define NEXUSLDAPCLIENT_NODE_HAS_SUBNODES               11
48   #define NEXUSLDAPCLIENT_NODE_EXISTS                     12
49   #define NEXUSLDAPCLIENT_NODE_DOES_NOT_EXIST             13
50
51
52
53   typedef struct {
```

```
54      char              *superTypeName;
55      char              *typeName;
56    } TypeEntry;
57
58    typedef struct {
59      char              *nexusAreaLocator;
60      char              *nexusType;
61      char              *nexusArea;
62      int               nexusLevelOfDetail;
63    } AugmentedArea;
64
65
66    int createType(
67      const char *user, const char *passwd,
68      const char *newType, const char *superType );
69
70    int dropType(
71      const char *user, const char *passwd, const char *oldType );
72
73    TypeEntry *listTypes();
74
75    int freeTypeList( TypeEntry *typeList );
76
77
78    int insertArea(
79      const char *user, const char *passwd, AugmentedArea *newArea );
80
81    int updateArea(
82      const char *user, const char *passwd, AugmentedArea *newArea );
83
84    int deleteArea(
85      const char *user, const char *passwd, const char *nal );
86
87    AugmentedArea *query(
88      const char *type, const char *area, int lod_min, int lod_max );
89
90    int freeAreaList( AugmentedArea *areaList );
91
92
93    #endif // NEXUSLDAPCLIENT_H
```

Literaturverzeichnis

[1] ITU-T Recommendation X.500 (2001) | ISO/IEC 9594-1: Information technology – Open Systems Interconnection – The Directory: Overview of concepts, models and services, 2001

[2] ITU-T Recommendation X.501 (2001) | ISO/IEC 9594-2: Information technology – Open Systems Interconnection – The Directory: Models, 2001

[3] ITU-T Recommendation X.509 (2001) | ISO/IEC 9594-8: Information technology – Open Systems Interconnection – The Directory: Public-Key and Attribute Certificate Frameworks, 2001

[4] ITU-T Recommendation X.511 (2001) | ISO/IEC 9594-3: Information technology – Open Systems Interconnection – The Directory: Abstract Service Definition. 2001

[5] ITU-T Recommendation X.518 (2001) | ISO/IEC 9594-4: Information technology – Open Systems Interconnection – The Directory: Procedures for distributed Operations, 2001

[6] ITU-T Recommendation X.519 (2001) | ISO/IEC 9594-5: Information technology – Open Systems Interconnection – The Directory: Protocol Specifications, 2001

[7] ITU-T Recommendation X.520 (2001) | ISO/IEC 9594-6: Information technology – Open Systems Interconnection – The Directory: Selected Attribute types, 2001

[8] ITU-T Recommendation X.521 (2001) | ISO/IEC 9594-7: Information technology – Open Systems Interconnection – The Directory: Selected object classes, 2001

[9] ITU-T Recommendation X.525 (2001) | ISO/IEC 9594-9: Information technology – Open Systems Interconnection – The Directory: Replication, 2001

[10] ITU-T Recommendation X.530 (2001) | ISO/IEC 9594-10: Information technology – Open Systems Interconnection – The Directory: Use of system management for Administration of the Directory, 2001

[11] Request for Comments (RFC) 2251: Lightweight Directory Access Protocol (v3) – The specification of the LDAP on-the-wire protocol, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2251.txt

[12] Request for Comments (RFC) 2252: Lightweight Directory Access Protocol (v3) – Attribute Syntax Definition, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2252.txt

[13] Request for Comments (RFC) 2253: Lightweight Directory Access Protocol (v3) – UTF-8 String Representation of Distinguished Names, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2253.txt

[14] Request for Comments (RFC) 2254: Lightweight Directory Access Protocol (v3) – The String Representation of LDAP Search Filters, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2254.txt

[15] Request for Comments (RFC) 2255: Lightweight Directory Access Protocol (v3) – The LDAP URL-Format, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2255.txt

[16] Request for Comments (RFC) 2256: Lightweight Directory Access Protocol (v3) – A Summary of the X.500(96) User Schema for use with LDAPv3, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc2256.txt

[17] Request for Comments (RFC) 2829: Lightweight Directory Access Protocol (v3) – Authentication Methods for LDAP, 1999, ftp://ftp.rfc-editor.org/in-notes/rfc2829.txt

[18] Request for Comments (RFC) 2830: Lightweight Directory Access Protocol (v3) – Extensions for Transport Layer Security, 1999. ftp://ftp.rfc-editor.org/in-notes/rfc2830.txt

[19] Request for Comments (RFC) 3377: Lightweight Directory Access Protocol (v3) – Technical Specification, 2002, ftp://ftp.rfc-editor.org/in-notes/rfc3377.txt

[20] Request for Comments (RFC) 1034: Domain Names – Concepts and Facilities, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc1034.txt

[21] Request for Comments (RFC) 1035: Domain Names – Implementation and Specification, 1997, ftp://ftp.rfc-editor.org/in-notes/rfc1035.txt

[22] Greenblatt, Bruce: Internet Directories – How to Build and Manage Application for LDAP, DNS and Other Directories, Prentice Hall, 2001

[23] Howes, Timothy A.; Smith, Mark C.; Good, Gordon S.: Understanding and Deploying LDAP Directory Services, New Riders Publishing, 2001

[24] Maaß, Henning: Open Mobility Platform based on Dynamic Directory Services, Shaker Verlag, Aachen 2002

[25] Nicklas, Daniela; Großmann, Matthias; Schwarz, Thomas; Volz, Steffen; Mitschang, Bernhard: A Model-Based, Open Architecture for Mobile, Spatially Aware Applications, Universität Stuttgart, 2001

[26] Nicklas, Daniela; Großmann, Matthias; Schwarz, Thomas; Volz, Steffen: Information Management and Exchange in Nexus – Technical Report V1.4, Universität Stuttgart, 2002

[27] Binder, Thomas: LDAP und X.500, TU Darmstadt, 2000
http://www.ito.tu-darmstadt.de/edu/sem-iuk-w99/ldap-x500.pdf

[28] Novell: Which directory offers the best LDAP server?, White Paper, 2001
http://www.novell.com/info/collateral/docs/4621218.01/4621218.pdf

[28] Netscape Directory Server Plug-In Programmer's Guide, 2002
http://enterprise.netscape.com/docs/directory/61/pdf/ds61plugin.pdf

[29] OpenLDAP 2.1 Administrator's Guide,
http://www.openldap.org/doc/admin21/guide.html

[30] YoLinux LDAP Tutorial: OpenLDAP Directory Objects and Attributes - Add new LDAP object and attribute definitions to your OpenLDAP (2.0) directory,
http://www.yolinux.com/TUTORIALS/
LinuxTutorialLDAP-DefineObjectsAndAttributes.html

[31] Steffen Brandhorst: Räumliches Verzeichnis für Informationsangebote in NEXUS, Diplomarbeit Nr. 1993, Universität Stuttgart, 2002

[32] OpenGIS Simple Features Specification for SQL, Revision 1.1
http://www.opengis.org/techno/specs/99-049.pdf

[33] DB2 Administration Guide and Reference
Appendix B. The OGIS Well-Known Text Representation
http://www-3.ibm.com/software/data/datajoiner/booksv2/djxs1121.htm

[34] Judith R. Davis: IBM's DB2 Spatial Extender: Managing Geo-Spatial Information within the DBMS, IBM Corporation, May 1998
http://www-4.ibm.com/software/data/pubs/papers/spatial/spatial.pdf

[35] Theo Härder, Erhard Rahm: Datenbanksysteme – Konzepte und Techniken der Implementierung, Springer Verlag, 1999

[36] Jens Banning: LDAP unter Linux – Netzwerkinformationen in Verzeichnis-diensten verwalten, Addison-Wesley, 2001

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Carsten Neumann)