Konzeption und Implementierung eines
Java-Backends für einen LDAP-Server

DIPLOMARBEIT

zur Erlangung des akademischen Grades
Diplomingenieur (FH)
an der Fachhochschule für Technik und Wirtschaft Berlin

Fachbereich Ingenieurwissenschaften I
Studiengang Technische Informatik

Betreuer: Prof. Dr. Johann Schmidek
Dr. Thomas Schmidt
Eingereicht von: Oliver Schönherr

Berlin, 13. Juli 1999

Inhaltsverzeichnis

1. Einleitung

Die Tatsache, daß die Vernetzung innerhalb der Computerwelt immer mehr expandiert, erfordert eine transparente Erreichbarkeit aller Informationen über die Ressourcen des Netzes. Diese Notwendigkeit wird am Beispiel einer größeren Institution, nämlich der FHTW Berlin, verdeutlicht. Hier stellt die Zusammenführung von Logins aus verschiedenen „Rechnerwelten“ (z.B. Unix und NetWare) in einen einheitlichen Authentisierungsraum ein spezielles Problem dar.

Für diese u.ä. Aufgaben wurden sogenannte „Verzeichnisdienste“ konzipiert. Eine Implementation eines solchen Dienstes stellt das „Lightweight Directory Access Protocol“ (LDAP) dar.

Aber besonders die Integration vorhandener Datenquellen erweißt sich als schwierig. Deswegen trägt diese Arbeit die Zielstellung, diese Problematik durch ein Java-Backend für einen LDAP-Server zu vereinfachen.

Die Arbeit gliedert sich in folgende Teile:

In Kapitel 2 wird eine Einführung in das Thema Verzeichnisdienste mit dem Schwerpunkt LDAP gegeben. Außerdem wird der von mir genutzte OpenLDAP-Server vorgestellt.

Das Kapitel 3 enthält einen knappen Exkurs über die Verwendung der Programmiersprache Java im Server-Umfeld. Desweiteren wird ein Überblick über die Schnittstelle zwischen den Programmiersprachen Java und C gegeben.

Kapitel 4 beschreibt die Problemanalyse und den Entwurf des Java-Backends für den OpenLDAP-Server. Im weiteren Verlauf der Arbeit wird in Kapitel 5 auf die Implementierung des Java-Backends eingegangen. In Kapitel 6 wird anhand einer Beispielanwendung das Java-Backend konzeptionell überprüft und getestet.

Eine Zusammenfassung der in dieser Arbeit gewonnenen Erkenntnise befindet sich in Kapitel 7. Der Ausblick auf zukünftige Entwicklungen befindet sich in Kapitel 8.

2. Verzeichnisdienste

2.1. Überblick

In diesem ersten Abschnitt möchte ich eine Einführung und einen Überblick über das Thema Verzeichnisdienste geben; dem in dieser Arbeit verwendeten Verzeichnisdienst LDAP wird ein spezielles Kapitel gewidmet.

Verzeichnisdienste sind Auskunftsdienste. Sie stellen Informationen über beliebige Objekte zur Verfügung, z.B. Drucker, Rechner, Modems, Länder, Organisationen oder Personen. Zusätzlich bieten Verzeichnisdienste die Möglichkeit, solche Objekte durch Namen benutzerfreundlich zu identifizieren. Der angebotene Funktionsumfang entspricht damit etwa dem der „weißen“ und der „gelben“ Seiten des Telefonbuchs (entnommen [26]).

Verzeichnisdienste sind keine Entwicklung der späten 90er Jahre, sondern schon seit Mitte der 80er bekannt. Hier trat erstmalig das Problem der zentralen Bereitstellung von Informationen in Computernetzwerken auf. Seit dieser Zeit entwickelten sich drei Hauptformen von Verzeichnisdiensten:

- Applikationsspezifisch
- Plattformabhängig
- Allgemeingültig

In der Tabelle 2.1 sind diesen Formen entsprechende Verzeichnisdienste zugeordnet. In den folgenden Kapiteln werden ihre Funktionsweisen, Einsatzgebiete und Unterschiede genauer betrachtet.

2.1.1. Applikationsspezifische Verzeichnisdienste

Applikationsspezifische Verzeichnisdienste wurden für speziell, abgegrenzte Aufgabengebiete konzipiert. Damit konnte ein Optimum zwischen Funktionalität und Ressourcenbedarf erzielt werden. Der zur Zeit wohl am häufigsten benutzte applikationsspezifische Verzeichnisdienst ist der „Domain Name Service“ (DNS), welcher im Internet hauptsächlich für die Umsetzung von IP-Adressen in Rechnernamen zuständig ist. Dazu wurde mit DNS-Servern eine weltweit verteilte Datenbank aufgebaut. Aber auch einfacher aufgebaute Informationsdienste, wie z.B. die `/etc/alias` Datei des Mailservers „sendmail“können als Verzeichnisdienst angesehen werden. In ihr können Mailnutzern verschiedene Mailadressen zugeordnet werden.

Die Schwäche applikationsspezifischer Verzeichnisdienste wird im Zusammenspiel der beiden Beispiele deutlich: So werden Mailserver im DNS durch „MX-Records“ (Mailexchanger) spezifiziert. Der Mailserver seinerseits nutzt außerdem die erwähnte `/etc/alias` Datei, die darin verwendeten Nutzer entstammen aber einem weiteren, nämlich dem Nutzerverzeichnisdienst (z.B. der `/etc/passwd`). Daraus können sehr schnell Inkonsistenzen durch die mehrfache Verwaltung von Einträgen entstehen. Ein Beispiel hierfür wäre der Eintrag für einen Nutzer in der `/etc/alias` Datei, welcher in der `/etc/passwd` (und damit im System) gar nicht mehr existiert.

2.1.2. Plattformabhängige Verzeichnisdienste

Die plattformabhängigen Verzeichnisdienste entstammen einer Zeit, in der eine starke Abgrenzung zwischen Betriebssystem-Plattformen und unterschiedlichen Einsatzgebieten üblich war.

Applikationsspezifisch	**Plattformabhängig**	**Allgemeingültig**
DNS	NIS	NDS
/etc/alias	Netware Bindery	X.500

Tabelle 2.1.: Formen von Verzeichnisdiensten

Network Information Service

So ist zwar der im Unix-Bereich etablierte Verzeichnisdienst „Network Information Service“ (NIS) [27] der Firma Sun plattformoffen konzipiert, aber doch für den Einsatz im Unix-Umfeld optimiert. Z.B. stützt er sich in seinem Sicherheitskonzept stark auf ein sicheres Client-Betriebssystem, wovon im Fall von Unix ausgegangen werden kann, aber beim Einsatz von MS-DOS z.B. nicht. Außerdem beschränkt sich seine Hierarchisierung auf nur eine Stufe — die NIS-Domains. Dies ist für einen Einsatz in großen Netzwerken bis hin zu einem weltweiten Verbund eine nicht akzeptable Einschränkung. Auch der verwendete Replikationsmechanismus zeigt die konzeptionelle Beschränkung dieses Verzeichnisdienstes auf kleinere Netzwerke mit bis zu 10.000 Nutzern. So wird im Fall einer Änderung in einem Datensatz (beispielsweise ändert ein Nutzer sein Passwort) die komplette Nutzer-Passwort-Datenbank mit allen (auch den nicht geänderten) Einträgen an die zur Absicherung des Betriebes benutzten Slave-Server übertragen.

NetWare Bindery

Die Bindery [28] wurde mit der NetWare Version 2 eingeführt und in der Version 3 nocheinmal weiterentwickelt. Sie basiert auf einer flachen Datenbankstruktur ohne Indizierung, was zur Folge hat, daß bei großer Nutzeranzahl der Zugriff langwierig ist. Außerdem gibt es kein Domain- und kein Replikationskonzept. Dies führt dazu, daß mehrere Server im Netzwerk von Hand abgeglichen werden müssen. Die daraus entstehenden Einschränkungen hat Novell mit der Einführung der Novell Directory Services für NetWare 4 behoben.

2.1.3. Allgemeine Verzeichnisdienste

X.500

Die in den vorhergehenden Kapiteln erläuterten Einschränkungen von plattformabhängigen oder applikationsspezifischen Verzeichnisdiensten führten schon in den späten 80er Jahren dazu, daß im Rahmen der ITU der weltweite und zentrale Verzeichnisdienst X.500 konzipiert und entwickelt wurde. Wie schon das X im Namen vermuten läßt, ist er ein Bestandteil der OSI-Protokoll-Suite. Die große Komplexität dieses Protokolls führte jedoch zu sehr hohen Systemanforderungen. So war es bis vor kurzem nicht möglich, einen Standard-PC OSI-konform zu betreiben. Trotzdem gilt X.500 als Vater aller allgemeinen Verzeichnisdienste.

Novell Directory Service

Auch der „Novell Directory Service“ (NDS) [28] baut auf viele Konzepte von X.500 auf. Er ist z.B. konsequent hierarchisch strukturiert und benutzt zur Abbildung der Netzressourcen ein Objektmodell mit Vererbungsmechanismen. Sein wohl größter

Vorteil ist, daß er schon seit der Netware Version 4 (Einführung 1994/95) im weltweiten Einsatz ist. Dadurch hat er eine beträchtliche Verbreitung erzielt. Dem gegenüber steht die Abhängigkeit von der Firma Novell; zwar läßt Novell schon seit längerem verlautbaren, daß an einer Öffnung der NDS anderen Anbietern und Betriebssystemen gegenüber gearbeitet wird, aber außer einer NDS-Variante für Linux und Solaris ist nichts Konkretes erschienen. Besonders ändern beide Implementierungen nichts an der Tatsache, daß die NDS in der Hand eines Herstellers liegt und nicht offen in ihren Schnittstellen vorliegt.

2.2. Lightweight Directory Access Protocol

2.2.1. X.500 light

Ursprünglich war LDAP als leichtgewichtige Alternative zum X.500-Zugriffsprotokoll DAP gedacht. X.500 enthält Informationen über Objekte. Dabei kann es sich um reale Objekte wie Personen oder Geräte wie auch um logische Objekte wie Gruppen handeln. Jedes dieser Objekte hat ein oder mehrere Attribute und diesen zugeordnete Werte, z.B. ein Attribut „Name“ mit dem Wert „Musterman“. Die Objekte können in einer baumartigen Struktur eingeordnet werden.

Aus der Zugehörigkeit zu einer im Verzeichnisschema definierten Objektklasse ergeben sich die Standard-Attribute eines Objektes. Hierbei wird zwischen zwingend vorgeschriebenen und optionalen unterschieden. Der Wert eines bestimmten Attributes des Objektes dient als „Relative Distinguished Name“ (RDN), – beispielsweise „Michael Musterman“. Zusammen mit den RDNs der Knoten im Verzeichnisbaum, über die man von der Baumwurzel bis zum Objekt gelangt, ergibt sich der Distinguished Name (DN) des Objektes. Der DN des Objektes ist somit der Schlüssel und muß deshalb baumweit eindeutig sein.

Durch die Tatsache, daß alle Objekte einem Knoten im globalen X.500 Verzeichnisbaum zugeordnet sind, ergibt sich ein Baum, welcher sich von der Wurzel ausgehend in weitere Teilbäume für Länder, Standorte, Organisationen und organisatorische Einheiten aufteilt. Dabei ist jeder Knoten selbst ein Objekt mit Attributen. Die Notation führt somit zu DNs wie „c=DE, o=FHTW, ou=RZ, cn=Michael Musterman“. Dieser Distinguished Name legt die Person namens Michael Musterman (Common Name) in der organisatorischen Einheit RZ der Organisation FHTW in Deutschland (Country) fest (siehe Abbildung 2.1).

Mit der weiteren Verbreitung von LDAP-Servern im Internet wird sich eine weitere Variante der Bildung eines Distinguished Name, und zwar basierend auf den verwendeten Internet Domain-Namen, verbreiten. Zum Beispiel für die FHTW: „dc=fhtw-berlin, dc=de“ (siehe RFC2247 [20]).

Wie schon im vorhergehenden Kapitel besprochen wurde, basiert das X.500 Protokoll DAP (Directory Access Protocol) auf dem in der Praxis nicht sehr weit verbreiteten OSI-Standard. Als Alternative bot sich ein direkt auf TCP/IP basierendes

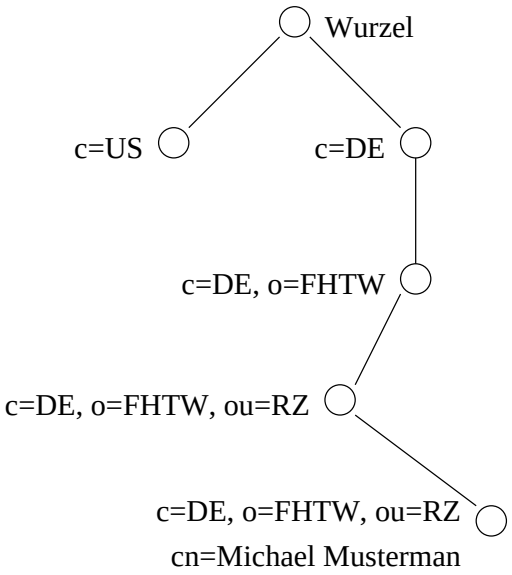

Abbildung 2.1.: Distinguished Name

Zugriffsprotokoll an, was schließlich zur Spezifikation [13] von LDAP führte. Diese Spezifikation von LDAP bietet drei grundlegende Vereinfachungen gegenüber DAP:

- LDAP setzt direkt auf TCP/IP auf.
- Die meisten Daten sind Textstrings, was zu einer einfacheren Kodierung der Daten für die Übertragung im Netzwerk führt.
- LDAP beschränkt sich auf die meistgenutzten Funktionen von X.500. Andere Funktionen lassen sich durch eine geschickte Parameterwahl simulieren.

Anfänglich war LDAP ausschließlich als Zugriffsprotokoll auf X.500-Server konzipiert. So entstanden LDAP-Server, welche Anfragen von Clients entgegennahmen und sie über DAP an einen X.500 Server weiterleiteten. Der LDAP-Server hielt also keine eigenen Daten. Doch sehr schnell erkannte man, daß ein LDAP-Server ohne den Ballast eines X.500-Servers als Datenquelle für viele Einsatzgebiete ausreichend wäre. Deswegen implementierte man an der University of Michigan einen solchen Standalone-Server. Dieser diente auch als Grundlage diverser kommerzieller Ableger, z.B. des Netscape Directory Server. Eine weitere Entwicklung geht dahin, daß Hersteller von proprietären Verzeichnisdiensten LDAP als mögliche Schnittstelle dazu anbieten. Zum Beispiel Novell mit der Unterstützung von LDAPv3 in Netware 5 als Zugriffsprotokoll auf die NDS [29]. Auch die Firma Microsoft wird in ihrem angekündigten Active Directory Service (ADS) [30] LDAPv3 als Kernprotoll verwenden.

Somit existieren drei Anwendungsgebiete für LDAP (siehe Abbildung 2.2):

- X.500-Zugriff über einen LDAP-Server (Protokollumsetzung).

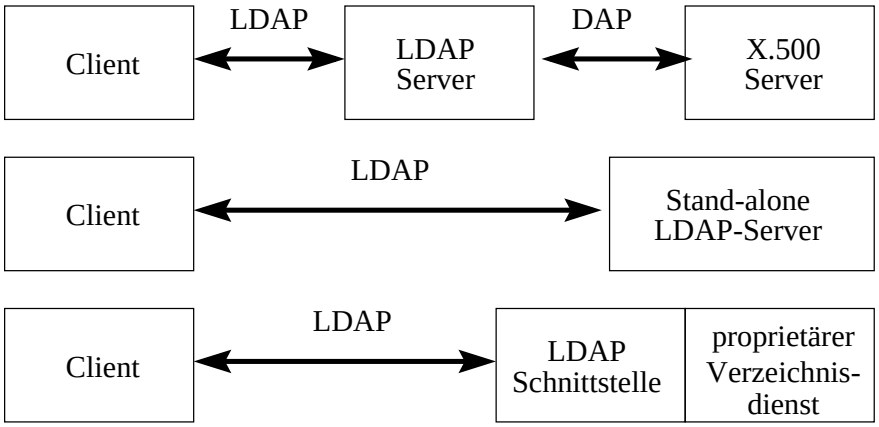

Abbildung 2.2.: LDAP Anwendungsgebiete

- Reiner LDAP-Betrieb mit Hilfe von Standalone-Servern.
- Zugriff auf proprietäre Verzeichnisdienste über LDAP-Schnittstelle.

2.2.2. LDAP-Operationen

LDAP-Verzeichniszugriffe lassen sich in drei Gruppen einordnen: Lesen, Schreiben und Verbinden. Ein Client muß sich beim Server zuerst mittels Bind authentifizieren. Dies erfolgt in der Version 2 von LDAP nur mit Hilfe eine Passwortes. Die neue Version 3 (LDAPv3 [21]) sieht auch andere Authentifizierungsmechanismen vor. Als Nutzer-Kennung dient der DN eines Objektes, dessen Passwort-Attribut abgefragt wird. Eine anonyme Authentifizierung kann durch das Weglassen von DN und Passwort erfolgen. LDAP kennt folgende weitere Operationen:

Search ermöglicht es dem Client, nach Objekten zu suchen, deren Attribute einem bestimmten Suchkriterium entprechen. LDAP-Server können sowohl phonetisch als auch nach Substrings suchen. Die dafür benutzte Syntax ist im RFC 1558 [14] spezifiziert.

Compare veranlaßt einen Server, die Übereinstimmung eines Vergleichswertes mit einem Attributwert im Server zu bestätigen oder zu verneinen. Dies bietet sich z.B. für Passwörter an.

Add fügt Einträge in das Verzeichnis ein. Parameter sind die Position im Verzeichnisbaum (DN) und die Attribute und Attributwerte der Einträge.

Delete löscht einen Eintrag im Verzeichnis.

Modify erlaubt es, Attribute existierender Einträge zu löschen, neue hinzuzufügen oder Attributwerte zu ändern.

ModifyRDN ändert den Relative Distinguished Name (RDN) eines Eintrages, in LDAPv3 können damit ganze Teilbäume verschoben werden.

Abandon ermöglicht das Abbrechen einer Suchanfrage, welche gerade vom Server durchgeführt wird.

2.2.3. Relationale Datenbanken oder LDAP

Nun kann als Gegenpol die Meinung vertreten werden, daß die Benutzung einer Relationalen Datenbank (RDBM) mit SQL als Abfragesprache auch diesen Ansprüchen entspricht. Aber es gibt Punkte, die gegen den Einsatz solcher RDBMs auf der Ebene von Verzeichnisdiensten sprechen:

- LDAP ist als ein offenes, standardisiertes Zugriffsprotokoll oberhalb von TCP konzipiert und so unabhängig vom benutzten Client, Server und Betriebssystem. Für RDBMs existieren zwar vereinheitlichte Schnittstellen wie ODBC (sehr Windows-lastig) oder JDBC (auf Java beschränkt). Diese benötigen aber einen Treibersupport für die RDBM und die Client-Betriebssystemplattform.
- LDAP etabliert sich als offener Verzeichnisdienst. Es werden schon jetzt diverse Applikationen mit LDAP-Support angeboten, von der Betriebssystemauthorisierung (Solaris, Linux) bis zur Konfiguration von Applikationen (Apache, Netscape Communicator).
- LDAP bietet zwar noch keinen standarisierten Replikationsmechanismus, aber die meisten Implementationen beinhalten einen lose gekoppelten Mechanismus, der auch über schmallbandige WAN-Verbindungen stabil funktioniert.

Diese Punkte beziehen sich natürlich nur auf den Einsatz einer Relationalen Datenbank im Verzeichnisdienst-Umfeld. Damit wird nicht der Einsatz in ihrem traditionellen Gebiet in Frage gestellt. Denn dort ist die Integrität der Daten auch bei simultanen Zugriffen und der bekannten Stabilität heutiger Computersysteme oberstes Gebot. Desweiteren fehlen bei LDAP Möglichkeiten, bestimmte Formate, z.B. die Datumsdarstellung, zu überwachen.

2.3. OpenLDAP

Das OpenLDAP-Projekt [3] ist eine Weiterentwicklung des an der Michigan University entwickelten LDAP-Servers. Es beinhaltet aber nicht nur den Standalone-Server, sondern auch eine Clientbibliothek sowie diverse Tools zur Administration eines LDAP-Servers. Wie schon das „Open“ im Namen vermuten läßt, basiert es auf dem Entwicklungsmodell der Open-Source-Vereinigung [2]. Somit liegt der Quelltext

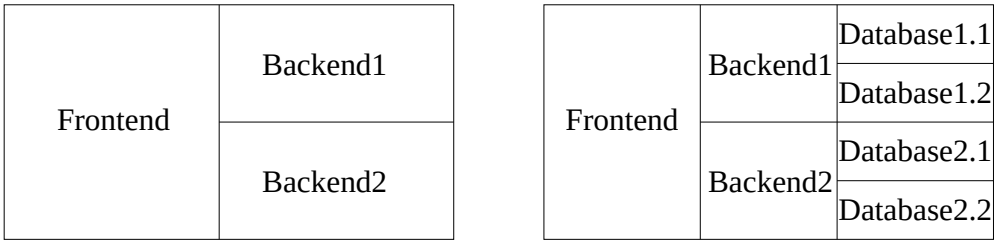

Abbildung 2.3.: Backendschnittstellen

für jeden interessierten Entwickler offen vor. Dadurch wird die Entwicklung von Erweiterungen erheblich vereinfacht, auch kann hierdurch eine Entwicklungsgeschwindigkeit und Marktposition ähnlich dem des Apache Servers [1] im Webserverbereich für die sich noch im Anfangsstadium befindliche Entwicklung von LDAP-Servern erhofft werden. Zur Zeit wird nur LDAPv2 vollständig unterstützt, erste Funktionen von LDAPv3 sind aber schon implementiert. Eine komplette Unterstützung von LDAPv3 ist mit der Version 2.0 zu erwarten. Mit ihr soll auch die Plattformverfügbarkeit auf Windows NT ausgedehnt werden. Derzeit werden alle relevanten UNIX-Plattformen unterstützt.

2.3.1. Backendtechnologie

Als Backend wird der Teil des LDAP-Servers bezeichnet, welcher für das persistente Speichern der Informationen zuständig ist. Im Standardfall sind dies UNIX-typische dbm-Dateien.

Dbm-Dateien enthalten Daten auf der Basis von Schlüssel/Wert- Paaren. Mit Hilfe von speziellen Bibliotheken können diese manipuliert werden.

Da aber die ersten Versionen von LDAP nur als Gateway zu X.500 konzipiert waren und somit keine eigene Speicherung von Informationen notwendig war, entwickelte sich eine saubere Schnittstelle zwischen dem Frontend, welches für die Protokollumsetzung, die Zugriffskontrolle und das Verbindungsmanagement zuständig ist, und den Backends.

Diese Backends können virtuell in den LDAP-Baum „eingehangen“ werden. Leider ist aber diese Schnittstelle gegenüber der Client C-API (RFC 1823 [18]) nie standardisiert worden. Glücklicherweise entstammen aber viele LDAP-Server der Michigan-University-Implementierung, so daß derzeit noch eine große Ähnlichkeit zwischen ihnen besteht. Dies wird aber wohl mit der größeren Verbreitung und der damit einhergehenden Weiterentwicklung in diesem Bereich nicht mehr lange zutreffen. Aber auch hier kann der in dieser Arbeit vorgestellte Entwurf mit seiner abstrahierenden Java-Schnittstelle durch eine spätere Portierung auf andere LDAP-Server Abhilfe schaffen.

Die C-Backend-Schnittstelle bildet dazu traditionell die im LDAP spezifizierten

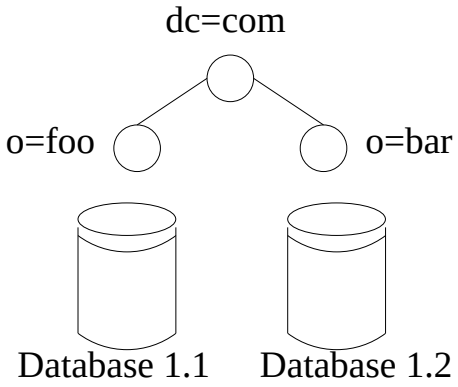

Abbildung 2.4.: Backendbeispiel

Funktionen sowie Erweiterungen zur internen Verwaltung (z.B. Initialiserung und Konfiguration) ab. Das Open-LDAP-Team erkannte aber, daß diese einfache Aufteilung sehr schnell an ihre Grenzen stößt. So entschloß man sich, das klassische Backend in zwei relativ eigenständige Bereiche aufzuteilen - Backend und Database (siehe Abbildung 2.3).

Das Backend ist nun nur noch für die Umsetzung der oben erwähnten Funktionen zuständig, die LDAP-Databases dagegen für die persistente Speicherung oder die weitere Verarbeitung der Daten. Dies bedeutet in der Praxis, daß mit einem Backend mehrere LDAP-Databases betrieben werden können. Ein kleines Beispiel soll die Problematik verdeutlichen.

Ich möchte in einem LDAP-Server zwei Teilbäume aus zwei verschiedenen SQL-Datenbankquellen speisen (siehe Abbildung 2.4). Also teilt sich das Problem in zwei Bereiche auf: die Umsetzung der LDAP-Befehle in SQL Statements und deren Ausführung auf den jeweiligen Datenbankquellen (z.B. verschiedene Tabellen oder Views). Es ist wahrscheinlich nachvollziehbar, daß der Umsetzungsteil zwischen LDAP und SQL unabhängig von den Datenquellen und dem Punkt[1] des Einhängens im LDAP Baum ist. Er benötigt in einfachen Fällen[2] nur andere Konfigurationsdaten für die verschiedenen Quellen, z.B. den Tabellennamen. Somit ist auch die Trennung beider Bereiche konsequent. Im alten Backendinterface war dies nicht möglich.

[1] Dieser Punkt wird auch Suffix genannt.
[2] Die generische Umsetzung zwischen LDAP und SQL ist alles andere als einfach.

3. Java

3.1. Überblick

Über die Programmiersprache Java wurde in letzter Zeit schon viel berichtet und diskutiert, so daß ich hier nicht mehr auf ihre grundlegenden Konzepte und Merkmale eingehen werde. Vielmehr werde ich mich auf den Schwerpunkt des Einsatzes von Java im Serverbereich beschränken.

Wer Javas einzigen Einsatzort im Zusammenhang mit interaktiven Webseiten sieht, verschenkt viel Potential dieser Sprache. Viele in diesem Einsatzgebiet entstandenen Vorurteile gegenüber Java, wie z.B. träge grafische Oberfläche oder Inkompatibiliäten der JVMs der verschiedenen Webbrowser, sind im Servereinsatz irrelevant. Hier besticht Java durch die plattformunabhängige Unterstützung vieler Schnittstellen (z.B. auf Datenbanken mittels Java Database Connection – JDBC [36]), eingebauter Netzwerkfähigkeit, von einer vom darunterliegenden Betriebssystem abstrahierenden Thread-Implementierung, der einfachen Entwicklung von verteilten Systemen (Remote Method Invocation – RMI [37], Common Object Request Broker Architecture – Corba [38]) und der Unterstützung der Komponententechnologie auch im Enterprise-Bereich (Enterprise Java Beans – EJB [41]). Dieser Trend wird auch durch die große Anzahl von heute schon verfügbaren Applikationsservern für Java belegt. Desweiteren sind viele namhafte Projekte in der Entwicklung, so z.B. das „San Francisco“ Framework der Firma IBM, welches das Erstellen von Business-Anwendungen mit Java stark vereinfachen soll. Aber besonders im Bereich neuer offener Technologien, wie z.B. der Beschreibungssprache XML [40], etabliert sich Java als Referenzplattform, welche mit dem „Java Naming and Directory Service Interface“(JNDI) [42] auch eine definierte Schnittstelle zu verschiedenen Verzeichnisdiensten anbietet.

3.2. Java Native Interface

Da in dieser Arbeit mit zwei Programmiersprachen entwickelt wird, muß eine Schnittstelle zwischen beiden verwendet werden – diese wird durch das Java Native Interface (JNI) [34, 35] vom Java Developer Kit (JDK) bereitgestellt. Das JNI versucht eine Standardisierung der Schnittstelle zwischen der Java Virtual Machine (JVM) und der Software, welche in anderen Sprachen geschrieben ist, durchzusetzen. Im gegenwärtigen Zustand sind nur Schnittstellen zu den Sprachen C und C++ definiert.

Das größte Augenmerk des JNI liegt auf der Portabilität zwischen verschiedenen JDKs und Plattformen. Zu Zeiten des JDK 1.0 waren die Aufrufe, welche eine C Bibliothek implementieren mußte, nur unklar spezifiziert. Ein Beispiel: Die Sun-VM suchte nach der nativen Methode `Klasse.eineMethode()` im Package `einPackage` unter dem C-Namen `einPackage_Klasse_eineMethode()`. Ein anderer JVM-Hersteller war aber der Ansicht, z.B. den Packagenamen am Ende des C-Namens zu erwarten. So konnte es nötig sein, für verschiedene JVMs unterschiedliche Bibliotheken zu entwickeln. Um diesen offenkundigen Problemen aus dem Weg zu gehen, spezifiziert JNI auch solche Details, darunter auch folgende:

- Abbildung der Java-Namen auf C-Bezeichner und umgekehrt. So erlaubt Java z.B. die Verwendung von beliebigen Unicode-Zeichen, C dagegen nur ASCII-Zeichen.
- Eine Schnittstelle zum Garbage Collector, denn die dynamische Speicherverwaltung von Java ist nicht kompatibel mit der statischen von C.
- Datentypen für die C-Programmierung. In Java z.B. sind Werte vom Typ Integer 64 Bit lang, dagegen in C nicht standardisiert.
- String-Umwandlungen sind nötig, weil Java auch hier mit Unicode arbeitet.
- Übergabe der Argumente von Java nach C.
- Exception Handling ist im Gegensatz zu Java in C nicht bekannt.
- Zusammenspiel von Native- und Java-Threads.
- Benutzen einer Embedded JVM in C (Invocation API).
- Einsatz von C-Bibliotheken mit verschiedenen JDK Versionen.

4. Entwurf eines Java-Backends

4.1. Zielstellung

Die Zielstellung dieser Arbeit ist es, eine einfache, aber flexible Möglichkeit zur Implementierung von Verknüpfungen zwischen dem Verzeichnisdienst LDAP und anderen Datenquellen anzubieten. Ein LDAP-Server soll also die Rolle eines Bindeglieds zwischen unterschiedlichen Datenquellen einerseits und einem netztransparenten Zugriff andererseits spielen.

4.2. Problemanalyse

4.2.1. Client/Server

Da es sich im Falle von LDAP um ein Client/Server-Protokoll handelt, gibt es zwei Seiten, an denen ein Eingriff zur Erreichung der Zielstellung möglich wäre – entweder

am Client, oder aber am Server. Eine Manipulation am Client ist jedoch schwer durchführbar: Erstens gibt es weitaus mehr LDAP-Clients als -Server, und zweitens hat man selten die Kontrolle über alle Clients, welche auf einen Server zugreifen. Demgegenüber bietet die Server-Seite einen idealen, weil transparenten und zentralen, Punkt zum Eingriff.

4.2.2. Einbinden von Datenquellen in LDAP-Server

Es gibt verschiedene Möglichkeiten, externe Datenquellen in einen LDAP-Server einzubinden. Alle haben ihre Vor- und Nachteile, welche im jeweils vorliegenden Einsatzszenario abzuwägen sind.

Vollständige Migration nach LDAP

Dies ist wohl das einfachste Szenario – es geht darum, einen bestehenden Datenbestand nach LDAP zu migrieren und dann auch über LDAP zu administrieren. In diesem Fall müssen folgende drei Schritte durchgeführt werden:

1. Auslesen der Datenquelle.
2. Transformieren der Daten in das LDAP Data Interchange Format (LDIF).
3. Importieren in den LDAP-Server.

Das LDIF [7] repräsentiert LDAP-Einträge in Text-Form und ist deshalb mit einfachen Mitteln zu erstellen. Als Beispiel sei hier ein kleines Perl-Skript angeführt, welches aus der Datei `/etc/passwd` eine Datei im LDIF-Format erzeugt:

passwd2ldif.pl

```
1 while (@p=getpwent){
2     print "dn: uid=$p[0], o=FHTW, c=DE\n";
3     print "uid: $p[0]\n";
4     print "password: {crypt}$p[1]\n";
5     print "cn: " . (split ",", $p[6])[0] . "\n";
6     print "\n";
7 }
```

Die so erzeugte LDIF-Datei kann nun mit Hilfsprogrammen - wie z.B. `ldif2ldbm` [9] beim OpenLDAP-Projekt - in den LDAP-Server importiert werden.

Koexistenz von LDAP-Server und externen Datenquellen

Ist es nicht möglich, den Datenbestand vollständig nach LDAP zu migrieren und auch über LDAP zu administrieren, weil z.B. vorhandene Frontends zur Datenbestandsaufnahme bzw. -pflege nicht verworfen werden sollen oder die im Kapitel 2.2.3 beschriebenen Vorteile eines relationalen Datenbanksystems benötigt werden, ist je nach Einsatzgebiet unterschiedlich komplex vorzugehen.

Werden die Daten nur in der externen Datenquelle verändert (erfolgen also keine Änderungsoperationen über die LDAP-Schnittstelle), so ist es möglich, ähnlich wie im vorhergehenden Kapitel beschrieben, die Datenquelle in bestimmten Zeitabständen zu exportieren und in den LDAP-Server einzuspielen. Dies ist natürlich bei größeren Datenbeständen nur bei der Verwendung von großen Zeitabständen zwischen den Updates sinnvoll. Dieses Verfahren kann auch durch die Benutzung von „Zuletzt geändert am“ -Zeitstempeln differenzierter erfolgen. Dies setzt aber eine vorhandene Infrastruktur mit solchen Möglichkeiten vorraus.

Sollen schließlich auch Änderungen über die LDAP-Schnittstelle erlaubt werden, ist ein komplexer Synchronisations-Mechanismus zwischen LDAP-Datenbestand und der externen Datenquelle nötig. Dies kann z.B. durch die Verwendung von Zeitstempeln auf beiden Seiten (LDAP-Server und externe Datenqelle) vereinfacht werden, ist aber unter Berücksichtigung von Ausfall-Möglichkeiten auf einer der beiden Seiten nicht immer einfach zu implementieren.

Einen weiteren Eingriffspunkt bietet der Replikationsmechanismus von LDAP-Servern. Leider ist dieser noch nicht standarisiert, es wurden aber erste Vorschläge publiziert [25]. So ist im Netscape Directory Server der Replikationsmechanismus ein interner Dienst. Demgegenüber realisiert das OpenLDAP-Projekt ihn als externen Server – Standalone LDAP Update Replication Daemon [8] (slurpd). Dieser liest eine vom LDAP-Server erstellte Replikations-Logdatei mit allen Änderungen ein und gleicht danach die weiteren LDAP-Server ab. Da das Format der Logdatei textbasierend und dokumentiert [6] ist, könnte man einen eigenen slurpd entwickeln und über ihn die externe Datenquelle bei Änderungen im LDAP-Datenbestand synchronisieren. Aber dies wäre, wie erwähnt, OpenLDAP-spezifisch und wird auch mit dem Erscheinen eines Standards für die Replikation von LDAP-Servern durch andere Mechanismen ersetzt werden.

Die letzte und in dieser Arbeit verwendete Technologie basiert auf der schon im Kapitel 2.3.1 beschriebenen Backendschnittstelle des OpenLDAP- Servers. Sie ermöglicht es, in Echtzeit Änderungen der externen Datenquelle im LDAP abzubilden oder über LDAP vorzunehmen.

4.2.3. Zugriffssyntax - Umsetzung

LDAP bietet eine standardisierte Zugriffssyntax auf Datenquellen. Da verschiedene Datenquellen sich bezüglich ihrer Zugriffssyntax aber unterscheiden können (siehe Tabelle 4.1), muß für jede eine spezielle Umsezung erfolgen, welche selten mit nur

Datenquelle	**Datenbank**	**Textdatei**	**LDAP Server**
Sprache	SQL	Reguläre Ausdrücke	LDAP Filter
Syntax	SELECT uid, name FROM table WHERE name LIKE "homer%"	/(^[^:]):(homer.*)$/	(name=homer*) uid, name

Tabelle 4.1.: Beispiele von verschiedenen Zugriffssyntaxen auf Datenquellen

einem einfachen Satz von Regeln auskommt. Um die daraus resultierende Komplexität programmtechnisch umzusetzen, ist eine Programmiersprache erforderlich, die folgenden Ansprüchen genügt:

- Unterstützung von **Standardisierten Schnittstellen** zum Zugriff auf möglichst viele unterschiedliche Datenquellen.
- **Plattformunabhängigkeit** vereinfacht den Einsatz im heterogenen Umfeld, für welches LDAP konzipiert ist.

Java bietet idealerweise all diese Eigenschaften und erleichtert durch seinen objektorientierten Ansatz das spätere Weiterentwickeln.

4.2.4. Serverfunktionen

Da die Entwicklung serverseitig stattfinden soll, sind dafür notwendige Funktionen bereitzustellen, welche nichts mit dem zentralen Punkt der Umsetzung zwischen LDAP und den Zugriffssyntaxen auf andere Datenquellen zu tun haben. Dies sind z.B.:

Verbindungsmanagement zum Verarbeiten und Handhaben mehrer Anfragen gleichzeitg.

Protokoll-Dekodierung ist notwendig, weil Java das verwendete Kodierungsverfahren ASN.1 nicht direkt unterstützt.

Replikation ist ein integraler Bestandteil von Verzeichnisdiensten wie LDAP.

Hierfür sollten sinnvollerweise schon vorhandene Infrastrukturen in Form von LDAP-Servern verwendet werden. So bietet der OpenLDAP-Server eine Schnittstelle (siehe Kapitel 2.3.1) zwischen den allgemeinen LDAP-Server- Funktionen, auch Frontend genannt, und den Datenquellen – den Backends.

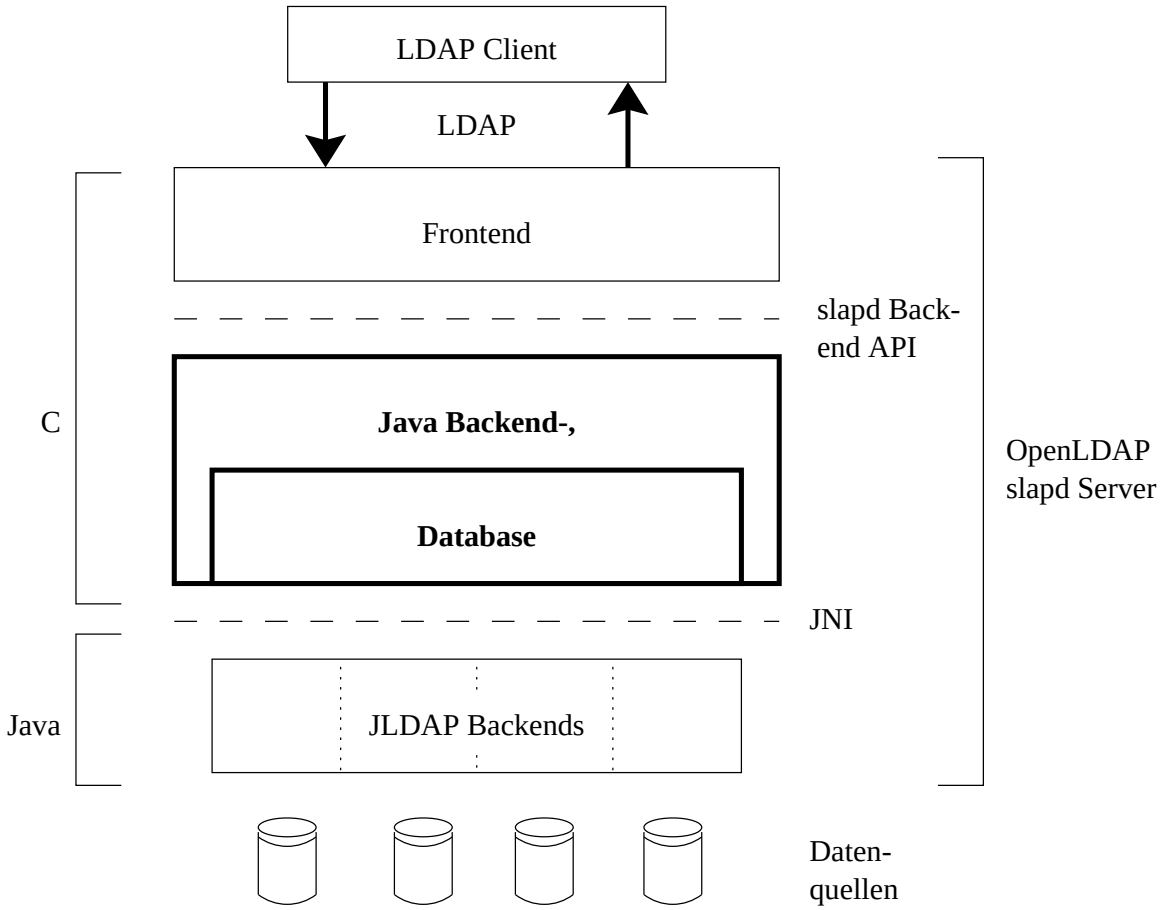

Abbildung 4.1.: Schichtenmodell zur Problemanalyse

4.2.5. Schichtenmodell

In der Abbildung 4.1 sind alle bis zu diesem Zeitpunkt bekannten Komponenten für diese Arbeit in einem Schichtenmodell zusammengefasst. Diese Darstellung wird in der konzeptionellen Phase weitergeführt und ausgebaut.

Auf der linken Seite der Abbildung sind die beiden in diesem Projekt verwendeten Sprachen - C und Java - den Implementierungsschichten zugeordnet. Bindeglied zwischen beiden ist das mit dem JDK mitgelieferte JNI (siehe Kapitel 3.2). Hierdurch beschränkt sich das sonst sehr komplexe Thema „Verbinden zweier Programmiersprachen miteinander“ auf das Studieren und Anwenden dieser Schnittstelle.

Desweiteren sind in der Abbildung 4.1 die benötigten Module mit den verbindenden Schnittstellen hierarchisch dargestellt. Module, welche im Verlauf dieser Arbeit konzipiert und implementiert werden, habe ich hervorgehoben dargestellt. Das Frontend-Modul ist Teil des „slapd“ -Standalone-LDAP- Servers aus dem OpenLDAP-Projekt. Es stellt die weiter oben besprochene Infrastruktur mit ihren Grundfunktionalitäten eines LDAP-Servers bereit. Dazu werden alle LDAP-Protokolloperationen

(siehe Kapitel 2.2.2) und -Erweiterungen (z.B. zur Konfiguration und Initalisierung) auf der „slapd Backend API“ abgebildet. An dieser Stelle setzt nun das zu entwickelnde Java-Backend an. Dieses Backend ist für die Transformation der über die strukturierte Schnitstelle eingehenden C-Funktionsaufrufe in objektorientierte Java-Methodenaufrufe zuständig. Die Implementierung dieser aufzurufenden Java-Methoden erfolgt in den „JLdap“-Backends, welche für die verschiedenen Datenquellen entwickelt werden können.

4.3. Anforderungen

Mit Hilfe des Java-Backends soll es ermöglicht werden, weitaus schneller und einfacher als in der Programmiersprache C ein „Backend“ für den OpenLDAP Server zu entwickeln. Dabei wird auf die Bereitstellung einer klaren, objektorientierten Schnittstelle viel Wert gelegt. Hierfür werden alle Kernfunktionalitäten von LDAPv2 sowie die für die Verwaltung des Backends benötigten Erweiterungen auf dieser Schnittstelle abgebildet. Als primäre Zielplattform ist Solaris 7 (Sparc-Architektur) mit dem Sun JDK 1.2 vorgesehen. Während der Entwicklung wird aber schon ein großes Augenmerk auf die Portierbarkeit des Projektes auf weitere Plattformen gelegt, stark gekoppelt mit der dortigen Verfügbarkeit des JDK 1.2 und des OpenLDAP-Servers.

Für die Verwaltung des Backends werden Möglichkeiten zur Initialisierung, Konfiguration und zum Beenden implementiert. Damit soll es Entwicklern möglich sein, Objekte zu entwickeln, welche über die komplette Laufzeit des OpenLDAP-Servers zur Verfügung stehen. Ein Anwendungsbeispiel dafür ist eine permanente Datenbankverbindung. Desweiteren soll die Konfiguration der Backends transparent über die Standardkonfigurationsdatei des OpenLDAP-Servers ermöglicht werden.

Die Funktionalität von LDAPv2 wird von mir mit folgenden Einschränkungen vollständig bereitgestellt:

bind Es wird nur die „simpleauthorisation“ mit einem Passwort unterstützt; die Kerberos-Erweiterung wird für die im Rahmen dieser Diplomarbeit angestrebten Einsatzgebiete nicht benötigt.

abandon Diese Funktion zum Abbrechen von Suchvorgängen erhöht den Implementierungsaufwand erheblich, obwohl nur wenige Clients sie unterstützen. Sie wird deswegen nicht weiter berücksichtigt.

4.4. Konzeption

In den folgenden Abschnitten werde ich zuerst die OpenLDAP-slapd-Backend-API und die von mir daraus konzipierte objektorientierte JLdap-Schnittstelle vorstellen sowie die hierfür benötigte Umsetzung und das Laufzeitverhalten erläutern.

4.4.1. Die OpenLDAP slapd Backend API

Die OpenLDAP-slapd-Backend-API enthält 19 Aufrufe. Neun dieser Aufrufe korrespondieren mit den neun LDAP-Protokolloperationen: bind, unbind, search, compare, modify, modify RDN, add, delete und abandon. Die anderen zehn Aufrufe dienen dem Initialisieren, Beenden und Konfigurieren des Backends sowie der LDAP-Databases. Den ersten neun Routinen werden immer die gleichen ersten drei Parameter übergeben:

BackendDB *bd: Informationen über die Backend Database.

Connection *c: Informationen über die Connection.

Operation *o: Informationen über die LDAP-Operation.

Die Strukturen `Backend DB *bd` und `Operation *o` können durch die Zeiger `void *be_private` bzw. `void *o_private` mit eigenen Strukturen erweitert werden. Dadurch wird es ermöglicht, spezielle Informationen in diesem Backend ohne die Verwendung von globalen Variablen allen Routinen bereitzustellen. Die restlichen Parameter sind von den entsprechenden Routinen abhängig.

Bind

```
java_back_op_bind(
      BackendDB *bd,
      Connection *conn,
      Operation *op,
      char *dn,
      int method,
      struct berval *cred
)
```

`dn` Der Distinguished Name (DN), wie er zur Autorisierung verwendet wird.

`method` Die verwendete Autorisierungsmethode. Dies kann eine von drei in `ldap.h` deklarierten Konstanten sein:

- `LDAP_AUTH_SIMPLE` Plain Passwort
- `LDAP_AUTH_KRBV41` Kerberos Version 4.1
- `LDAP_AUTH_KRBV42` Kerberos Version 4.2

`cred` Das Passwort.

Die `bind`-Routine muß den Wert 0 zurückgeben, wenn die Autorisierung erfolgreich war. Weiterhin ist zu beachten, daß eine anonyme Autorisierung vom Frontend abgefangen wird, so daß in diesem Fall die `bind`-Routine nicht aufgerufen wird.

Unbind

```
java_back_op_unbind(
      BackendDB *bd,
      Connection *conn,
      Operation *op
)
```

Die Verbindung wird vom Frontend geschlossen. Mit Hilfe der `unbind` -Routine kann das Backend lokale Informationen entfernen, welche mit dieser Verbindung verknüpft sind.

Compare

```
java_back_op_compare(
      BackendDB *bd,
      Connection *conn,
      Operation *op,
      char *dn,
      Ava *ava
)
```

`dn` Der Distinguished Name des zu vergleichenden Eintrages.

`ava` Das Attribute Type- / Wert-Tupel, mit welchem der Eintrag verglichen werden soll.

Die `ava` Struktur ist folgendermaßen definiert:

```
typedef struct ava {
      char *ava_type;
      struct berval ava_value;
} Ava;
```

Der Typ für den Vergleich ist in `ava_type` (z.B. „maildrop“) enthalten und der Wert in `ava_value` (z.B. „foo@bar.com“).

Search

```
java_back_op_search(
```

```
      BackendDB *bd,
      Connection *conn,
      Operation *op,
      char *base,
      int scope,
      int sizelimit,
      int timelimit,
      Filter *filter,
      char *filterstr,
      char **attrs,
      int attrsonly
)
```

`base`	Der DN des Basis-Objektes, von dem aus die Suche beginnen soll.
`scope`	Der Suchbereich. Eine der Konstanten aus `ldap.h`: • `LDAP_SCOPE_BASEOBJECT` • `LDAP_SCOPE_ONELEVEL` • `LDAP_SCOPE_SUBTREE`
`sizelimit`	Ein vom Client übermitteltes Limit für die Höchstzahl der Einträge, die zurückgegeben werden sollen. Null bedeutet: „kein Limit“.
`timelimit`	Ein vom Client übermitteltes Zeitlimit für den Suchvorgang, siehe auch `sizelimit`.
`filter`	Eine Datenstruktur, die den Suchfilter repräsentiert.
`filterstr`	Der Suchfilter als String. Das verwendete Format ist im RFC 1588 beschrieben. Ein Backend verwendet im allgemeinen nur einen der beiden `filter`- Parameter.
`attrs`	In diesem Array von `char *` werden die Attribute für die Rückgabe des Suchergebnisses übergeben. `NULL` bedeutet: alle Attribute sollen zurückgegeben werden.
`attrsonly`	Dieser Boolean-Parameter zeigt an, ob nur die Typen der Attribute (`TRUE`) oder auch die Werte (`FALSE`) zurückgegeben werden sollen.

Add

```
java_back_op_add(
      BackendDB *bd,
```

```
      Connection *conn,
      Operation *op,
      Entry *e
)
```

`e` Ein Zeiger auf die Entry-Struktur, welche den hinzuzufügenden Eintrag beschreibt.

Diese Entry-Struktur ist in `slap.h` definiert:

```
typedef struct entry {
      char *e_dn;
      Attribute *e_attrs;
} Entry;
```

Das Feld `e_dn` enthält den DN des einzutragenden Objektes. Im Feld `e_attrs` befindet sich eine verkettete Liste der Attribute dieses Eintrages:

```
typedef struct attr {
      char *a_type;
      struct berval **a_vals;
      int a_syntax;
      struct attr *a_next;
} Attribute;
```

Durch das Feld `a_syntax` wird die Syntax des Attributes bestimmt. Es kann einen von fünf in der Datei `slap.h` spezifizierten Werten besitzen:

```
SYNTAX_CIS /* case insensitive string */
SYNTAX_CES /* case sensitive string */
SYNTAX_BIN /* binary data */
SYNTAX_TEL /* telephone number string */
SYNTAX_DN  /* dn string */
```

Diese Syntax-Werte können auch miteinander verbunden werden. So hat der Typ `telephoneNumber` die Syntax `(SYNTAX_CIS | SYNTAX_TEL)`. Zum Vergleich der Beispieleintrag eines Nutzers:

```
dn: uid=4711, o=FHTW Berlin, c=DE
uid=4711
mail=m4711@fhtw-berlin.de
mail=Musterman@mail.de
name=Musterman
```

Delete

```
java_back_op_delete(
      BackendDB *bd,
      Connection *conn,
      Operation *op,
      char *dn,
)
```

`dn` Der Distinguished Name des zu löschenden Eintrages.

Modify

```
java_back_op_modify(
      BackendDB *bd,
      Connection *conn,
      Operation *op,
      char *dn,
      LDAPMod *mods
)
```

`dn` Der Distinguished Name des zu verändernden Eintrages.
`mods` Die Liste der Modifikationen für den Eintrag.

Die Struktur `LDAPMod` ist folgendermaßen definiert:

```
typedef struct ldapmod {
      int mod_op;
      char *mod_type;
      union {
            char **modv_strvals;
            struct berval **modv_bvals;
      } mod_vals;
#define mod_values mod_vals.modv_strvals
#define mod_bvalues mod_vals.modv_bvals
      struct ldapmod *mod_next;
} LDAPMod;
```

Das Feld `mod_op` identifiziert den Typ der Änderung und kann einen der folgenden, in der Datei `ldap.h` definierten Werte haben:

- `LDAP_MOD_ADD`

- LDAP_MOD_DELETE
- LDAP_MOD_REPLACE

Der Eintrag mod_type enthält den attribute type und mod_vals die Attributwerte.

Modify RDN

```
java_back_op_modifyrdn(
      BackendDB *bd,
      Connection *conn,
      Operation *op,
      char *dn,
  char *newrdn,
  int deleteoldrdn
)
```

dn	Der Distinguished Name des zu verändernden Eintrages.
newrdn	Der neue Relative Distinguished Name des zu verändernden Eintrages.
deleteoldrdn	Soll der alte DN erhalten bleiben (beim Kopieren), steht hier eine Null, enthält er einen anderen Wert, so soll der alte DN gelöscht (verschoben) werden.

Senden von Sucheinträgen

Die Funktion send_search_entry dient zum Kodieren und Senden der gefundenen Einträge zu einer Suchanfrage an den LDAP Client. Sie ist folgendermaßen deklariert:

```
send_search_entry(
                  BackendDB *bd,
                  Connection *conn,
                  Operation *op,
                  Entry *e,
                  char **attrs,
                  int attrsonly
)
```

Zur Beschreibung der Parameter siehe auch search auf Seite 24.

Senden des Ergebnisses

Ein LDAP-Result wird zum Client geschickt, wenn die Routine send_ldap_result aufgerufen wird.

```
send_ldap_result(
                  Connection *conn,
                  Operation *op,
                  int err,
                  char *matched,
                  char *text
)
```

Der Parameter `err` stellt den LDAP-Fehlerkode dar. Diese Kodes werden in der Includedatei `ldap.h` definiert. Der Parameter `matched` ist nur dann nicht `NULL`, wenn als Fehlerkode `LDAP_NO_SUCH_OBJECT` angegeben wird. In diesem Fall wird in `matched` der Teil des DN zurückgegeben, der erfolgreich aufgelöst werden konnte. Der letzte Parameter, `text`, kann eine beliebige für den Client bestimmte Message enthalten. Diese sollte aber einer sinnvollen Meldung entsprechen (z.B. Fehlerbeschreibung).

4.4.2. Die objektorientierte JLdap-Schnittstelle

In der folgenden Darstellung 4.2 habe ich das in der Problemanalyse (siehe Kapitel 4.2) entwickelte Schichtenmodell um zwei neue Module erweitert, die Klassen Connection und BackendDB. Beide Klassen spielen die zentrale Rolle bei JLdap – der Abbildung der OpenLDAP-slapd-Backend-API auf die objektorientierte Java Schnittstelle.

Die Klasse Connection

Die Verbindung zwischen Client und Server hat eine zentrale Bedeutung: Jede LDAP-Anfrage beginnt mit dem Aufbau einer solchen Verbindung, und bis zu ihrer Beendigung können mehrere LDAP-Operationen aufgerufen werden (z.B. bind, search, add, unbind). An diese Verfahrensweise lehnt sich auch die objektorientierte Java-Schnittstelle an, d.h., mit jeder neuen Verbindung eines Clients wird eine Instanz der Klasse Connection erzeugt. Über Methoden dieses Objektes werden dann alle LDAP-Operationen während einer Verbindungsdauer abgearbeitet. Die Aufgabe des späteren Entwicklers eines JLdap-Backends wird es sein, eigene Klassen von Connection abzuleiten.

Die Methoden und deren Parameter in der Connection-Klasse halten sich an die C-Funktionen aus der Backend-API des OpenLDAP-Servers slapd, erweitert um den Gesichtspunkt der Objektorientierung. Deswegen habe ich beide Deklarationen in der Tabelle 4.2 zum Vergleich gegenübergestellt.

Es ist bei allen Java-Funktionen auffallend, daß die im Kapitel 4.4.1 beschriebenen ersten drei Parameter der korrespondierenden C-Funktionen nicht übergeben werden. Im objektorientierten Ansatz entfällt z.B. derzeit die `struct Operation`. Sie enthält keine für das Schreiben eines JLdap-Backends wichtigen Daten. Die `struct Connection` hingegen wird als das Connection-Objekt abgebildet. Alle wichtigen

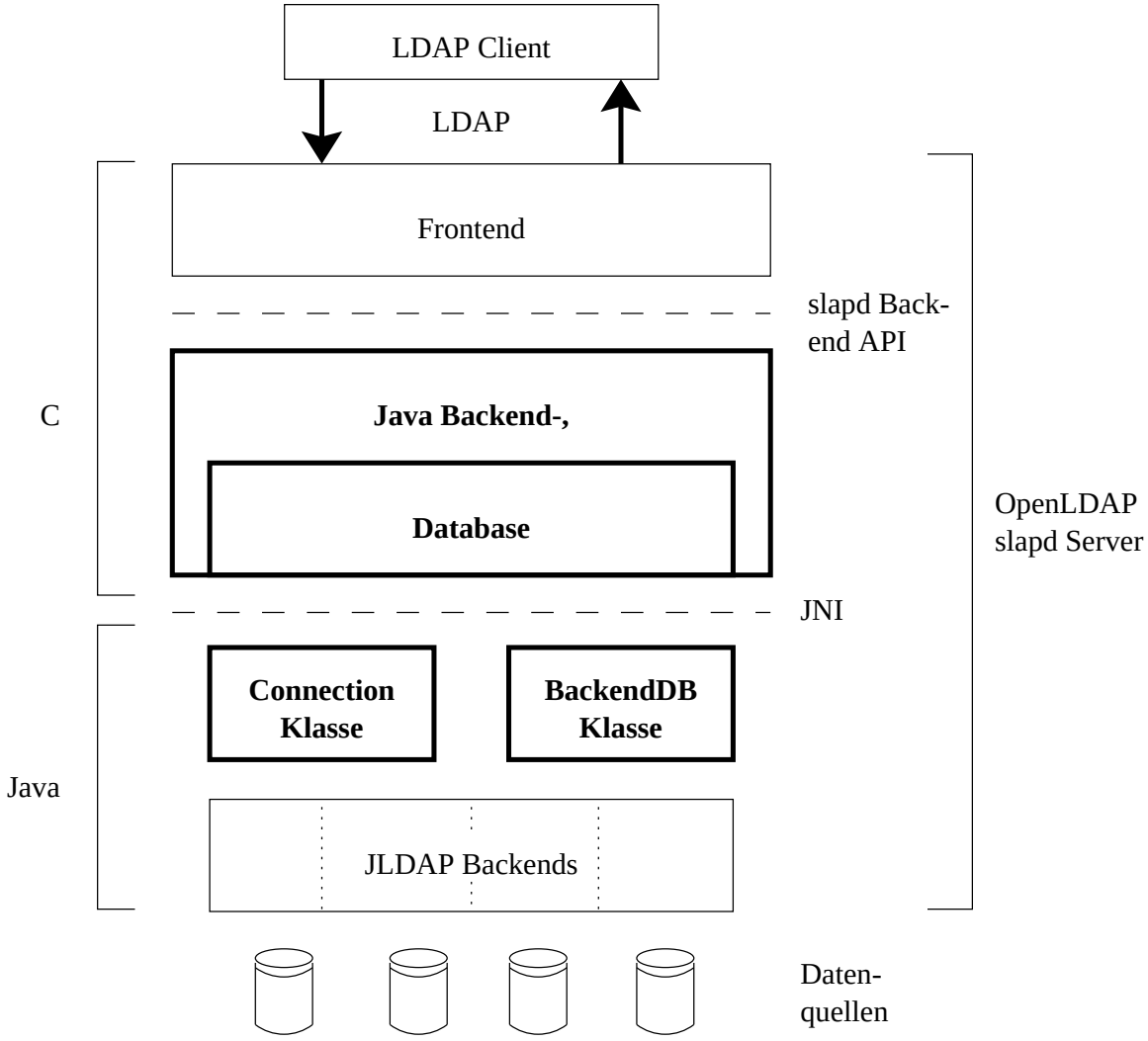

Abbildung 4.2.: Die objektorientierte JLdap-Schnittstelle

LDAP Op.	**C Funktion**	**Java Methode**
bind	int java_bind(BackendDB *bd, Connection *conn, Operation *op, char *dn, int method, struct berval *cred)	boolean bind(String dn, String passwd)
unbind	int java_unbind(BackendDB *bd, Connection *conn, Operation *op)	void unbind()
compare	int java_compare(BackendDB *bd, Connection *conn, Operation *op, char *dn, Ava *ava)	boolean compare(String dn, Attribute attr)
search	int java_search(BackendDB *bd, Connection *conn, Operation *op, char *base, int scope, int sizelimit, int timelimit, Filter *filter, char *filterstr, char **attrs, int attrsonly)	SearchResults search(String base, int Scope, int sizelimit, int timelimit, String filterstr)
add	int java_add(BackendDB *bd, Connection *conn, Operation *op, Entry *e)	void add(Entry entry)
delete	int java_delete (BackendDB *bd, Connection *conn, Operation *op, char *dn)	delete(String dn)
modify	int java_modify (BackendDB *bd, Connection *conn, Operation *op, char *dn, LDAPModList *modlist)	modify(String dn, ModificationSet mset)
modrdn	int java_modrdn (BackendDB *bd, Connection *conn, Operation *op, char *dn, char*newrdn, int deleteoldrdn, char *newSuperior)	modrdn(String dn, String newrdn, boolean deleteoldrdn)

Tabelle 4.2.: Gegenüberstellung: C Funktionen - Java Methoden

Informationen darüber (derzeit nur die eindeutige ConnectionID) können als Membervariablen angesehen werden. Informationen, welche in `BackendDB` zu finden sind, stehen in einer „hat eine“-Beziehung zur Klasse `Connection` und werden somit als eine eigenständige Klasse `BackendDB` abgebildet, die ihrerseits über eine Membervariablen mit der Klasse `Connection` verbunden wird. Die weiteren C-Strukturen als Übergabeparameter werden in folgende Java-Klassen umgewandelt: `Entry`, `SearchResults`, `Attribute` und `Modification`.

Weiterhin ist zu erkennen, daß die Rückgabeparameter der C-Funktionen nicht mit denen der Java-Methoden übereinstimmen. Die Ursache dafür ist, daß der OpenLDAP-Server die Rückgabewerte der Backend-Funktionen (mit Ausnahme von `bind` siehe Seite 23) nicht beachtet. Die Funktionen sind somit selbst dafür verantwortlich, wann und wie sie mit dem LDAP-Client kommunizieren. Hierfür stehen aber zwei Hilfsfunktionen zur Verfügung (siehe Seite 28). Der objektorientierte Ansatz bietet im Gegensatz dazu einen eleganteren Weg: Es wird mit Rückgabeparametern gearbeitet, wann immer dies mit der LDAP-Operation im Einklang steht. So arbeitet `compare` mit einem Rückgabewert vom Typ Boolean. Dagegen benötigt z.B. die `add`-Methode keinen expliziten Rückgabewert. Hier wird davon ausgegangen, daß bei einem „normalen“ Beenden der Methode ein `LDAP_OPERATION_SUCCESSFUL` zu senden ist, denn zum Signalisieren eines Fehlers werden javatypische Exceptions verwendet, in diesem speziellen Fall mit Hilfe der bereitgestellten Klasse `LDAPException`.

Die Klasse BackendDB

Die Lebensdauer der Klasse `BackendDB` erstreckt sich über die komplette Laufzeit des JLdap-Backends, sie wird beim Start des slapd für jedes JLdap-Backend instantiiert und kann somit persistente Operationen für alle eingehenden Verbindungen bereitstellen. Ein Beispiel hierfür wäre eine Datenbankverbindung. Außerdem erfolgt über diese Klasse auch die Konfiguration des JLdap-Backends, hierzu werden alle in der slapd-Konfigurationdatei für dieses JLdap-Backend gefundenen Einträge der Methode `config` übergeben. Über die Methode `close` wird das Schließen des JLdap-Backends signalisiert. Hier kann dann z.B. die Datenbankverbindung wieder abgebaut werden. Auch in diesem Fall ist es die Aufgabe des Entwicklers eines JLdap-Backends, eine eigene Klasse von `BackendDB` abzuleiten und nach seinen Anforderungen zu implementieren.

Die Klasse LDAPException

Die Klasse `LDAPException` ist stark an die Funktion `send_ldap_result` in der slapd-Backend-API angelehnt (siehe Seite 28). Denn der slapd unterscheidet nicht zwischen dem Senden eines Fehlers und z.B. der Bestätigung eines Vergleiches - beide werden nur durch verschiedene Fehlerkodes dargestellt. Wie in Java notwendig, ist die Klasse `LDAPException` von der allgemeinen Klasse `Exception` abgeleitet. Sie erweitert

diese aber um das Setzen und Auslesen der schon in der Funktion `send_ldap_result` besprochenen Einträge: `err`, `matchedDN` und `text`.

4.4.3. Die Java Shadow-Klasse

In den vorhergehenden Kapiteln ist der Unterschied zwischen den beiden Schnittstellen „slapd Backend API“ und glqq Java JLdap“ verdeutlicht worden. Um diese Transformation zu vereinfachen, wurde ein weiteres Modul, die Java Shadow-Klasse (siehe Abbildung 4.3) eingefügt. Dadurch kann die slapd-Backend-API in einem ersten Schritt auf eine ihr sehr ähnliche Java-Schnittstelle abgebildet werden und danach in Java die weitere Umwandlung in eine objektorientierte Schnittstelle vorgenommen werden. Hierdurch wird der Anteil des C-Kodes in diesem Projekt auf ein Minimum reduziert. Die vereinfachte Implementierung und erhöhte Portabilität sollten die entstehenden Laufzeiteneinbußen rechtfertigen. Weitere Gründe sind:

- Das Arbeiten mit Rückgabewerten in C, die nicht primitiven Java-Typen entsprechen, ist komplex.
- C kennt keine standardisierten Ausnahmen, so daß die Behandlung erschwert wird.
- Die Umsetzung der strukturierten API in eine objektorientierte Schnittstelle erfordert Strukturen, wie z.B. Hashes. Diese sind in Java weitaus einfacher zu implementieren als in C.

Diese Shadow-Klasse wird für jedes JLdap-Backend einmal instantiiert und ist dann für folgende Aufgaben verantwortlich:

- Erzeugung je einer BackendDB-Objektinstanz für jedes JLdap-Backend.
- Erkennung neuer Verbindungen und instantiieren eines neuen Connection-Objektes. Dies beinhaltet auch das Löschen eines Objektes nach dem Schließen einer Verbindung.
- Behandlung der Zuordnung zwischen der eindeutigen Connection-ID und dem dazugehörigen Connection-Objekt.
- Umwandlung der Java-Rückgabewerte und `LDAPException`s in für C angepaßte Datenstrukturen.
- Umwandlung der einfachen C-Übergabeparameter in spezielle Java-Objekte, z.B. wird aus dem Integer-Parameter `deleteoldrdn` in C ein Boolean-Wert in Java.

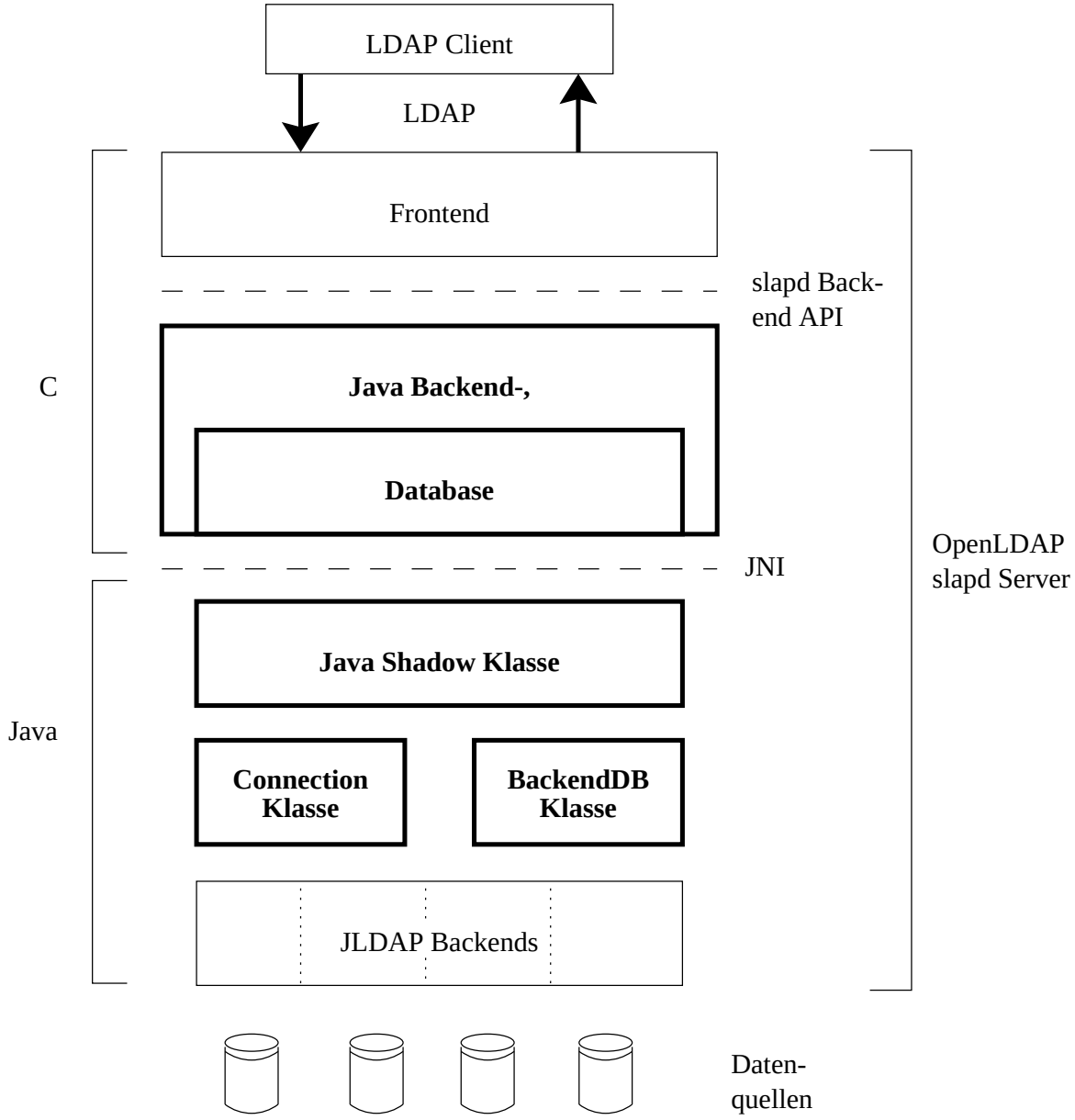

Abbildung 4.3.: Die Java Shadow-Klasse

4.4.4. Das dynamische Modell

Die Initialisierungsphase eines JLdap Backends

Ich werde hier mit Hilfe eines Ablaufdiagrammes (siehe Abbildung 4.4) und einem Ausschnitt der OpenLDAP-slapd-Konfigurationsdatei die Initialisierungphase eines JLdap-Backends genauer erläutern.

```
#Backendteil                                          (1)
Backend          Java                                 (2)
classpath        /path/to/be/classes.jar              (3)
                                                      (4)
#Databaseteil                                         (5)
Database         Java  Foo                           (6)
Suffix           "ou=foo, o=bar, c=org"               (7)
connection_class "org.bar.MyConnection"               (8)
backendDB_class  "org.bar.MyBackendDB"                (9)
dburl            "jdbc://mysql.com"                  (10)
```

Die Konfigurationsdatei ist nach dem Prinzip Schlüsselwort - Parameter aufgebaut. Mit einem `#` beginnende Zeilen sind Kommentare und werden nicht berücksichtigt. Unterstrichene Schlüsselworte und ihre Parameter werden vom Frontend verarbeitet, die übrigen werden dem Backend bzw. der Database übergeben. Die Initialisierung teilt sich somit in zwei einander überschneidende Hauptphasen auf:

- Backendinitialisierung - Funktionsaufrufe beginnen mit `bi`.
- Databasesinitialisierung - Funktionsaufrufe beginnen mit `bd`.

Die Backendinitialisierung beginnt mit dem Aufrufen einer für das Java-Backend im OpenLDAP-Server fest einkompilierten Funktion [1] `bi_init`. Hier wird nun die später benötigte Backend-Datenstruktur `Backend *bi` initialisiert. In ihr werden zum Beispiel die Zeiger für die weiteren zur Initialisierung bzw. für die LDAP-Operationen (siehe Kapitel 4.4.1) benötigten Funktionen gespeichert. Mit dem Interpretieren der Zeile 3 wird die Funktion `bi_config` des Java- Backends aufgerufen. Diese parst die Parameter und speichert die in ihr gefundenen Informationen (in diesem Fall den Classpath der JVM) für eine spätere Verwendung in der erweiterten Backend-Datenstruktur ab.

[1] Dies führt zum Nachteil, daß für jedes neues Backend der slapd neucompiliert werden muß.

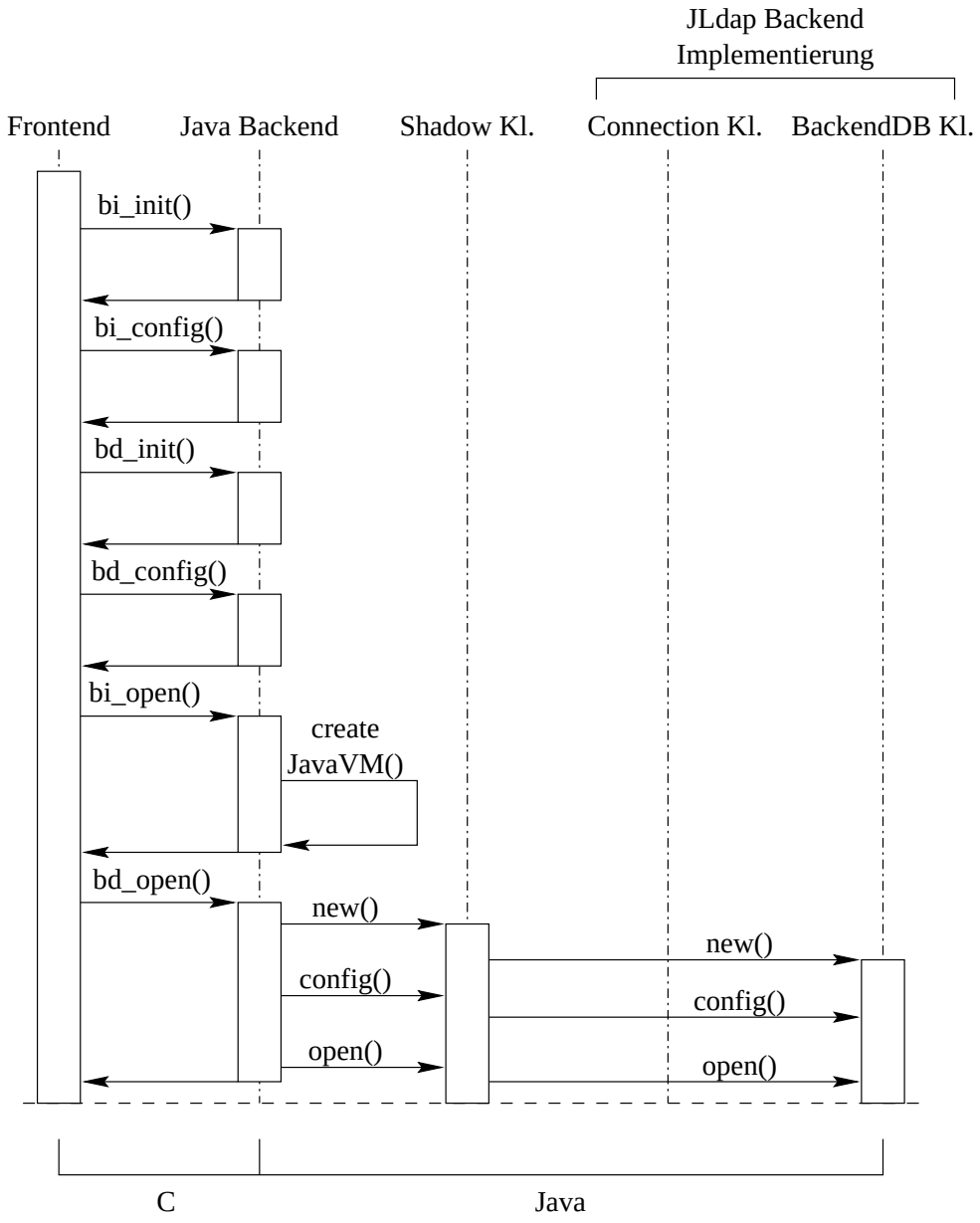

Abbildung 4.4.: Die Initialisierungsphase eines JLdap-Backends

Die Database-Initialisierung beginnt in der Zeile 6. Hierzu wird eine Funktion `bd_init` aufgerufen. In dieser Funktion wird die Database-Datenstruktur `BackendDB *bd` initialisiert. Die Zeile 7 wird vom slapd abgefangen. Sie dient zur Angabe des Teilbaums im LDAP, für den das JLdap-Backend zuständig ist (siehe Kapitel 2.3.1). In den letzten Zeilen wird die Funktion `db_config` des Java-Backends aufgerufen. Ähnlich wie für das Backend werden die Informationen in der Database-Struktur abgelegt (siehe Kapitel 4.4.1).

Nach dem Einlesen der Konfigurationsdatei wird die Funktion `bd_open` des Java-Backends aufgerufen. Da nun alle Informationen bereitstehen, wird die JVM initialisiert. Hiernach erfolgt der Aufruf der Funktion `db_open` des Java-Backends, in dieser wird eine Instanz der Shadow-Klasse (siehe Kapitel 4.4.3) erzeugt. Diese kreiert eine neue Instanz der in der Konfigurationsdatei angegebenen Klasse `BackendDB` und übermittelt dieser die gefundenen Konfigurationseinträge (dburl).

Die zur Initialisierung und Konfiguration benötigten C-Funktionen sind in der slapd-Backend-API folgendermaßen definiert:

```
/* Backend */
int (*bi_init) LDAP_P((BackendInfo *bi));
int (*bi_config) LDAP_P((BackendInfo *bi,
           char *fname, int lineno, int argc, char **argv ));
int (*bi_open) LDAP_P((BackendInfo *bi));

/* Database */
int (*bi_db_init) LDAP_P((Backend *bd));
int (*bi_db_config) LDAP_P((Backend *bd,
            char *fname, int lineno, int argc, char **argv ));
int (*bi_db_open) LDAP_P((Backend *bd));
```

Die Struktur `BackendInfo *bi` spielt eine ähnliche Rolle für die Backends wie die `Backend *bd` für die Databases. Auch hier kann durch einen Zeiger `void *bi_private` die Struktur erweitert werden. Die Funktionsweise der letzteren beiden Parameter der `config`-Routinen ist vergleichbar mit der Übergabe der Kommandozeilenparameter in Standard C. `Argc` enthält die Anzahl der Parameter, `argv[0]` das Schlüsselwort und `argv[1...n]` die Parameter. `Fname` (Dateiname) und `lineno` (Zeilennummer) dienen im Problemfalle der Erzeugung einer aussagekräftigen Fehlermeldung.

Verbindungen

In der Abbildung 4.5 ist eine beispielhafte Verbindung über ein JLdap- Backend dargestellt. Dabei ruft das Java-Backend die Methode `bind` des bereits initialisierten Shadow-Objektes auf. Dieses versucht nun anhand der übergebenen `ConnectionID` ein schon vorhandenes `Connection`-Objekt zu finden. Da der `bind`-Aufruf

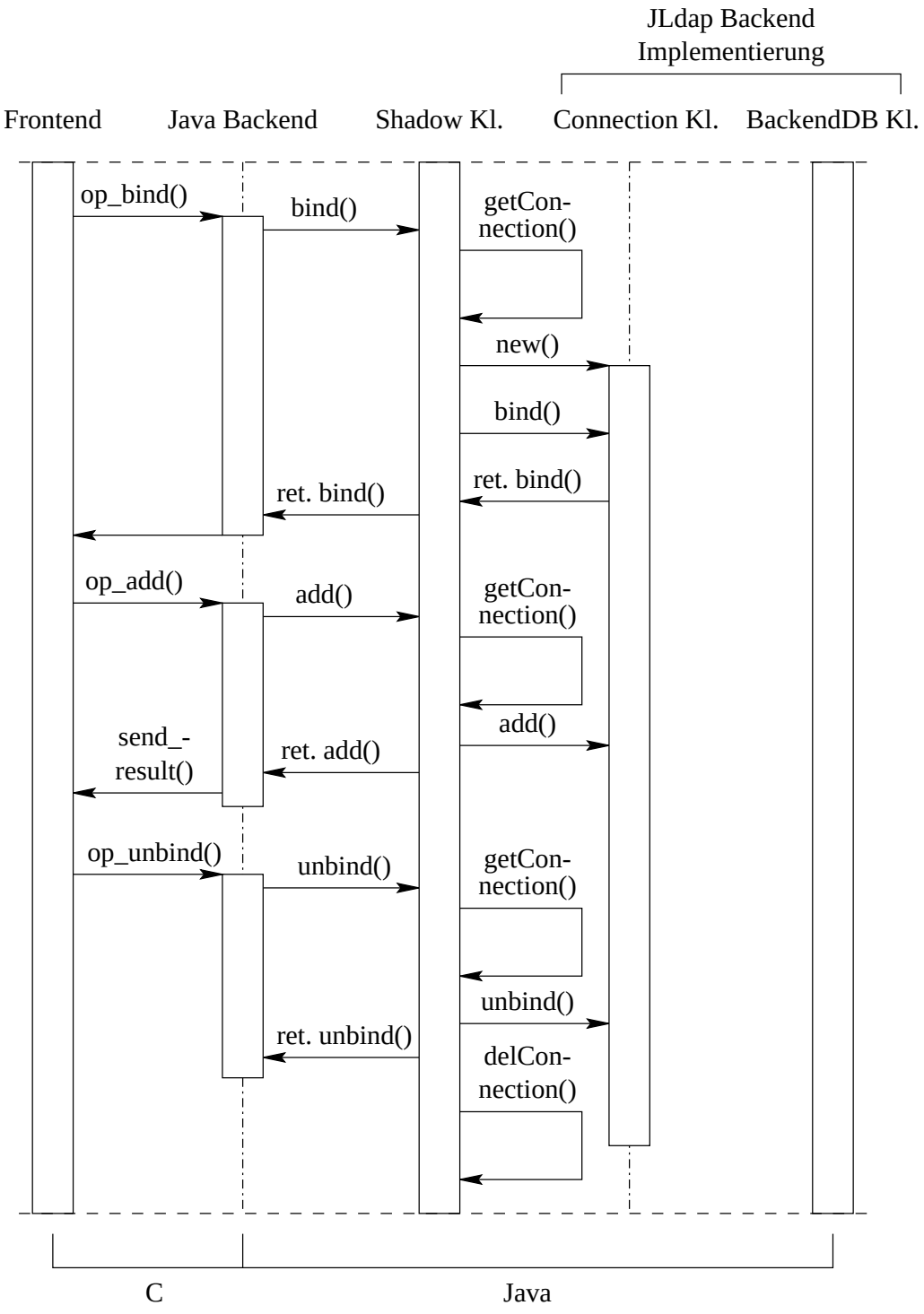

Abbildung 4.5.: Beispiel-Verbindung eines JLdap-Backends

aber zu einer neuen Verbindung gehört, schlägt dies fehl. Daraufhin wird ein neues `Connection`-Objekt erzeugt, welches ab jetzt diese Verbindung repräsentiert. Zur späteren Weiterverwendung wird es mittels eines Hash (die `ConnectionID` ist der eindeutige Schlüssel) persistent gespeichert. Darauf aufbauend kann die Methode `bind` die Implementierung der `Connection`-Klasse des JLdap-Backends aufgerufen werden. Der daraus resultierende Rückgabewert wird an das Frontend übergeben.

Während der Operation `add` findet das Shadow-Objekt ein `Connection`-Objekt zur übermittelten `ConnectionID`, über dieses Objekt wird dann die Methode `add` des JLdap-Backends aufgerufen. Diese Methode arbeitet ohne expliziten Rückgabewert. Dagegen gibt das Shadow-Objekt dem Java-Backend eine Rückmeldung (z.B. `LDAP_OPERATION_SUCCESS`), das Backend sendet dann mit Hilfe der Funktion `ldap_send_result` dieses Ergebnis über das Frontend zum LDAP Client.

Nach dem Aufrufen der Methode `unbind` im zugehörigen `Connection`-Objekt wird dieses aus dem oben erwähnten Hash gelöscht. Rückgabewerte zum LDAP-Client sind nicht mehr möglich, da die Verbindung zu diesem Zeitpunkt vom Frontend bereits geschlossen wurde.

Beenden eines JLdap-Backends

Das Beenden eines Backends wird in zwei Schritten vollzogen: `close` und `destroy`. In beiden Phasen werden immer zuerst die dazugehörigen Databases-Routinen, gefolgt von den Backend-Funktionen, aufgerufen. In der Funktion `bd_close` wird über das Shadow-Objekt das BackendDB-Objekt des JLdap-Backends mittels der Methode `close` vom Beenden des slapd benachrichtigt. In der slapd-Backend-API sind folgende Deklarationen vorhanden:

```
#Database
int (*bi_db_close) LDAP_P((Backend *bd));
int (*bi_db_destroy) LDAP_P((Backend *db));

#Backend
int (*bi_close) LDAP_P((BackendInfo *bi));
int (*bi_destroy) LDAP_P((BackendInfo *bi));
```

5. Implementierung eines Java-Backends

5.1. Überblick

Die Implementierung eines Java-Backends teilt sich in die beiden Schwerpunkte C-Backend und Java-Klassen mit dem Hauptaugenmerk auf den Übergang zwischen beiden Programmiersprachen mittels JNI (siehe Kapitel 3.2) auf.

5.2. C-Backend

5.2.1. Entwickeln eines OpenLDAP-Backends

Folgende grundlegende Schritte sind zum Erstellen eines OpenLDAP-Backends notwendig:

1. Wählen eines Namens für das Backend, in diesem Fall `java` und Erstellen eines Verzeichnisses `back-java` in der OpenLDAP-Source-Struktur unter `servers/slapd/`. In diesem Verzeichnis werden alle Routinen, die für dieses Backend

nötig sind, abgelegt. Desweiteren muß noch die Datei `Makefile.in` erzeugt werden, deren Inhalt, genauso wie die Änderungen in der Datei `configure.in`, aus schon vorhandenen Backends entnommen und angepasst werden kann.

2. Schreiben der Backend-Routinen für jede Funktion, die das Backend bereitstellen soll. Genauere Details über die verwendete API sind in Kapitel 4.4.1 zu finden. Die eigenen Backend-Routinen sollten mit dem Namen des Backends als Präfix beginnen, z.B. `java_back_op_add` für die „Add“ -Funktion, so daß ein eindeutiger Bezeichner entsteht.

3. Anpassen der Datei `servers/slapd/backend.c` durch das Hinzufügen des Aufrufs der Backend-Initialisierungsroutine.

5.2.2. Initialisierung

Die beiden Phasen der Initialisierung eines OpenLDAP-Backends wurden schon im Kapitel 4.4.4 konzeptionell behandelt. Ich werde deswegen hier nur auf Implementierungsschwerpunkte eingehen.

Während der Initialisierung werden drei C-Strukturen für die weitere Verwendung in den `void private`-Zeigern der `BackendDB *bd`- und `BackendInfo *bi`-Strukturen initialisiert. Dies sind:

java_back.h

```
1  typedef struct java_backend_instance {
2          char *classpath;
3  } JavaBackend;
4
5  typedef struct java_config_list {
6          char **argv;
7          int argc;
8          struct java_config_list *next;
9  } JavaConfigList;
10
11 typedef struct java_backend_db_instance {
12         char *conn_classname;
13         char *bdb_classname;
14         jclass class;
15         jobject object;
16         struct JavaConfigList *config_first;
17         struct JavaConfigList *config_last;
18 } JavaBackendDB;
```

JavaBackend enthält nur die Komponente classpath, welche den Klassenpfad der noch zu startenden Java Virtual Machine (JVM) beinhaltet. Dieser Wert wird in der Funktion java_back_config beim Auftreten des Schlüsselwortes Classpath in der Konfigurationsdatei des slapd gesetzt. JavaBackendDB exisiert für jedes JLdap-Backend und enthält Informationen für dieses. Dies sind conn_classname und bdb_classname für die Connection- und BackendDB-Java-Klassen, welche das JLdap-Backend darstellen. Desweiteren die einfach verkettete Liste JavaConfigList. Sie enthält die Schlüssel-/Wert-Paare, welche in der slapd-Konfigurationsdatei für dieses JLdap-Backend gefunden wurden. Auf das erste und letzte Element in dieser Liste wird mit den Zeigern config_first und config_last zugegriffen. Die Komponente class repräsentiert die Java Shadow Klasse und ist somit bei jedem JLdap-Backend gleich – im Gegensatz zu object, welche auf die Instanz der zum JLdap-Backend gehörenden Java Shadow Klasse verweist.

In der Funktion java_back_open wird die JVM initialisiert. Dies kann nicht früher erfolgen, weil erst nach dem Aufrufen der Funktion java_back_config der Java-Klassenpfad aus der slapd-Konfigurationsdatei gelesen wird.

java_back_open

```
1  /* initialize vm_args struct */
2  vm_args.version = JNI_VERSION_1_2;
3  vm_args.options = options;
4  vm_args.nOptions = 3;
5  vm_args.ignoreUnrecognized = JNI_FALSE;
6
7  /* Create the JVM */
8  res= JNI_CreateJavaVM(&jvm, (void **)&env, &vm_args);
9  if (res < 0) {
10         Debug( LDAP_DEBUG_TRACE, "java_back_open - Can't create Java
11         VM: %i\n", res, 0, 0 );
12         return -1;
13 }
14 return 0;
```

In der Struktur vm_args werden die Startparameter für die JVM hinterlegt. Hierbei enthält options ein Feld von Parameter-Strings in der von einem Kommandozeilenaufruf des Java-Interpreters bekannten Form, z.B. -classpath java/mybackend.jar. Die Komponente nOptions gibt die Anzahl dieser Parameter an. Mit Hilfe der Option ignoreUnrecognized kann das Verhalten beim Auftreten von nicht bekannten Parametern gesteuert werden. Das Setzen auf JNI_FALSE erzwingt in diesem Fall eine Fehlermeldung. Der Eintrag JNI_VERSION_1_2 legt die benutzte Java-Version auf 1.2 fest. Mit dem Aufruf der Funktion JNI_CreateJavaVM wird die JVM gestartet. In den beiden Parametern jvm und env werden Informationen über die JVM (jvm)

und über die Umgebung des erzeugten Java-Threads abgelegt. Damit kann später auf Java-Methoden zugegriffen werden. Tritt ein Fehler während der Initialisierung der JVM auf, wird eine Fehlermeldung mit einem bestimmten Level (`LDAP_DEBUG_TRACE`) erzeugt. Es ist zu beachten, daß bei dieser Funktion das slapd-Frontend noch den Rückgabewert beachtet (im Gegensatz zu den LDAP-Operationsfunktionen). So wird bei einem Rückgabewert, der nicht Null entspricht, das Starten des slapd abgebrochen.

In der Funktion `java_back_db_open` wird erstmalig über die initialisierte JVM ein neues Java-Objekt instantiiert und eine Java-Methode aufgerufen. Dafür geht man in folgenden Schritten vor:

1. Die Klasse mittels ihres Namens und der Funktion `FindClass` auffinden.
2. Die benötigte Methoden-ID über ihren Namen, die zugehörige Klasse und die Übergabeparameter (Signatur) ermitteln. Für den Konstruktor einer Klasse besteht die Konvention, daß als Methodenname `<init>` verwendet wird.
3. Eine neue Instanz mit Hilfe der `NewObject`-Funktion und der Methoden-ID, der Klasse sowie der Methodenparameter bilden.

Über das so erstellte Objekt können nach gleichem Muster weitere Methoden aufgerufen werden. Dafür stellt das JNI die `CallVMethod`-Funktionen zur Verfügung. Hierbei ist zu beachten, daß für jeden primitiven Java-Typ als Rückgabeparameter eine spezielle Funktion benutzt werden muß; z.B. für Integer-Werte zur Rückgabe `CallIntMethod` oder bei nicht primitiven Werten wie z.B. Strings `CallObjectMethod`.

java_back_db_open

```
1  /* find Shadow class org.openldap.jldap.JLdap */
2  java_back_db->class = (*env)->FindClass(env, JAVA_SHADOW_CLASS);
3
4  ccls = (*env)->FindClass(env, java_back_db->conn_classname);
5  bcls = (*env)->FindClass(env, java_back_db->bdb_classname);
6
7  /* find Shadow class contructor */
8  mid = (*env)->GetMethodID(env, java_back_db->class, "<init>",
9                        "(Ljava/lang/Class;Ljava/lang/Class;)V");
10
11 /* create Shadow class instance */
12 java_back_db->object=(*env)->NewObject(env, java_back_db->class,
13                                         mid, ccls, bcls);
```

Den Quelltextausschnitt der Funktion `java_back_db_open` habe ich der Übersichtlichkeit wegen um das Abfangen der Fehler gekürzt. In den Zeilen 1 – 5 werden die Shadow-, `Connection-` und `BackendDB-` Java-Klasse mittels der Funktion

`FindClass` gesucht — die beiden letzteren Klassen sind je nach JLdap-Backend unterschiedlich. Die Klassen `Connection` und `BackendDB` dienen später (Zeile 12) als Übergabeparameter für den Konstruktor der Java Shadow-Klasse. Daß der Zeiger `env` doppelt im Funktionsaufruf vorkommt, ist der Tatsache geschuldet, daß das JNI gleichzeitig für den Gebrauch unter C und C++ konzipiert wurde. Sind alle drei Klassen erfolgreich ermittelt, wird unter Zuhilfenahme der Funktion `GetMethodID` die Methoden-ID des Konstruktors (`<init>`) der Shadow-Klasse erzeugt. Die dazu verwendete Methoden-Signatur kann über das Hilfsprogram `javap` mit der Option `-s` und der Java-Klasse als Parameter erstellt werden. In der Zeile 12 wird dann eine neues Objekt der Shadow-Klasse instantiiert, welches für eine spätere Verwendung in anderen C-Funktionen genau wie die Shadow-Klasse in der Struktur `java_back_db` hinterlegt wird.

java_back_db_open

```
1 /* call config method */
2 mid = (*env)->GetMethodID(env, java_back_db->class, "config",
3                            "(I[Ljava/lang/String;)I");
4 res_conf=(*env)->CallIntMethod(env,java_back_db->object,mid,
5                                 (jint) clist->argc, jargv);
```

Nach dem eine Instanz der Shadow-Klasse erstellt wurde, werden die in der slapd-Konfigurationsdatei für dieses JLdap-Backend enthaltenen Einträge an das Shadow-Objekt durch das Aufrufen der Methode `config` übergeben. Dafür wird über die in der Funktion `java_back_db_config` erzeugte, einfach verkettete `JavaConfigList` iteriert und für jeden Eintrag die Methode `config` aufgerufen.

5.2.3. LDAP-Operationen

Nach der Initialisierung der JLdap-Backends steht der slapd-Server für LDAP-Anfragen bereit. Betreffen diese Anfragen Teile des LDAP-Baums, für den ein JLdap-Backend zuständig ist, wird die zur LDAP-Operation korrespondierende C-Funktion mit den im Kapitel 4.4.1 beschriebenen Parametern aufgerufen. Im Falle eines Vergleiches heißt diese Funktion `java_back_op_compare`. Die C-Funktionen setzen nun die Anfrage in einen Java-Methodenaufruf um. Dabei wird immer der gleiche Ablauf eingehalten:

1. Sperren der JVM. Dies geschieht aus Sicherheitsgründen, obwohl die JVM „thread-safe“ ist. Aber kleine Testprogramme ergaben nur schwer nachvollziehbare Probleme in der Zusammenarbeit von Native- und Java-Threads. Eine vollständige Analyse dieser Probleme konnte nicht im Rahmen dieser Diplomarbeit erfolgen.

2. Anhängen (attach) der JVM an den Native-Thread. Dies muß in jeder Funktion erfolgen, denn das Frontend erzeugt für jede Verbindung einen neuen Native-Thread und es ist nicht voraussagbar, welche LDAP-Operation (und damit: welche dazugehörige C-Funktion) als erste aufgerufen wird.

3. Ermitteln der Methoden-ID der in der Java Shadow-Klasse aufzurufenden Methode.

4. Umwandeln der C-Parameter in für Java verständliche Typen. So muß z.B. aus einem `char *` in C ein `java.lang.String` Objekt für Java erstellt werden.

5. Aufrufen der Java-Methode in der zum JLdap-Backend gehörenden Instanz der Shadow-Klasse.

6. Freigeben (unlock) der JVM.

7. Aufbereiten des Rückgabewertes der Java-Methode und Zurücksenden an den LDAP-Client.

Als Beispiel zur genaueren Erklärung dieses Ablaufes habe ich die Funktion `java_back_op_compare` gewählt. Diese beinhaltet alle aufgezählten Aspekte und läßt somit auch einfach Rückschlüsse auf andere LDAP-Operationsfunktionen zu:

java_back_op_compare

```
1  /* JVM lock */
2  ldap_pvt_thread_mutex_lock( &jvm_mutex );
3
4  /* attache JVM to current thread */
5  attres=(*jvm)->AttachCurrentThread(&jvm, (void **)&env, NULL);
6  if (attres != 0){
7          Debug( LDAP_DEBUG_ANY, "java_back_compare - Can't attach
8                  current thread to JVM\n", 0, 0, 0 );
9          send_ldap_result(conn, op, LDAP_OPERATIONS_ERROR, NULL,
10                         "Internal Server Error");
11         ldap_pvt_thread_mutex_unlock( &jvm_mutex );
12         return 1;
13 }
```

In der Zeile 2 wird die JVM mit Hilfe eines globalen Mutex gesperrt. Danach wird der Native-Thread bei der JVM angemeldet. Die Parameter sind identisch mit der schon besprochenen `CreateJavaVM`-Funktion. Der danach folgende Test auf einen Fehler wurde bei den weiteren Quelltextausschnitten aus Gründen der Übersichtlichkeit wieder weggelassen; er läuft immer nach dem gleichen Schema ab. Es ist dabei zu beachten, daß das Frontend den Rückgabewert der Funktion ignoriert.

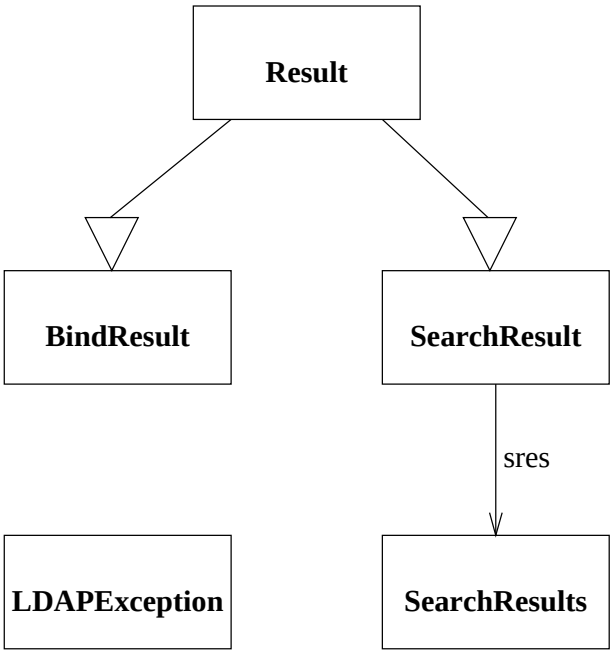

Abbildung 5.1.: Klassendiagramm der `Resultat` Klassen

Deshalb muß der LDAP-Client innerhalb der Funktion mit der in der slapd-Backend-API bereitgestellten Hilfsfunktion `send_ldap_result` über den aufgetretenen Fehler benachrichtigt und die JVM wieder entsperrt werden.

java_back_op_compare

```
1 /* find methode compare in Shadow class */
2 mid = (*env)->GetMethodID(env, java_back_db->class, "compare",
3       "(JLjava/lang/String; Ljava/lang/String;Ljava/lang/String;)
4       Lorg/openldap/jldap/Result;");
5
6 /* create Java dn string */
7 jdn = (*env)->NewStringUTF(env, dn);
```

Treten keine Fehler auf, wird die Methoden-ID der Methode `compare` in der Shadow-Klasse ermittelt. Danach wird der Parameter `dn` in ein Java String-Objekt unter Verwendung der im JNI enthaltenen Funktion `NewStringUTF` umgewandelt. Dies erfolgt nach dem gleichen Verfahren auch mit dem Attributtyp und -wert (nicht dargestellt).

java_back_op_compare

```
1 /* call methode compare in Shadow class */
2 jres=(*env)->CallObjectMethod(env,java_back_db->object, mid,
```

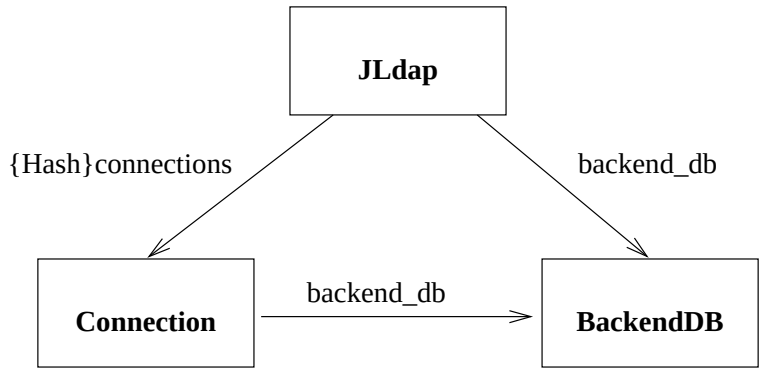

Abbildung 5.2.: Klassendiagramm JLdap-Schnittstelle

```
3       (jlong)conn->c_connid, jdn, jtype, jvalue);
4
5   /* unlock JVM */
6   ldap_pvt_thread_mutex_unlock( &jvm_mutex );
7
8   /* send Java Result to ldap client */
9   send_result(bd, conn, op, jres);
```

Sind alle Parameter umgewandelt, wird die Methode `compare` in der zugehörigen Instanz der Shadow-Klasse aufgerufen. Zusätzlich wird ihr die ID der Verbindung (`conn->c_connid`) übergeben. Da es sich beim Rückgabewert um ein von mir entwickeltes Java- `Result`-Objekt (`org.openldap.jldap.Result`) handelt, muß der Aufruf `CallObjectMethod` verwendet werden. Danach kann die JVM entsperrt werden. Zuletzt wird noch das `Result`-Objekt in der `send_result`- Funktion ausgewertet und das Ergebnis an den Client gesendet. Hierbei wird nicht zwischen einer Fehlermeldung (z.B. durch eine LDAPException ausgelöst) und der Rückgabe des Vergleichresultates unterschieden, denn beide Ereignisse werden nur über einen unterschiedlichen LDAP-Fehlerkode dargestellt. Bei komplexeren Resultaten, z.B. einem Suchresultat, wird mit einer von `Result` abgeleiteten Klasse gearbeitet (Abb. 5.1), so daß für die Rückgabe eines Fehlers immer noch `send_result` benutzt werden kann. Für das Senden des Suchresultates selbst steht dann die Funktion `send_search_result` zur Verfügung.

5.3. Java-Klassen

Die in dieser Arbeit entwickelten Java-Klassen sind in dem Java-Paket `org.openldap.jldap` zusammengefasst. Die Hauptaufgabe des Verbindens der in C definierten slapd-Backend-API mit der objektorientierten JLdap-Schnittstelle übernimmt die

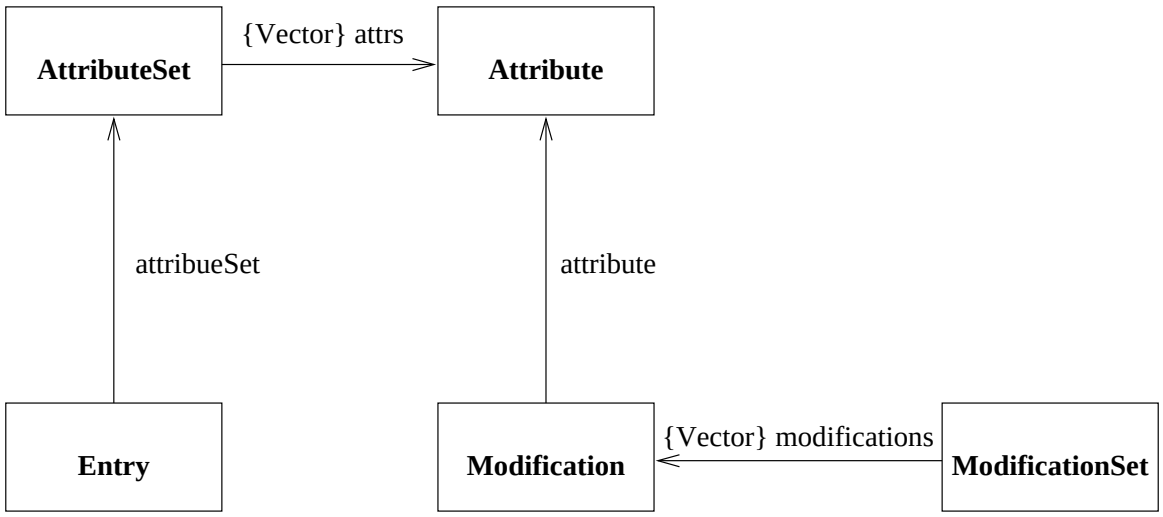

Abbildung 5.3.: Klassendiagramm weiterer verwendeter Klassen

Shadow-Klasse namens `JLdap`. Die beiden Klassen `Connection` und `BackendDB` bilden die JLdap-Schnittstelle selbst (Abb. 5.2) und dienen somit als Vorlage für die Entwicklung eines JLdap-Backends, denn diese Backends müssen mindestens zwei Klassen implementieren, welche einmal von `Connection` sowie von `BackendDB` abgeleitet sind. Desweiteren stehen noch Klassen zur Verfügung, die die weiteren in der slapd-Backend-API verwendeten C-Datenstrukturen abbilden (Abb. 5.3).

5.3.1. JLdap-Klasse

Die JLdap-Klasse bildet die Shadow-Klasse und wird somit für jedes JLdap-Backend einmal instantiiert. Dabei werden dem Konstruktor die beiden Klassen übergeben, welche die `Connection` und die `BackenDB` für dieses JLdap-Backend repräsentieren.

JLdap

```
1  private Class conn_class;
2  private BackendDB backend_db;
3  private Hashtable connections;
4
5  public JLdap(Class cclass, Class bdbclass)
6              throws java.lang.InstantiationException,
7                     java.lang.IllegalAccessException {
8          connections=new Hashtable();
9          conn_class=cclass;
10         backend_db=(BackendDB) bdbclass.newInstance();
11 }
```

Wie schon in der Konzeption der Java-Shadow-Klasse (siehe Kapitel 4.4.3) erwähnt, ist diese Klasse unter anderem für das Erzeugen und Verwalten von Objekten, welche eine LDAP-Verbindung abbilden, zuständig. Hierzu dient die Member-Variable `connections` vom Typ `Hashtable`. Diese wird im Konstruktor der Shadow-Klasse initialisiert. Weitere Member-Variablen der Shadow-Klasse sind `conn_class` (die `Connection`-Klasse des JLdap-Backends) und `backend_db` (die Instanz der Klasse `BackendDB` des JLdap-Backends). Letztere wird im Konstruktor der Shadow-Klasse aus der übergebenen Klasse `bdbclass` in Zeile 10 erzeugt.

JLdap.getConnectionObject

```
1  protected Connection getConnectionObject(Long conn) throws
2  java.lang.InstantiationException, java.lang.IllegalAccessException {
3
4          Connection cobject;
5
6          if (connections.containsKey(conn)){
7                  return (Connection) connections.get(conn);
8          } else {
9                  Constructor c = conn_class.getConstructor(new Class[]
10                                         {Connection.class, Long.class });
11                         cobject=(Connection) c.newInstance(
12                                 new Object[] {backend_db,conn});
13                 connections.put(conn, cobject);
14         }
15         return cobject;
16 }
```

Für die Handhabung der Objekte, die eine LDAP-Verbindung repräsentieren, ist die Methode `getConnectionObject` zuständig. Sie wird von jeder Methode in der Shadow-Klasse aufgerufen, welche für eine LDAP-Operation steht, um das zur Verbindung gehörende Objekt zu ermitteln. Als Parameter dient die eindeutige Connection-ID. Diese wird auch intern als Schlüssel für die Hashtabelle (`connections`) verwendet, der dazu gehörige Wert entspricht dem Verbindungs-Objekt. Die `getConnectionObject`-Methode geht dabei nach einem einfachen Algorithmus vor: Existiert zur übergebenen Connection-ID ein Eintrag in der Hashtabelle, wird das als Wert gespeicherte Objekt zurückgegeben. Ist in der Hashtabelle noch kein passendes Objekt vorhanden, wird ein neues vom Typ der `conn_class` instantiiert (Zeile 11) und ihm seine zugehörige Connection-ID sowie das `BackendDB`-Objekt übermittelt. Abschließend wird es noch in die Hashtabelle (mit der Connection-ID als Schlüssel) eingetragen und an den Aufrufenden zurückgegeben. Das Löschen von Einträgen in der Hashtabelle geschieht in der Methode `unbind`.

Als Beispiel für die Implementierung einer LDAP-Operation habe ich wieder `compare` gewählt. Sie verdeutlicht sehr gut die grundlegenden Schritte aller derjenigen Java-Methoden in der Shadow-Klasse, welche LDAP-Operationen abbilden.

JLdap.compare

```
1  public Result compare(long conn, String dn, String ava_type,
2                    String ava_value)
3                    throws java.lang.InstantiationException,
4                           java.lang.IllegalAccessException {
5       Long lconn= new Long(conn);
6       Connection cobject= getConnectionObject(lconn);
7       try{
8          boolean res=cobject.compare(dn,
9          new Attribute(ava_type, ava_value));
10         if (res == true){
11                 return new Result (6, "", "");
12         }
13      }
14      catch (LDAPException e){
15         return new Result (e.getLDAPResultCode(),e.getMatchedDN(),
16                            e.getLDAPErrorMessage());
17      }
18      return new Result (5, "", "");
19 }
```

Zuerst wird die Connection-ID (`conn`) in Zeile 5 vom primitven Java-Typ `long` in das Objekt `lconn` der Klasse `Long` gewandelt. Dies geschieht, weil die Benutzung von primitiven Java-Typen in C sehr einfach ist, aber die Java-Hashtabelle `connections` ein Objekt als Schlüssel (keine primitiven Typen) benötigt. In der Zeile 6 wird dann das zur Connection-ID gehörende Objekt ermittelt und der Variablen `cobject` zugewiesen. Über sie wird die Methode `compare` aufgerufen, welche im JLdap-Backend implementiert ist. Dabei werden die beiden Übergabeparameter `ava_type` und `ava_value` in ein Objekt der Klasse `Attribute` umgewandelt. Je nach Rückgabewert der aufgerufenen Methode `compare` wird ein Objekt `Result` erstellt. Tritt eine Ausnahme auf (`LDAPException`), so wird sie abgefangen und als Objekt `Result` an die aufrufende C-Funktion `java_back_op_compare` zurückgegeben. Solch eine Ausnahme tritt z.B. dann auf, wenn keine Methode `compare` im JLdap-Backend existiert.

5.3.2. Die Klassen BackendDB und Connection

Die beiden Klassen `BackendDB` und `Connection` bilden die Grundlage für die Implementierung eine JLdap-Backends. Von beiden muß eine abgeleitete Klasse im

jeweiligen JLdap-Backend existieren. Die sich im Paket `org.openldap.jldap` befindlichen Superklassen fungieren dafür gleichzeitig als Schnittstellendefinition und Adapter. Sie erstellen mit ihren Methoden und deren Parameter eine Vorlage und implementieren diese gleich mit den notwendigsten Funktionen.

Connection.compare

```
1 public boolean compare(String dn, Attribute attr)
2         throws LDAPException {
3       if (1==1) {
4           throw new LDAPException("Function compare not implemented",
5                                   LDAPException.UNWILLING_TO_PERFORM,null);
6       }
7       return false;
8 }
```

Diese Funktionen beschränken sich darauf, daß eine Ausnahme mit dem Fehlerkode „Funktion nicht implementiert“ ausgelöst wird. Dieses Vorgehen vereinfacht das Entwickeln von JLdap-Backends, denn dadurch kann man sich bei einer Implementierung auf die benötigten Methoden konzentrieren. Besonders in einer frühen Phase der Entwicklung wird dies sehr hilfreich sein.

5.4. Installation und Konfiguration

Das Installieren des slapd erfolgt Unix-typisch über ein Skript `configure`; genauere Hinweise zu allen Optionen und Möglichkeiten gibt es in den `README`- und `INSTALL`-Dateien, welche sich im Wurzel-Verzeichnis des Sourcebaumes des Projektes OpenLDAP befinden. Eine beispielhafte Installation könnte folgendermaßen aussehen:

Installation

```
1 tar xvzf openldap.tgz
2 cd ldap
3 export JDKHOME=/opt/jdk1.2.1
4 ./configure --prefix=/opt/ldap --enable-java --disable-ldbm
5 make dep
6 make
7 make install
```

Zuerst muß das Archiv mit dem Quellcode entpackt werden. Außerdem ist die Umgebungsvariable `JDKHOME` auf das Installationsverzeichnis des JDK zu setzen. Dies wird zum Auffinden der JNI-Header und -Bibliotheken benötigt. Der Aufruf

`configure` erstellt speziell für das benutzte System geeignete `Makefiles`. Die Option `prefix` gibt das Zielverzeichnis für die spätere Installation an; `enable-java` aktiviert das Java-Backend, wogegen `disable-ldbm` das standardmäßig benutzte Backend `ldbm` deaktiviert. Dies geschieht, weil die von mir benutzte Entwicklerversion des slapd Probleme mit der Kompilierung dieses Backends unter Solaris 7 hat. Der folgende Aufruf `make dep` erstellt die Abhängigkeiten zwischen den einzelnen Quellkode-Dateien. Durch das Ausführen des Befehls `make` wird das Kompilieren angestoßen. Nach dem erfolgreichen Linken können mittels `make install` alle Komponenten installiert werden.

Die Konfiguration des slapd erfolgt hauptsächlich über die Datei `etc/openldap/slapd.conf`. Allgemeine Hinweise gibt es in der entsprechenden Manpage [5]; ein Beispiel mit einer Erläuterung der für das Java-Backend wichtigen Schlüsselworte `Classpath`, `BackendDB` und `Connection` sind im Kapitel 4.4.4 zu finden.

Vor dem Starten des slapd muß beachtet werden, daß sich die Native-Thread-Versionen der JNI-Bibliotheken im Suchpfad des Runtime-Linkers befinden. Dies erfolgt am einfachsten durch das Setzen der Umgebungsvariable `LD_LIBRARY_PATH` auf: `$JDKHOME/jre/lib/sparc/lib/`. Der Start des slapd erfolgt dann z.B. folgendermaßen:

```
sbin/slapd -d 255 -p 2604
```

Der Parameter `d` schaltet die Debugmeldungen ein, und der Schalter `p` teilt dem slapd mit, daß er am TCP/IP Port 2604 auf eingehende Verbindungen warten soll. Dieses Vorgehen ermöglicht es, den Server zu Testzwecken als normaler Nutzer zu starten. Denn die Benutzung des Standard-LDAP-Ports 319 ist unter UNIX nur dem Administrator (root) gestattet[1]. Weitere Optionen und genaue Erklärungen dazu können der slapd-Manpage [4] entnommen werden.

[1] Dieses Sicherheitskonzept betrifft alle Ports kleiner 1024.

6. Beispielanwendung

6.1. Überblick

Die in dieser Arbeit entwickelte JLdap-Schnittstelle wird mit dieser Beispielanwendung konzeptionell überprüft und getestet. Dabei wurde das Hauptaugenmerk nicht auf die vollständige Implementierung des Lösungsansatzes für das Einsatzgebiet gelegt, sondern auf die Überprüfung der im Entwurf definierten Schnittstelle (die Referenz befindet sich im Anhang A) im praktischen Einsatz.

6.2. SQL-Backend

Eine der Motivationen zum Erstellen eines LDAP-Backends ist es, eine bestehende relationale Datenbank [46] im LDAP-Baum zu integrieren. Dafür bietet Java mit JDBC eine ideale Schnittstelle. Beim genaueren Analysieren des Problems werden mehrere Schwerpunkte ersichtlich, die zu beachten sind:

- Die erforderliche Befehlsumsetzung zwischen LDAP und SQL; z.B. muß aus einer LDAP- `modify` eine SQL- `update`, `delete` oder `insert into` Operation entstehen.

- Die Suchsyntax ist in LDAP und SQL verschieden definiert. Zum Beispiel `cn=foo*` könnte in einer SQL-`where`-Klausel `cn LIKE "foo%"` lauten.
- Das Mapping zwischen LDAP-Attribut-Typen und SQL-Tabellen und -Spalten; z.B. könnte das LDAP-Attribut `cn` sich in der SQL-Tabelle `user` und der Spalte `name` befinden.
- Der Typ „Case Ignore Strings“(Groß-/Kleinschreibung wird nicht beachtet) ist in SQL nicht bekannt, wird aber in LDAP von vielen Attribut-Typen genutzt.
- Die hierarchische Struktur von LDAP ist in SQL-Tabellen nur über zusätzliche Verbindungstabellen oder „Self Joins“ abbildbar. Eine solche Struktur kann aber nicht mit nur einer einzigen `select`-Anweisung vollständig durchsucht werden.
- SQL verwendet eine genauere Typisierung als LDAP. Beispielsweise besteht in SQL ein Unterschied zwischen `answer=42` (Integer) und `answer="42"` (Char). Demgegenüber kann die Anfrage in LDAP jeweils `answer=42` lauten.

6.2.1. Suchsyntax

Für die Umsetzung der Suchsyntax zwischen LDAP und der `where`-Klausel einer SQL-`select` Anweisung ist zwischen einem iterativen oder rekursiven Ansatz zu wählen. Da die Tiefe der Rekursion nicht an die Grenzen heutiger verfügbarer Systemressourcen stößt, habe ich mich für den rekursiven Ansatz entschieden.

Filter.Filter

```
1         if (filter.substring(1,2).equals("&")){
2                         System.out.println("AND detected");
3                         filtercomp=new String("AND");
4                 }
5                 else {...
```

Zuerst wird der Filterstring [14] nach dem Typ der Verknüpfung durchsucht. Beispielsweise würde der LDAP-Filterstring `(&(uid=test) (mail=test@fhtw-berlin.de))` als UND-Verküpfung erkannt werden. Wenn keine Verknüpfung gefunden wird, handelt es sich um ein Attribute/Wert-Paar.

Filter.Filter

```
1 item=filter.replace('*','%');
2 int index=item.indexOf("=");
3 if (filter.compareTo(item)!=0){
4         item= "("+ item.substring(1,index) + " LIKE '"
```

```
5           + item.substring(index + 1, item.length() - 1) + "')";
6   }
7   else{
8           item="("+ item.substring(1,index) + "='"
9                 + item.substring(index+1,item.length()-1) + "')";
10  }
```

Als nächster Schritt werden alle im Attribut/Wert-Paar enthaltenen „*“ durch „%“ ersetzt. Wenn ein „*“ enthalten war, wird auch das „=“ durch den SQL-Vergleichsoperator „LIKE“ ersetzt. Zum Beispiel wird aus dem LDAP-Suchstring „cn=Mik*“ in der SQL-where-Klausel „cn LIKE "Mik%"“.

Filter.Filter

```
1   for(int i=parse_bg; i<filter.length();i++){
2           if(filter.substring(i, i+1).equals("(")){
3                   bcounter++;
4           }
5           if(filter.substring(i, i+1).equals(")")){
6                   bcounter--;
7           }
8           if (bcounter==0){
9                   Filter fpart=new Filter(filter.substring(part_bg,i+1));
10                  fparts.addElement(fpart);
11                  part_bg=i+1;
12          }
13  }
```

Wenn der LDAP Suchstring weitere Unterfilter enthält, z.B. in (&(|(uid=test) (uid=foo)) (cn=Mike*)) im UND (&) ein ODER (|), so werden diese aufgeteilt und als eigenständige Filter rekursiv weiterverarbeitet.

In der Methode toSql werden dann alle Teilfilter wieder zusammengesetzt und mit den entsprechen SQL-Verknüpfungsoperatoren versehen, z.B. (uid="test")OR(uid= "foo") AND (cn LIKE "Mike%").

6.2.2. SQL-Views

Das Mapping zwischen den LDAP Attribut-Typen und den SQL-Tabellen und -Spalten wurde auf der Datenbankseite gelöst. Hier wurde ein SQL-View mit dem Namen ldap angelegt.

Views sind virtuelle Tabellen. Durch sie ist es möglich, Spalten aus verschiedenen Basistabellen zusammenzuführen und deren Spaltennamen umzubenennen (siehe Abbildung 6.1).

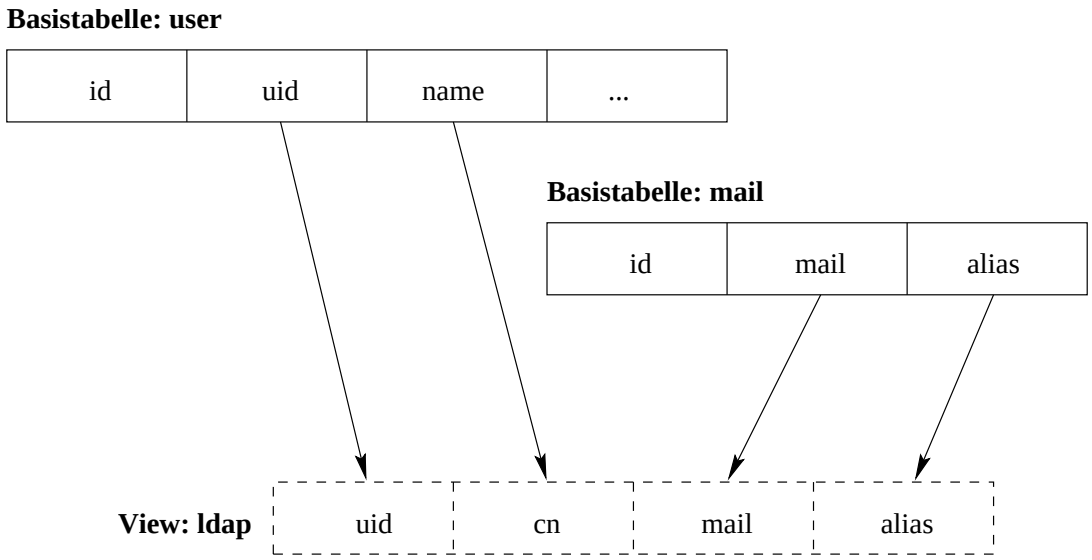

Abbildung 6.1.: Zusammenstellung eines Views

Die Benutzung solcher Views vereinfacht die Problematik stark. Durch die Verwendung eines Views läßt sich eine für das JLdap-Backend transparente Schnittstelle aufbauen. Ändern sich z.B. dahinterliegende Basistabellen, sind keine Änderungen im JLdap-Backend notwendig, weil nur das View angepasst werden muß.

Leider stellen SQL-Views auch eine Einschränkung dar. So kann nicht jede SQL-Änderungsoperation in einer Datenbank auf einem View ausgeführt werden. Ist diese Limitierung bei Spalten welche aus einem arithmetischen Ausdruck abgeleitet sind noch einfach nachvollziehbar, kann die Beschränkung auf das Verwenden einer einzigen Basistabelle nur durch die dadurch entstehende Komplexität erklärt werden. Diese Einschränkung konnte in dieser Arbeit durch das Nutzen von vorhandenen „Stored Procedures“ umgangen werden. Dadurch kann dieser Ansatz aber nicht mehr als allgemein angesehen werden.

6.2.3. Case Ignore Strings

Die Problematik der Verwendung von „Case Ignore Strings“ in LDAP und der Tatsache, daß in SQL kein Äquivalent dazu existiert, wird dadurch verstärkt, daß das slapd-Frontend bei einer Suche keine Informationen über die Verwendung eines „Case Exact“ oder „Case Ignore Strings“ übermittelt. Dies kann nur durch ein eigenständiges Parsen der `etc/openldap/slapd.at.conf` Datei ermittelt werden. In ihr befinden sich standardmäßig die Syntaxbeschreibungen der einzelnen LDAP-Attribute, z.B. für die Nutzerkennung (uid): `attribute uid cis`. Das Kürzel `cis` bedeutet: „case ignore string“.

Ist die Syntaxbeschreibung bekannt (bei kleineren Projekten könnten sie z.B. im JLdap-Backend selbst verwaltet werden), kann in SQL durch die Verwendung der

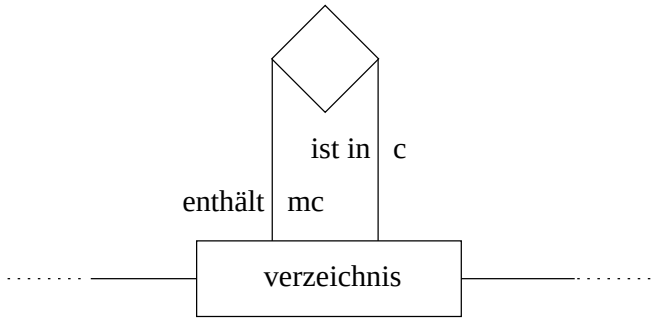

Abbildung 6.2.: Entity Relationship Modell einer hierarchischen Struktur

Funktion `UCASE` ein „Case Ignore String“ simuliert werden. Ermöglicht wird dieses Vorgehen dadurch, daß das slapd-Frontend im Falle von „Case Ignore Strings“ den Suchstring komplett in Großbuchstaben umwandelt. Eine SQL-Abfrage über „Case Ignore Strings“ muß also folgendermaßen aussehen:

```
select * from ldap where UCASE(uid)="FOO"
```

6.2.4. Typisierung

Erfolgt der Zugriff auf verschiedene SQL-Datentypen (Integer, Char usw.), muß das Quoting dem Typ angepaßt werden. Dazu kann der Typ einer Spalte in der SQL-Datenbank entweder dynamisch aus den sogenannten Metadaten einer Tabelle ermittelt oder im JLdap-Backend statisch verankert werden.

6.2.5. Hierarchische Struktur

Eine hierarchische Struktur könnte z.B., wie im Entity Relationship Model (Abb. 6.2) dargestellt, in einer relationalen Datenbank abgebildet sein. In diesem Falle kann die Struktur nicht mehr in einem SQL-View dargestellt werden, der Abfragen oder Änderungen ermöglichen würde. Daher muß die Struktur bei einer LDAP-Anfrage über einen Teilbaum (`SCOPE_SUB`) rekursiv mittels mehrer SQL-Anweisungen durchsucht werden. Dies kann auf der Datenbankseite mit „Stored Procedures“geschehen, oder im JLdap-Backend – welcher aus Sicht der Datenbank einen Client darstellt – durch einen Java-Programmabschnitt realisiert werden. In dieser Beispielanwendung wurde der Fall der rekursiven LDAP-Anfragen aus Gründen der Komplexität und dem Schwerpunkt der Überprüfung der JLdap-Schnittstelle, nicht beachtet.

6.2.6. Implementierung

Wie für die Implementierung eines JLdap-Backends notwendig, habe ich jeweils von den Klassen `BackendDB` und `Connection` eine Klasse abgeleitet. Dies sind `Dorm-`

`LdapBackendDB` sowie `DormLdapConnection`, beide befinden sich mit noch weiteren Hilfsklassen im Java-Paket `de.fhtw.jldap`.

DormLdapBackendDB

Die Klasse `DormLdapBackendDB` implementiert die Methoden `config`, `open`, `close` und `getSqlConnection`. Der Konstruktor wird nicht neu implementiert, weil keine Spezialisierung gegenüber der Superklasse notwendig ist.

DormLdapBackendDB.config

```
1  public void config(int argc, String[] argv){
2          if (argv[0].equals("jdbc")){
3                  jdbc=argv[1];
4          }
5          if (argv[0].equals("dburl")){
6                  dburl=argv[1];
7          }
8          if (argv[0].equals("dbdb")){
9                  dbdb=argv[1];
10         }
11         if (argv[0].equals("dbpasswd")){
12                 dbpasswd=argv[1];
13         }
14 };
```

In der Methode `config` werden die in der slapd-Konfigurationsdatei eingetragenen Einstellungen für dieses Backend in Membervariablen des Objektes für eine spätere Verwendung gespeichert. Die zusätzlichen Konfigurationsparameter sind:

jdbc Die Java-Klasse, die den JDBC-Treiber darstellt.

dburl Die Angabe über den Host, auf welchem der Datenbankserver läuft.

dbdb Die für die Anfragen verwendete Datenbank.

dbpasswd Das für die Datenbank benötigte Passwort.

DormLdapBackendDB.open

```
1  public void open(){
2          Class.forName(jdbc);
3          Conn = DriverManager.getConnection(dburl,dbdb,dbpasswd);
4  }
```

Die Methode `open` wird nach dem Übergeben aller Konfigurationsparameter aufgerufen und ermöglicht es somit, nun die Datenbankverbindung aufzubauen. Über die Methode `getSqlConnection` kann dann aus der LDAP-Verbindung auf diese Datenbankverbindung zugegriffen werden. Dadurch kann das sonst notwendige Auf- und Abbauen von Datenbankverbindungen während einer LDAP-Verbindung entfallen. Beim Aufruf der Methode`close` wird die Datenbankverbindung geschlossen, dies geschieht beim Beenden des slapd.

DormLdapConnection

In der Klasse `DormLdapConnection` werden alle Methoden der Superklasse `Connection` überschrieben. Die Methoden `search` und `replace`, welche LDAP-Operationen entsprechen, wurden mit der in diesem Abschnitt beschriebenen Funktionalität zur Umsetzung nach SQL implementiert.

DormLdapConnection.DormLdapConnection

```
1 public DormLdapConnection(BackendDB bdb){
2         super();
3         backend_db=(DormLdapBackendDB) bdb;
4         sql_conn=backend_db.getSqlConnection();
5 }
```

Im Konstruktor der Klasse `DormLdapConnection` wird das zugehörige Objekt `BackendDB` auf den spezialisierten Typ `DormLdapBackendDB` konvertiert und als Membervariable abgelegt. Über diese wird auch auf die im Objekt `DormLdapBackendDB` aufgebaute Datenbankverbindung mittels der Methode `getSqlConnection` zugegriffen und als Membervariable `sql_conn` für weitere Methoden zugänglich gemacht.

6.2.7. Test

Mit der vorgestellten Beispielanwendung wurde das Java-Backend getestet. Dazu simulierten die im OpenLDAP-Projekt vorhandenen LDAP-Client-Tools verschiedene Einsatzsituationen. Darunter z.B. das Aurufen aller LDAP-Operationen, den gleichzeitigen Zugriff mehrerer Clients und auch eine Betriebszeit über mehrere Tage. Für den Test von gleichzeitigen Zugriffen mehrerer Clients wurden spezielle Java-Backend Routinen benutzt, die durch `sleep` Aufrufe künstlich verlangsamt wurden. Keiner dieser Tests führte zu Fehlfunktionen im Java-Backend.

7. Zusammenfassung

Diese Arbeit verfolgte das Ziel, Konzepte zur Verknüpfung des Verzeichnisdienstes LDAP und anderer Datenquellen zu entwickeln und eine Implementierung für den OpenLDAP-Server vorzustellen. Hierzu wurden im theoretischen Teil zunächst die Grundlagen von Verzeichnisdiensten und die verschiedenen Ansätze diskutiert.

In der praktischen Implementierung wurde eine funktionsfähiges Java-Backend entwickelt, mit dessen Hilfe es möglich ist, unterschiedliche Datenquellen und den OpenLDAP-Server über die Programmiersprache Java zu verknüpfen. Dazu wird die objektorientierte Schnittstelle JLdap bereitgestellt. Sie bildet alle gebräuchlichen LDAPv2 Operationen erweitert um Funktionen zur Verwaltung von LDAP-Backends ab. Eine Beispielanwendung zur Anbindung einer SQL-Datenbank diente zum Test der implementierten Schnittstelle. Während der Implementierung traten nur wenige Schwierigkeiten auf, die größte Einschränkung ergab sich aus der Tatsache, daß nur die neusten JVM in der Arbeit eingestzt werden konnten. Die Verwendung von älteren JVMs war nicht möglich, weil ihre Thread Implementierung in Zusammenarbeit mit dem OpenLDAP-Server zu Fehlfunktionen führten.

Abschließend kann gesagt werden, daß das Java-Backend das Einbinden von Datenquellen in den OpenLDAP-Server vereinfacht. Auf Grund der strukturellen Unterschiede zwischen LDAP und verschiedener Datenquellen, z.B. von Relationalen Datenbanken müssen spezielle Lösungen auf diesem Backend aufbauen.

8. Ausblick

Die Verbreitung von LDAP-Servern steht erst am Anfang. So prognostizieren die Marktforscher von IDC, daß im Jahr 2003 über 4 Millionen LDAP-Server im Einsatz sein werden. Vor dem Hintergrund expandierender Intranets und Extranets dürfte kaum ein größeres Unternehmen ohne Verzeichnisdienst auskommen. Ob dabei LDAP-Server eine zentrale Marktposition einnehmen, hängt von vielen Faktoren ab, z.B. von der Standardisierung weiterer Objektklassen und der Synchronisierungs- sowie Replikationsmechanismen. Auch die Verfügbarkeit eines freien LDAPv3-Servers würde einen nicht unerheblichen Faktor bedeuten; das OpenLDAP-Projekt möchte dies mit der Version 2.0 erreichen.

Desweiteren enthält das entwickelte Java-Backend noch Möglichkeiten, die Laufzeit zu optimieren und die Schnittstelle zu verbessern. Beispielsweise könnten alle Java Methoden-IDs zentral beim Starten des Servers ermittelt werden. In der JLdap-Schnittstelle sollten außerdem Objekte, welche zur Zeit vom Java-Typ „String “ sind, durch eine speziellere Klasse dargestellt werden. Zum Beispiel könnte ein DN auch durch eine Klasse `DistinguishedName` repräsentiert sein. Als Vorlage kann hier das „Netscape Directory Service Developer Kit“ [45] dienen. Aber auch die schnelle Weiterentwicklung des OpenLDAP-Servers läßt Vereinfachungen zu. So kann die neueste Entwicklerversion ein Backend über eine neue Verbindung benachrichtigen. Damit wäre der in der Java Shadow-Klasse verwendete Mechanismus zum Erkennen neuer Verbindungen nicht mehr notwendig.

Auch in der Verwendung der Programmiersprache Java liegt noch viel Potential. So ermöglicht die JINI [43] Technologie eine einfache Einbindung vieler Geräte in ein Netzwerk und könnte in Verbindung mit einem Verzeichnisdienst wie LDAP zum Management dieser Geräte einen großen Synergieeffekt erzeugen.

A. JLdap API Referenz

A.1. Übersicht

Dieses Kapitel ist eine Referenz der JLdap Schnittstelle. Sie wurde mit Hilfe des im JDK enthaltenen Programms `javadoc` [44] und die von mir im Java Quellkode eingefügten Kommentare, erstellt. Eine ausführlichere Referenz, in dem auch alle nicht öffentliche Klassen enthalten sind, liegt dem Quellkode bei.

A.2. Package org.openldap.jldap

Interfaces in package:

- None

Classes in package:

A.2.1. org.openldap.jldap.Attribute

java.lang.Object

---

public *Attribute*
extends Object

Represents the name and values of an attribute in an entry.

Constructor Summary

Description
Attribute(java.lang.String attrType) Constructs an attribute with no values.
Attribute(java.lang.String attrType, java.lang.String attrValue) Constructs an attribute that has a single string value.

Method Summary

Returns	Description
public void	**addValue(java.lang.String attrValue)** Adds a value to the attribute.
public String	**getType()** Returns the type of the attribute.
public String	**getValueAt(int index)** Returns the attribute value at the position specified by the index.
public Enumeration	**getValues()** Returns an enumerator for the values of an attribute.
public void	**setType(java.lang.String attrType)** Set the type of the attribute.
public int	**size()** Returns the number of values of the attribute.

A.2.2. org.openldap.jldap.AttributeSet

java.lang.Object

---

public *AttributeSet*
extends Object

Represents a set of attributes.
See Also:

- Attribute (see section A.2.1, page=63)

Constructor Summary

Description
AttributeSet() Constructs a new set of attributes.

Method Summary

Returns	Description
public void	**add(org.openldap.jldap.Attribute attr)** Adds the specified attribute to this attribute set.
public Attribute	**elementAt(int index)** Returns the attribute at the position specified by the index.
public Enumeration	**getAttributes()** Returns an enumeration of the attributes in this attribute set.
public int	**size()** Returns the number of attributes in this set.

A.2.3. org.openldap.jldap.BackendDB

java.lang.Object

---

public *BackendDB*
extends Object

Represents a Backend Database for an OpenLDAP server. Extends your BackendDB class from this.

See Also:

- Connection (see section A.2.4, page=66)

Constructor Summary

Description
BackendDB() Called to initialize each database

Method Summary

Returns	Description
public void	**close()** When OpenLDAP slapd exits normally, it calls a close routine provided by each backend database, allowing the backends to clean up and shut down properly.
public void	**config(int argc, java.lang.String[] argv)** When a configuration option unknown to the OpenLDAP slapd front end is encountered in a database definition in the slapd configuration file, it is parsed and passed to the backend-specific configuration routine for processing.
public void	**open()** Called to open each database, called once after configuration file is read.

A.2.4. org.openldap.jldap.Connection

java.lang.Object

---

public *Connection*
extends Object

Represents a connection to an LDAP server. Extends your Connection class from this.

See Also:

- BackendDB (see section A.2.3, page=65)

Constructor Summary

Description
Connection(org.openldap.jldap.BackendDB bdb, java.lang.Long conn) New LDAP connection.

Method Summary

Returns	Description
public void	**add(org.openldap.jldap.Entry entry)** LDAP add request.
public boolean	**bind(java.lang.String dn, java.lang.String passwd)** LDAP bind request.
public boolean	**compare(java.lang.String dn, org.openldap.jldap.Attribute attr)** LDAP compare request.
public void	**delete(java.lang.String dn)** LDAP delete request.
public void	**modify(java.lang.String dn, org.openldap.jldap.ModificationSet mset)** Modifies an entry in the directory (for example, changes attribute values, adds new attribute values, or removes existing attribute values).
public void	**modrdn(java.lang.String dn, java.lang.String newrdn, boolean deleteoldrdn)** LDAP modrdn request.

Returns	Description
public SearchResults	**search(java.lang.String base, int scope, int sizelimit, int timelimit, java.lang.String filterst)** LDAP search request.
public void	**unbind()** Connection closed.

A.2.5. org.openldap.jldap.Entry

java.lang.Object

---

public *Entry*
extends Object

The Entry class represents an entry in the directory. Objects of this class contain the distinguished name of the entry and the set of attributes in the entry.

Constructor Summary

Description
Entry(java.lang.String dn) Constructs a new LDAP Entry object with the specified information.

Method Summary

Returns	Description
public void	**addAttribute(org.openldap.jldap.Attribute attr)** Add an attribute.
public Attribute	**getAttributeAt(int index)** Gets the Attribute on specified AttributeSet index.
public AttributeSet	**getAttributeSet()** Gets the AttributeSet of the attribute.
public String	**getAttributeTypeAt(int index)** Gets the Type of Attribute on specified AttributeSet index.
public String	**getDN()** Gets the DN of the attribute.
public int	**getSize()** Gets the size of the AttributeSet.

A.2.6. org.openldap.jldap.LDAPException

java.lang.Object

java.lang.Throwable

java.lang.Exception

---

public *LDAPException*
extends Exception

Represent an LDAP Error.

Field Summary

Type	Description
public static final int	**ADMIN_LIMIT_EXCEEDED**
public static final int	**AFFECTS_MULTIPLE_DSAS**
public static final int	**ALIAS_DEREFERENCING_PROBLEM**
public static final int	**ALIAS_PROBLEM**
public static final int	**ATTRIBUTE_OR_VALUE_EXISTS**
public static final int	**AUTH_METHOD_NOT_SUPPORTED**
public static final int	**BUSY**
public static final int	**CLIENT_LOOP**

Type	Description
public static final int	**COMPARE_FALSE**
public static final int	**COMPARE_TRUE**
public static final int	**CONFIDENTIALITY_REQUIRED**
public static final int	**CONNECT_ERROR**
public static final int	**CONSTRAINT_VIOLATION**
public static final int	**CONTROL_NOT_FOUND**
public static final int	**ENTRY_ALREADY_EXISTS**
public static final int	**INAPPROPRIATE_AUTHENTICATION**
public static final int	**INAPPROPRIATE_MATCHING**
public static final int	**INDEX_RANGE_ERROR**
public static final int	**INSUFFICIENT_ACCESS_RIGHTS**
public static final int	**INVALID_ATTRIBUTE_SYNTAX**
public static final int	**INVALID_CREDENTIALS**

Type	Description
public static final int	**INVALID_DN_SYNTAX**
public static final int	**IS_LEAF**
public static final int	**LDAP_NOT_SUPPORTED**
public static final int	**LDAP_PARTIAL_RESULTS**
public static final int	**LOOP_DETECT**
public static final int	**MORE_RESULTS_TO_RETURN**
public static final int	**NAMING_VIOLATION**
public static final int	**NO_RESULTS_RETURNED**
public static final int	**NO_SUCH_ATTRIBUTE**
public static final int	**NO_SUCH_OBJECT**
public static final int	**NOT_ALLOWED_ON_NONLEAF**
public static final int	**NOT_ALLOWED_ON_RDN**
public static final int	**OBJECT_CLASS_MODS_PROHIBITED**

Type	Description
public static final int	**OBJECT_CLASS_VIOLATION**
public static final int	**OPERATION_ERROR**
public static final int	**OTHER**
public static final int	**PARAM_ERROR**
public static final int	**PROTOCOL_ERROR**
public static final int	**REFERRAL**
public static final int	**REFERRAL_LIMIT_EXCEEDED**
public static final int	**SASL_BIND_IN_PROGRESS**
public static final int	**SERVER_DOWN**
public static final int	**SIZE_LIMIT_EXCEEDED**
public static final int	**SORT_CONTROL_MISSING**
public static final int	**STRONG_AUTH_REQUIRED**
public static final int	**SUCCESS**

Type	Description
public static final int	**TIME_LIMIT_EXCEEDED**
public static final int	**UNAVAILABLE**
public static final int	**UNAVAILABLE_CRITICAL_EXTENSION**
public static final int	**UNDEFINED_ATTRIBUTE_TYPE**
public static final int	**UNWILLING_TO_PERFORM**

Constructor Summary

Description
LDAPException() Constructs a default exception with no specific error information.
LDAPException(java.lang.String text) Constructs an LDAPException object with a specified error information.
LDAPException(java.lang.String text, int resultCode)
LDAPException(java.lang.String text, int resultCode, java.lang.String matchedDN)

Method Summary

Returns	Description
public String	**getLDAPErrorMessage()**
public int	**getLDAPResultCode()** Gets specified error information.
public String	**getMatchedDN()**
public String	**toString()**

A.2.7. org.openldap.jldap.Modification

java.lang.Object

---

public *Modification*
extends Object

Specifies changes to be made to the values of an attribute. The change is specified in terms of the following aspects:

- the type of modification (add, replace, or delete the value of an attribute)
- the type of the attribute being modified
- the actual value

Field Summary

Type	Description
public static final int	**ADD** Specifies that a value should be added to an attribute.
public static final int	**DELETE** Specifies that a value should be removed from an attribute.
public static final int	**REPLACE** Specifies that a value should replace the existing value in an attribute.

Constructor Summary

Description
Modification(int op, org.openldap.jldap.Attribute attr) Specifies a modification to be made to an attribute.

Method Summary

Returns	Description
public Attribute	**getAttribute()** Returns the attribute (possibly with values) to be modified.

Returns	Description
public int	**getOp()** Returns the type of modification specified by this object.

A.2.8. org.openldap.jldap.ModificationSet

java.lang.Object

---

public *ModificationSet*
extends Object

Represents a set of modifications to be made to attributes in an entry. A set of modifications is made up of Modification objects.

See Also:

- Modification (see section A.2.7, page=74)

Constructor Summary

Description
ModificationSet() Constructs a new, empty set of modifications.

Method Summary

Returns	Description
public void	**add(int op, org.openldap.jldap.Attribute attr)** Specifies another modification to be added to the set of modifications.
public Modification	**elementAt(int index)** Retrieves a particular Modification object at the position specified by the index.
public int	**size()** Retrieves the number of Modification objects in this set.

A.2.9. org.openldap.jldap.SearchResults

java.lang.Object

---

public *SearchResults*
extends Object

Represents the results of an LDAP search operation.

Constructor Summary

Description
SearchResults() Constructs an new SearchResults Object.

Method Summary

Returns	Description
public void	**addEntry(org.openldap.jldap.Entry entry)** Add an Entry to the Search result.
public Entry	**getEntry(int index)** Gets the Entry on specified SearchResultSet index.
public Vector	**getEntrySet()** Gets all Entries.
public int	**getSize()** Gets size of all Entries.

B. Entwicklungsumgebung

Zur Entwicklung des Java-Backends wurde folgende Hardware eingesetzt:

- Entwicklungsrechner
  Sun UltraSPARC-IIi
  1 CPU 300Mhz, 256 MB RAM, 6GB HD

Als Software kamen folgende Produkte zum Einsatz:

- Betriebssystem
  Sun Solaris 7
- C-Compiler
  Sun Workshop 5.0
- Java Developer Kit
  JDK 1.2.1 Production Release
- LDAP-Server
  OpenLDAP Developer HEAD Branche vom 15.4.1999
- Datenbank
  Sybase Adaptive Server 11.0
- Text-Editor
  VIM - Vi IMproved 5.3
- Weitere GNU-Entwicklungstools
- LaTeXTextsatzsystem für diese Arbeit

C. Anlagen

Dieser Arbeit liegen zwei Disketten mit folgenden Inhalten bei:

Diskette 1	
`/doc/darbeit.ps`	Diese Arbeit im Postscript-Format
`/doc/darbeit.pdf`	Diese Arbeit im PDF-Format
`/doc/java.zip`	Die Referenz der von mir entwickelten Java-Klassen im HTML-Format
`/lib/jldap.jar`	Die kompilierten JLdap Klassen in einem Java-Archive zusammengefasst.
`/lib/dorm.jar`	Die kompilierten Klassen des SQL-Backends in einem Java-Archive zusammengefasst.

Diskette 2	
`/src/openldap.tgz`	Der komplette Quellkode des OpenLDAP-Servers inklusive des Java-Backends
`/src/java.tgz`	Der Quellkode der von mir entwickelten Java-Klassen.
`/src/j-back.tgz`	Der Quellkode des Java-Backends.

D. Selbständigkeitserklärung

Ich erkläre, daß ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Berlin, 13. Juli 1999 Oliver Schönherr

Literaturverzeichnis

[1] *Apache Webserver Homepage*. http://www.apache.org

[2] *OpenSource Homepage*. http://www.opensource.org

[3] *OpenLDAP Homepage*. http://www.openldap.org

[4] *slapd* Manpage slapd Section 8.

[5] *slapd.conf* Manpage slapd.conf Section 5.

[6] *slapd replication log format*. Manpage slapd.replog Section 5

[7] *ldif - LDAP Data Interchange Format*. Manpage ldif Section 5

[8] *Standalone LDAP Update Replication Daemon*. Manpage slurpd Section 8

[9] *LDIF to LDBM database format conversion utilities*. Manpage ldif2ldbm Section 8

[10] Weider, C.; Reynolds, J. *Executive Introduction to Directory Services Using the X.500 Protocol*. RFC 1308, 1992

[11] Weider, C.; Reynolds, J.; Heker, S. *Technical Overview of Directory Services Using the X.500 Protocol*. RFC 1309, 1992

[12] Hardcastle-Kille, S.; Huizer, E.; Cerf, V.; Hobby, R.; Kent, S.*A Strategic Plan for Deploying an Internet X.500 Directory Service*. RFC 11430, 1993

[13] Yeong, W.; Howes, T.; Kille, S. *Lightweight Directory Access Protocol*. RFC 1487, 1995

[14] Howes, T. *A String Representation of LDAP Search Filters*. RFC 1558, 1993

[15] Barker, P; Kille, S.; Lenggenhager, T. *Naming and Structuring Guidelines for X.500 Directory Pilots*. RFC 1617, 1994

[16] Kille, S. *A String Representation of Distinguished Names*. RFC 1779, 1995

[17] Howes, T.; Kille, S.; Yeong, W.; Robbins, C. *The String Representation of Standard Attribute Syntaxes*. RFC 1787, 1995

[18] Howes, T.; Smith, C. *The LDAP Application Program Interface*. RFC 1823, 1995

[19] Howes, T. *A String Representation of LDAP Search Filters*. RFC 1960, 1996

[20] Kille, S.; Wahl, M.; Grimstad, A.; Huber, R.; Sataluri, S. *Using Domains in LDAP/X.500 Distinguished Names*. RFC 2247, 1998

[21] Kille, S.; Wahl, M.; Howes, T. *Lightweight Directory Access Protocol (v3)*. RFC 2251, 1997

[22] Howes, T. *The String Representation of LDAP Search Filters*. RFC 2254, 1997

[23] Howes, T.; Smith, M. *The LDAP URL Format*. RFC 2255, 1997

[24] Howard, L. *An Approach for Using LDAP as a Network Information Service*. RFC 2307, 1998

[25] Good, Gordon *Definition of an Object Class to Hold LDAP Change Records*. INTERNET-DRAFT, 1998

[26] Broy, Manfred; Spaniol, Otto *VDI-Lexikon Informatik und Kommunikationstechnik*. Springer, 1999

[27] Stern, Hal *Managing NFS and NIS*. O'Reilly & Associates Inc., 1991

[28] Kearns, David; Iverson, Brian *The Complete Guide To Novell Directory Service*. Sybex, 1998

[29] *Novell Directory Service*. http://www.novell.com/products/nds/

[30] *Active Directory Features at a Glance*. http://www.microsoft.com/Windows/server/Overview/features/ataglance.asp

[31] Wilcox, Mark *Implementing LDAP*. Wrox, 1999

[32] Jünemann, Markus *LDAP Universalservice*. iX 8/1997, S.118

[33] Lütkebohl, Ingo; Kirsch, Christian *Einsatz eines freien LDAP-Servers*. iX 5/1999, S.155

[34] Wegener, Hans *Definierte Schnittstellen mit JNI - Grenzkontrolle*. iX 7/1998, S.111

[35] Wegener, Hans *Die Arbeit mit JNI - Pendler*. iX 11/1998, S.192

[36] *JDBC Technology*. http://www.javasoft.com/products/jdbc/

[37] *Java Remote Method Invocation*. http://www.javasoft.com/products/jdk/rmi/

[38] *Object Managment Group Home Page*. http://www.omg.org/

[39] *IBM SanFrancisco*. http://www.software.ibm.com/ad/sanfrancisco/

[40] *Extensible Markup Language*. http://www.w3.org/XML/

[41] *Enterprise JavaBeans Technology*. http://www.javasoft.com/products/ejb/

[42] *Java Naming and Directory Interface*. http://www.javasoft.com/products/jndi/

[43] *Jini Connection Technology*. http://www.sun.com/jini/

[44] *Javadoc Home Page*. http://java.sun.com/products/jdk/javadoc/

[45] *Netscape Directory SDK*. http://www.mozilla.org/directory/

[46] Kleinschmidt, P.; Rank C. *Relationale Datenbanksysteme*. Springer, 1997