View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Universität Rostock, Lehrstuhl Datenbank- und Informationssysteme: Dbis Repository

Electronic–Mail–Verwaltung auf objektrelationalen Datenbank–Systemen

Ilvio Bruder, Andreas Heuer[†], Stefan Knorr, Joachim Kröger[†], Astrid Lubinski[†], Thomas Marquardt–Strehlow, Holger Meyer[†], Raiko Nitzsche, Mirko Rzehak, Lars Schneider, Götz Waschk

Universität Rostock
Fachbereich Informatik
Lehrstuhl Datenbank– und Informationssysteme
18051 Rostock

[†] {heuer, jo, lubinski, hme}@informatik.uni-rostock.de
http://wwwdb.informatik.uni-rostock.de/

Zusammenfassung

Electronic–Mails (E–mails) sind semistrukturierte Massendaten, die von herkömmlichen Mail–Systemen (Mail–Servern und Mail–Clients) in einfachen flachen Dateien, sogenannten Mail–Foldern, verwaltet werden. Aufgrund des Fehlens einer dedizierten Zugriffsunterstützung benötigt jeglicher Zugriff auf in Mail–Foldern gespeicherte E–mails dabei ein Parsing des entsprechenden Mail–Folders; dies gilt insbesondere auch für Zugriffe auf strukturierte E–mail–Bestandteile. Bei Nutzung von E–mail–Funktionalität in einer mobilen Umgebung unter Verwendung unterschiedlicher Rechner besteht darüber hinaus im Allgemeinen das Problem, daß E–mails verteilt oder repliziert gespeichert und Informationen über den Bearbeitungs–Status von E–mails auf den verschiedenen Systemen unabhängig voneinander geführt werden. Zumindest diese Status–Informationen können später zumeist nicht mehr integriert werden.

Diese Art der Behandlung von E–mails scheint aus einer datenbankzentrierten Sicht keinesfalls angemessen zu sein und motiviert daher die Konzeption und Implementierung einer E–mail–Verwaltung auf Basis objektrelationaler Datenbank–Technologie.

Dieser Artikel beschreibt den Inhalt und die Ergebnisse einer Pflichtveranstaltung zu “Komplexen Software–Systemen” (KSWS) im Informatik–Hauptstudium, die im Sommersemester 1999 am Fachbereich Informatik der Universität Rostock durch den Lehrstuhl Datenbank– und Informationssysteme durchgeführt wurde, speziell die im Rahmen dieser Lehrveranstaltung als studentische Projekte durchgeführte Implementierung. Das Hauptaugenmerk bei KSWS–Veranstaltungen liegt auf der Vermittlung der Fähigkeit zur Team–Arbeit in einem komplexen Software–Projekt durch eine praktische, industrienahe/–ähnliche Arbeit. Aufgrund des beschränkten Umfangs derartiger Lehrveranstaltungen steht dabei allerdings die tatsächliche prototypische Implementierung — quasi als Machbarkeitsstudie — im Vordergrund und erlaubt etwa die Vernachlässigung der Vollständigkeit oder von Performance–Gesichtspunkten.

1 Einleitung

Ausgehend von der aus Nutzer–Sicht vollkommen unzureichenden Management–Unterstützung bei der Verwaltung von E–mails in herkömmlichen E–mail–Systemen — als Stichpunkte seien

hier das Fehlen adäquater Anfragemechanismen und das Problem der Synchronisation replizierter Datenbestände im mobilen Umfeld genannt — und der Erkenntnis, daß E–mails aufgrund ihres standardisierten Aufbaus sehr wohl einen hohen Strukturierungsgrad aufweisen, entstand die Idee der Konzeption und Implementierung einer E–mail–Verwaltung auf Basis objektrelationaler Datenbank–Technologie (s. hierzu auch die erste Projektskizze in [KH99]).

Nachfolgend beschreiben wir zunächst einige grundlegende Aspekte in eigenen Unterabschnitten, bevor wir in weiteren Abschnitten näher auf die von den drei beteiligten Arbeitsgruppen implementierten Teil–Lösungen eingehen. Im Einzelnen folgen daher ein Architektur–Vorschlag für eine E–mail–Verwaltung in Datenbanken (in Unterabschnitt 1.1), kurze Betrachtungen des Aufbaus von E–mails nach den zugehörigen Internet–Standards (1.2) sowie des IMAP4–Protokolls zum Mail–Zugriff (1.3), das den weiteren Arbeiten zugrundeliegende Datenbank–Schema (1.4) und die tatsächliche Architektur der implementierten Lösung (1.5).

Die umfangreichen Implementierungen umfassen den in Abschnitt 2 beschriebenen E–mail–Parser, den in Abschnitt 3 dargestellten E–mail–Datenbank–Server sowie den E–mail–Datenbank-Client (s. Abschnitt 4). Darüber hinaus wird die Synchronisation von Client und Server gesondert in Abschnitt 5 behandelt, bevor wir in Abschnitt 6 mit einer kurzen Zusammenfassung und einem Ausblick schließen.

Anhang A gibt den Datenbank–Entwurf detailliert wieder; Anhang B enthält ein automatisch generiertes Beispiel–Skript, das eine E–mail in die Datenbank importiert.

1.1 Ein Architektur–Vorschlag für eine E–mail–Verwaltung in Datenbanken

Als Basis für die KSWS–Veranstaltung, deren Beschreibung das Ziel des vorliegenden Artikels ist, diente der Architektur–Vorschlag in Abbildung 1.

Die Grundlage für den Transport von E–mails zwischen verschiedenen Rechnern bilden MTAs (Mail Transport Agents), wie z.B. die Protokolle SMTP (Simple Mail Transfer Protocol — RFC 821) [Pos82] und das (zukünftige) Erweiterungen ermöglichende Protokoll ESMTP (Extended SMTP — RFC 1869) [KFR+95].

Eingehende E–mails sollen im Rahmen der angestrebten Lösung mittels des lokalen Mail–Verteilers (MDA — Mail Delivery Agent; in unserem Falle *procmail* [Aal99]) und Programmen oder Skripten in eine Datenbank auf dem immer verfügbaren Mail–Server des Nutzers (etwa im *Unix*–Home–Verzeichnis) importiert werden. Um konsistent zu anderen Mail–Systemen zu arbeiten und somit den Zugriff auf die in der Datenbank gespeicherten E–mails auch für andere Mail–Clients (MUA — Mail User Agent; z.B. *Netscape mail*, *mutt* oder *pegasus*) in gewohnter Weise zu ermöglichen, ist es notwendig, die E–mails zumindest virtuell in Mail–Foldern zu speichern. Die tatsächliche Art der Speicherung kann vor dem Nutzer und einem von diesem verwendeten MUA dadurch verborgen werden, daß der Mail–Server durch eine Protokoll–Schnittstelle, über die der gesamte Zugriff auf die E–mails erfolgt, vollständig gekapselt wird.

Im Rahmen der angestrebten Lösung haben wir uns auf E–mail–Protokolle und –Standards aus dem Internet konzentriert; andere, proprietäre Verfahrensweisen wurden nicht berücksichtigt. Es wurden insbesondere die beiden zeichenorientierten Protokolle auf TCP–Basis, POP3 (Post Office Protocol 3 — definiert in RFC 1939) [MR96] und IMAP4 (Internet Message Access Protocol 4 — RFC 2060) [Cri96], die für den Zugriff auf Mail–Server Verwendung finden, betrachtet, da beide potentiell in Frage kamen. POP3 ist für kurzzeitige Verbindungen des Clients zum Server gedacht, in deren Rahmen die E–mails auf den Client "heruntergeladen" werden, die in dem dortigen Eingangs–Mail–Folder liegen. Die einzige mögliche Manipulation auf dem Server umfaßt bei POP3 das Löschen von E–mails; die eigentliche Verwaltung der E–mails aller

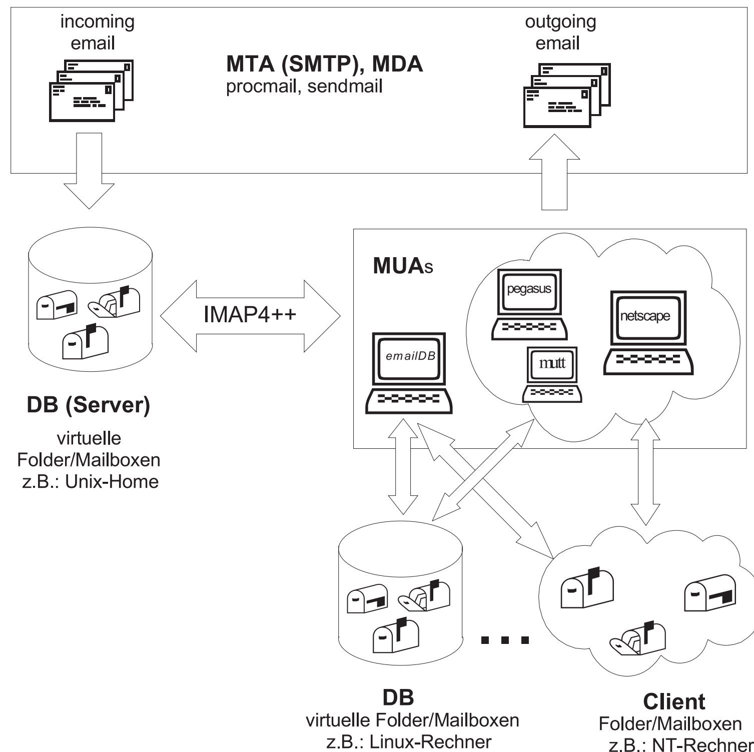

Abbildung 1: Architektur–Vorschlag für eine E–mail–Verwaltung in Datenbanken

Nutzer muß jedoch auf deren Clients durchgeführt werden. Ein sinnvolles Arbeiten ist in diesem Umfeld eigentlich nur auf je einem einzigen Client pro Nutzer möglich. IMAP4 erlaubt hingegen die Nutzung verschiedener Clients, da die E–mails grundsätzlich auf dem Server verbleiben und dort sowohl bearbeitet, als auch in beliebigen Mail–Foldern verwaltet werden können. Der Wunsch, unsere Lösung auch in einem mobilen Umfeld verwenden zu können, führte zur Wahl von IMAP4 als zu verwendendes E–mail–Protokoll.

Die zuvor aufgeführten MUAs sind allesamt IMAP4–fähig, basieren also selbst auf diesem Protokoll, so daß ihrer Verwendung in Kombination mit dem Mail–Server auf Datenbank–Basis nichts entgegensteht. Der Zugriff durch die MUAs auf die Datenbank geschieht daher wie auf jeden anderen IMAP4–fähigen Mail–Server. Der Server muß dazu natürlich vollständig IMAP4–kompatibel sein, um alle IMAP4–fähigen Clients unterstützen zu können. Ein Nutzer kann somit von einem beliebigen Rechner mit Internet–Anbindung mittels IMAP4 auf die gespeicherten E–mails auf dem Server bzw. in der Datenbank zugreifen. “IMAP4++” steht in Abbildung 1 für erweiterte nützliche Funktionalitäten des Protokolls für Server und Client (z.B. können Datenbank–Funktionalität oder Replikationstechniken [SH99] im IMAP4++–Protokoll eingebettet werden). Die MUAs speichern die vom Server übertragenen E–mails lokal auf jedem Client (z.B. *WindowsNT*–Rechnern) wie gewohnt in Mail–Foldern ab. Ein im Rahmen der angestrebten Lösung zu implementierender Mail–Client (hier als “KSWS” bezeichnet) soll darüber hinaus die Vorteile der Fähigkeit zur Speicherung von E–mails in einer Datenbank ebenfalls für seine lokale Datenhaltung ausnutzen und die erweiterte Funktionalität von IMAP4++ unterstützen.

Da IMAP4 keine eigene Schnittstelle zum Versenden von E–mails beinhaltet, verlassen auch wir uns in der angestrebten Lösung auf die Verwendung von SMTP, insbesondere die unter *Unix* gebräuchlichste Variante *sendmail*.

Zur Unterstützung der mobilen Verwendung von E–mail–Funktionalität aus unterschiedlichen Umgebungen wird für die angestrebte Lösung ein geeigneter Replikations-/Synchronisationsmechanismus benötigt, der eine Integration verteilt gespeicherter, replizierter Information erlaubt.

Als Ausgangsbasis für die Implementierung der mit dem Architektur–Vorschlag verbundenen Komponenten wurde die Verwendung vorhandener, frei verfügbarer Bibliotheken für das IMAP4–Protokoll angestrebt.

1.2 Der Aufbau von E–mails nach Internet–Standards

Das Internet Standard Mail Format (RFC 822) [Cro82] und die in diesem Zusammenhang einsetzbaren MIME–Erweiterungen (Multipurpose Internet Mail Extension — RFCs 2045–2049) [FB96a, FB96b, Moo96, FKP96, FB96c] spezifizieren den Aufbau/die Struktur von E–mails. E–mails bestehen demnach aus einem Envelope (d.h. Briefumschlag), der den Postweg beschreibt und von den beteiligten MTAs erzeugt wird, einem Header (Briefkopf), der vom MUA des Erzeugers der E–mail bzw. dem ersten mit der Übertragung betrauten MTA erzeugt wird, und einem Body, der den eigentlichen E–mail–Inhalt umfaßt.

Der Envelope besteht aus vier Attributen/Attributarten, die als Attribut–Wert–Paare notiert werden. Auf eine genauere Beschreibung dieser Attribute wird an dieser Stelle verzichtet, da sie für die angestrebte Lösung bedeutungslos ist. Die Envelope–Attribute umfassen “From”, “Received:”, “Return-Path:” und “Resent-”header.

Der Header einer E–mail besteht ebenfalls aus Attribut–Wert–Paaren, wobei sich die Bedeutung der einzelnen Attribute hinreichend genau aus ihrem Namen ableiten läßt: “Message-ID:”, “From:”, “To:”, “Subject:”, “Date:”, “Reply-To:”, “Cc:”, “Bcc:”, “Fcc:”, “Sender:”, “Priority:”, “Precedence:”, “Return-Receipt-To:”, “Lines:”, “Status:” und “X-”beliebige Erweiterungen.

Die Grenze zwischen Header und Body einer E–mail wird durch eine Leerzeile markiert.

Der Body einer E–mail ist zunächst nach RFC 822 ein beliebiger Text in 7bit–US–ASCII–Zeichen. Durch die MIME–Erweiterungen wird eine Kodierung von Nicht–ASCII–Daten mittels “Base64” definiert, die somit eine Übertragung beliebiger Daten erlaubt. MIME definiert sechs zusätzliche Header–Attributarten: “MIME-Version:”, “Content-Type:”, “Content-Transfer-Encoding:”, “Content-Length:”, “Content-ID:” und “Content-”beliebige Erweiterungen. Dadurch wird zugleich eine Typisierung und Strukturierung von E–mail–Bodies eingeführt: ein einfacher, MIME–konformer E–mail–Body kann vom Typ “text”, “image”, “audio” oder “video” sein. Ist der Body vom Typ “multipart”, so kann er rekursiv aus mehreren MIME–Bodies bestehen; “application” bezeichnet eine erweiterbare Klasse von Typen, die anwendungsspezifische Daten kennzeichnen, während durch “message” die Kennzeichnung von RFC 822–Bodies, Teilen davon oder Verweisen auf im Dateisystem verfügbare Daten, und damit die Übertragung überlanger E–mails ermöglicht wird.

Insgesamt bestehen E–mails also aus einer Menge attributierter Daten und einer Liste von Bodies höchst unterschiedlicher Inhalte und sind daher als “semistrukturierte Daten” zu bezeichnen. Für die Speicherung von E–mails sind daher objektrelationale Datenbankmanagementsysteme (DBMS) eine geeignete Plattform. Auf den sich aus dem Aufbau von E–mails ergebenden Datenbank–Entwurf gehen wir in Unterabschnitt 1.4 näher ein.

1.3 Das IMAP4–Protokoll zum Mail–Zugriff

IMAP4 [Cri96] unterscheidet drei Arbeitsmodi: "offline", "online" und "disconnected". Im "offline"–Modus verhält sich IMAP4 wie POP3: der Nutzer/Client lädt seine E–mails vom Server und beendet die Verbindung anschließend, um die E–mails lokal zu verwalten. Im "online"–Modus arbeitet der Nutzer direkt auf dem Datenbestand des Servers. Der für das mobile Umfeld und auch für die von uns angestrebte Lösung interessante Fall ist der "disconnected"–Modus: nach einem verbindungslosen Arbeiten mit lokal auf dem Client gespeicherten Kopien der E–mails (Replikaten) nimmt der Nutzer wieder Kontakt mit dem Server auf und synchronisiert die Datenbestände. Die Vor– und Nachteile der drei Modi sind in [Gra95a, Gra95b] ausführlich beschrieben.

Charakteristische Eigenschaften von IMAP4 sind die Möglichkeit des "remote" (entfernten) E–mail–Zugriffs und der Manipulation/Verwaltung von E–mails auf dem Server. Dadurch kann der Nutzer auf dem Server arbeiten, als seien die Mail–Folder lokal auf seinem Client vorhanden. Desweiteren geschieht die Datenübertragung bei IMAP4 "on demand", d.h. der Client entscheidet, welche Daten er benötigt; dies schließt den expliziten Verzicht auf die Notwendigkeit der Übertragung kompletter Mail–Folder und kompletter E–mails ein.

Durch den selektiven Zugriff auf Daten (E–mails) wird die Nutzung von E–mail-Funktionalität somit auch im Falle des Vorliegens großer Mail–Folder und kleiner Übertragungsbandbreiten bei gleichzeitig hohen Verbindungskosten, dem typischen mobilen Szenario, möglich. Die Grundlage für die selektive Übertragung von (Teil–)Daten schafft ein serverbasiertes RFC 822– und MIME–Parsing, das den Aufbau einer E–mail nach Unterabschnitt 1.2 analysiert.

Das Protokoll IMAP4 ist zustandsbasiert. Im Rahmen unserer Lösung sind lediglich die beiden Zustände, die das Arbeiten mit Mail–Foldern bzw. E–mails gestatten, von Interesse. Die anderen beiden Zustände umfassen die An– und Abmeldung.

Im "Authenticated state" ist es möglich, Mail–Folder anzulegen, umzubenennen und zu löschen, zu testen, ob neue E–mail in einem Mail–Folder vorliegen, (neue) E–mails an einen Mail–Folder anzufügen, sowie einen Mail–Folder als aktiv auszuwählen.

Nach Auswahl eines Mail–Folders befindet sich das Protokoll im "Selected state", der das Arbeiten auf den E–mails des ausgewählten Mail–Folders erlaubt: so können E–mails gelöscht oder in andere Mail–Folder kopiert oder verschoben, Flags, die den aktuellen Bearbeitungs–Status jeder E–mail anzeigen, gesetzt und gelöscht, E–mails oder Teile von E–mails selektiv übertragen und auf dem E–mail–Bestand des Mail–Folders nach unterschiedlichen Kriterien gesucht werden. Auch die Suche auf E–mails wird durch das serverbasierte RFC 822– und MIME–Parsing ermöglicht.

Die Identifikation von E–mails geschieht in IMAP4 durch zwei unterschiedliche Konzepte:

1. Sequenz–Nummern werden den E–mails eines Mail–Folders zugeordnet, sobald dieser Mail–Folder als aktiver Mail–Folder ausgewählt wurde.

   Sequenz–Nummern sind nicht persistent, weshalb es auch keinen Sinn macht, sie in der Datenbank abzulegen; ihre Gültigkeit ist nicht nur auf eine Sitzung (Verbindung) beschränkt, sondern sogar auf den Zeitraum, in dem sich der Datenbestand des Mail–Folders nicht ändert. Sobald eine E–mail endgültig aus dem Mail–Folder entfernt oder an den Mail–Folder angefügt wird, können sich die Sequenz–Nummern der E–mails ändern.

   Die Abbildung von Sequenz–Nummern auf die unter 2.) genannten UIDs wird von der Oberfläche des Clients (bei Verwendung des offline oder disconnected–Modus) bzw. im

Rahmen der Protokoll–Auswertung vom Server übernommen (bei Arbeiten im online–Modus). Die Abbildung ist denkbar einfach: Sequenz–Nummern dienen dem einfachen Durchnumerieren der E–mails des Mail–Folders auf Basis der UIDs der E–mails. Da UIDs längerfristig vergeben werden (persistent sind), kann ihre Folge "Lücken" aufweisen. Sequenz-Nummern hingegen stellen zu jedem beliebigen Zeitpunkt eine ununterbrochene Folge von Zahlen dar, die bei Eins startet und bei <Anzahl E–mails im Mail–Folder> endet. Wir vertiefen die Betrachtung von Sequenz-Nummern an dieser Stelle nicht weiter.

2. Der UID (Unique IDentifier) wird vom IMAP4–Protokoll als eindeutiger Bezeichner einer E–mail innerhalb eines Mail–Folders definiert.

   Jede E–mail behält ihre UID solange, bis sie oder der Mail–Folder gelöscht wird — oder aber der Mail–Folder umstrukturiert, z.B. sortiert, wird. UIDs dürfen sich nicht während einer Sitzung ändern und sollten zwischen Sitzungen unverändert bleiben. Ist es unmöglich, die UIDs aus einer Sitzung für eine nachfolgende Sitzung zu erhalten, so muß dies durch eine Erhöhung eines persistenten Zählers "Unique IDentifier VALIDITY", der jedem Mail–Folder zugeordnet ist, angezeigt werden.

Insgesamt eignet sich IMAP4 hervorragend als Basis für die angestrebte Datenbank–Lösung, da es die Möglichkeit der Kapselung der zur Speicherung der E–mails verwendeten Datenbank bietet (IMAP4 macht als Zugriffs–Protokoll keinerlei Aussagen zur durchzuführenden Speicherung), dabei durch seine Funktionalität ein mobiles Anwendungs–Szenario ermöglicht und gleichzeitig bereits von einer Vielzahl von MUAs unterstützt wird.

Die Stärken von IMAP4 können zugleich von der Datenbank–Lösung profitieren:

- Durch die Speicherung von E–mails in einer Datenbank wird es möglich, die Anfrage– und Optimierungs–Funktionalität des verwendeten DBMS beispielsweise für die Such–Operationen von IMAP4 zu nutzen.
- Das zur selektiven Übertragung von E–mails notwendige Parsing von E–mails bzw. Mail–Foldern kann in seiner Anzahl begrenzt werden. Alle E–mails müssen nur genau einmal, vor ihrem Import in die Datenbank, syntaktisch analysiert werden. Bei IMAP4–Anforderungen von E–mail(–Bestandteilen) ist dieser Schritt der Verarbeitung daher bereits immer durchgeführt.

Auch die Verwendung von Mail–Identifikatoren (UIDs) kann durch ein DBMS für einen effizienten Zugriff sinnvoll ausgenutzt werden. Darüber hinaus kann durch zusätzliche Indexe auf ausgewählten Attributen der Mail–Zugriff weiter verbessert werden.

1.4 Das zugrundeliegende Datenbank–Schema

Zur Implementierung der E–mail–Verwaltung wurde das objektrelationale DBMS *DB2* von IBM in der vorliegenden Version *DB2 UDB 5.2.0 (Fixpack 6 – Sep. 98)* ausgewählt. Es wurden möglichst die spezifischen Features von *DB2* genutzt, als Nachschlagewerk diente [Cha98]. Für weitere Informationen stehen außerdem die Online–Handbücher der *DB2*–Installation zur Verfügung [IBM98].

Das verwendete Datenbank–Schema (für eine detaillierte Erläuterung s. auch Anhang A) ist zum Teil aus Tests mit Vorversionen der hier beschriebenen Programme hervorgegangen, d.h. aufgrund praktischer Erfahrungen angepaßt worden.

Betrachtungen zu *Informix* als weitere Datenbank–Plattform werden in einem Nachfolge–Projekt angestellt.

Das zugrundeliegende Datenbank–Schema ist in Abbildung 11 auf Seite 39 dargestellt und wird in Anhang A ausführlich erläutert. An dieser Stelle wird daher nur ein kurzer Überblick gegeben.

Es wird für jeden Nutzer des Mail–Servers ein eigenes Datenbank–Schema erzeugt. Dies ergab sich als Konsequenz aus Effizienz– und Sicherheits–Überlegungen. Gesetzt den Fall, die E–mails aller Nutzer würden in einer Tabelle aufgenommen werden, dann wäre eine Selektion in dieser Tabelle — bei einer großen Nutzeranzahl — aufgrund der Datenbank–Größe relativ langsam. Ein gutes Sicherheitskonzept ist natürlich auch mit einer Tabelle möglich, aber zum einen ist die Nutzung voneinander unabhängiger Tabellen einfacher zu realisieren und zum anderen ist das subjektive Sicherheitsgefühl bei einer exklusiven Nutzung eines Datenbank–Schemas größer.

Die Grundanforderung an die Datenbank–Lösung besteht darin, jede mögliche E–mail vollkommen verlustfrei in der Datenbank zu speichern. Da es nach Unterabschnitt 1.2 optionale und unbestimmte E–mail–Bestandteile gibt, muß dafür Sorge getragen werden, daß auch diese Teile in die Datenbank aufgenommen werden können. Es gibt daher für die im Allgemeinen weniger nachgefragten/verwendeten Header–Attribute eine generische Speicherungsmöglichkeit. Nicht identifizierbare E–mail–Body–Typen werden als "anwendungsspezifisch" verstanden und entsprechend behandelt. Als Gegenleistung sichert das DBMS die Verfügbarkeit und transaktionale Sicherheit bei der Speicherung der E–mails zu.

Damit E–mails gemäß dem Datenbank–Schema zerlegt gespeichert werden können, müssen die einzelnen Mail–Teile über OID–Referenzen (ObjectIDentifier; zur eindeutigen Auszeichnung von Objekten) miteinander in Beziehung gesetzt werden. OIDs können bei *DB2* (in der verwendeten Version) nur "nutzergeneriert" erzeugt werden. Da wir OIDs benötigen, muß dieser Umstand bei der Handhabung der Objekte in den unterschiedlichen Programmen beachtet werden. Um die Eindeutigkeit der OIDs nicht bei deren Generierung zu verletzen, existiert eine "next_oid"–Tabelle zur Festlegung des nächsten zu vergebenden OID–Wertes für jede Tabelle.

Die Tabelle "mime" bildet den Obertyp aller Mail–Teile und kann auch selbst Instanzen hervorbringen, z.B. sind die rekursiv definierten "multipart"–Bodies wieder vom Typ "mime". Die Tabellen "mail", "text", "appl" (für "application"), "image", "audio" und "video" dienen der Aufnahme der entsprechenden MIME–Bodies (vgl. Unterabschnitt 1.2), bzw. auch der Header–Attribute ("mail" wurde als Untertyp von "mime" definiert, da MIME–Bodies vom Typ "message" wiederum komplette E–mails sein können).

Neben den E–Mails selbst müssen natürlich auch die Mail–Folder (Tabelle "mailfolder") und auch die Zuordnung der E–mails zu den einzelnen Mail–Foldern ("foldercontent") gespeichert werden.

Aufgrund eines Bugs im IMPORT–Tool des *DB2*–Interpreters benötigen wir außerdem eine Tabelle "import". Alle für die Synchronisation benötigten Protokoll–Daten werden in der Tabelle "logtable" abgelegt.

Ein Problem bei der Replikation ist das Laden der Bodies von E–mails, die nach dem Einfügen in einen Mail–Folder in einen anderen Mail–Folder verschoben wurden. Wenn ein Client im Rahmen der inkrementellen Synchronisation die Liste der protokollierten Log–Einträge unter ausschließlicher Verwendung von IMAP4–Kommandos abarbeitet und einen "APPEND"–Eintrag (der das Anfügen einer E–mail an einen Mail–Folder signalisiert) findet, so kann er nicht sicher sein, daß die eingefügte E–mail immer noch in dem angegebenen Mail–Folder vorliegt. Der Client müßte daher zuerst alle anderen Log–Einträge daraufhin kontrollieren, ob die E–mail in einen anderen Mail–Folder kopiert oder verschoben wurde, bevor die E–mail vom Server angefordert

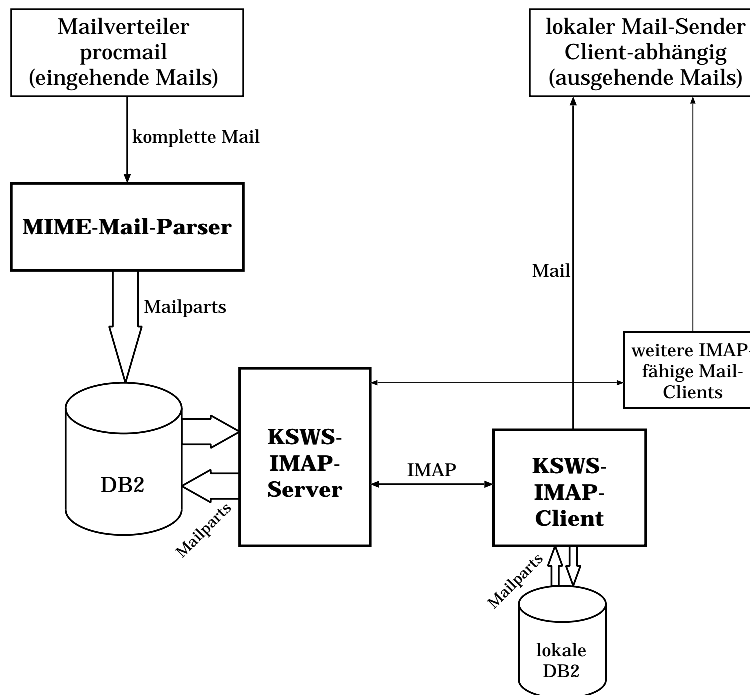

Abbildung 2: Architekturskizze der implementierten Lösung

werden könnte. Die Übertragung jeder E–mail vom Server zum Client geschieht dadurch, daß der Client den gewünschten Mail–Folder auswählt, in der die E–mail zur Zeit ist, und dort ein "FETCH"–Kommando (IMAP4–Kommando zur Spezifikation der Übertragung einer E–mail) ausführt. Um diese ganze Prozedur abzukürzen/zu vereinfachen, führt der Server einen virtuellen Mail–Folder "rootmail", der sämtliche E–mails aller Mail–Folder enthält. Diese Sicht dient somit vor allem dazu, E–mails einfach übertragen zu können.

1.5 Die Architektur der implementierten Lösung

Die Architektur der implementierten Lösung weicht etwas vom allgemeinen Architektur-Vorschlag für eine E–mail–Verwaltung in Datenbanken in Unterabschnitt 1.1 ab. Die Architekturskizze gemäß Abbildung 2 beschreibt die tatsächlich implementierte Lösung, die eine funktional auf das Wesentliche beschränkte, lauffähige Architektur darstellt.

Die Architekturskizze zeigt den momentanen Stand der Implementierung nach der ersten Projektphase. *procmail* ist der Mail–Verteiler, der am Fachbereich Informatik der Universität Rostock für die Verteilung der E–mails an die Nutzer eingesetzt wird. Das DBMS *DB2* wird für die Verwaltung der E–mails sowohl beim Server als auch beim Client eingesetzt. Das Senden von E–mails geschieht über den lokalen Mail–Sender (meist *sendmail*). Die in der Architekturskizze hervorgehobenen Teile werden nachfolgend in eigenen Abschnitten beschrieben: der MIME–Mail–Parser in Abschnitt 2, der KSWS–IMAP–Server in Abschnitt 3 sowie der KSWS–IMAP–Client in Abschnitt 4.

Das Problem, ausgehende E–mails zu erfassen, die vom Client nicht an den Server, sondern direkt an den lokalen MTA (*sendmail*) übergeben werden, kann dadurch gelöst werden, daß auf jedem Client ein "Sent"–Folder existiert, in dem Kopien der verschickten E–mails gespei-

chert werden. Im Rahmen der Synchronisation können diese E–mails dann auch auf den Server übertragen werden.

Die Verbindungen zwischen den beteiligten Komponenten beschreiben die Art und den Weg der E–mails vom abschickenden Nutzer zum empfangenden Nutzer durch das System. Ein– und ausgehende E–mails sind immer vollständige E–mails nach RFC 822 [Cro82], während der E–mail–Parser die E–mails in Teile zerlegt, die gespeichert und zwischen Server und Client mittels IMAP4 übertragen werden. Der entwickelte Server wurde darüber hinaus auch bereits mit anderen IMAP4–fähigen Clients erfolgreich getestet.

2 Der E–mail–Parser

Die Hauptaufgabe des E–mail–Parsers liegt darin, den Weg der E–mails in die Datenbank zu beschreiben und zu implementieren. Die Implementierung ist also dafür zuständig, die Datenbank mit Daten zu füllen. Gefüllt wird die Datenbank einmal aus Richtung der mittels *procmail* eingehenden E–mails und zum anderen durch E–mails, die vom E–mail–Datenbank–Server im Synchronisationsfall mittels des IMAP4–Befehls "APPEND" in die Datenbank geschrieben werden.

Ein weiteres Einsatzgebiet der Implementierung des E–mail–Parsers ist gemäß der Architektur aus Abbildung 2 das Füllen einer lokalen Datenbank auf Client–Seite. Auch hier müssen eingehende und per "APPEND"–Befehl erzeugte E–mails in die Datenbank eingefügt werden.

Diese Aufgaben können konzeptionell gut zusammengefaßt werden. Wir betrachten deshalb in den weiteren Ausführungen vorrangig den Einsatz der Software beim Füllen der Datenbank aus Richtung des Mail–Verteilers *procmail*.

Die vollständige Beschreibung der Implementierung des E–mail–Parsers findet sich in [BS99].

In den folgenden Unterabschnitten wird zunächst kurz eine Problemanalyse und Konzeption des E–mail–Parsers durchgeführt (2.1), bevor ausführlich auf die Implementierung und Nutzung dieses E–mail–Parsers eingegangen wird (2.2). Der Abschnitt schließt in Unterabschnitt 2.3 mit einer Beschreibung inhärenter Probleme der Lösung sowie von Problemen, die durch die verwendete Software zu verantworten sind, und einem Ausblick.

2.1 Problemanalyse und Konzeption

Aus den zuvor beschriebenen Aufgaben des E–mail–Parsers und der Einordnung der Aufgabe in die Architekturskizze in Abbildung 2 ergibt sich für den E–mail–Parser die in Abbildung 3 dargestellte Architektur, die nachfolgend genauer beschrieben wird.

Ausgangspunkte für die Entwicklung des Parsers sind zum Einen der lokale Mail–Verteiler (MDA; *procmail*) und seine Funktionen zur Weitergabe von E–mails und zum Anderen das Datenbank–Schema.

2.1.1 Das Szenario

Abbildung 3 zeigt die lokale Architektur für die Aufbereitung und den Transport der E–mails vom Mail–Verteiler bis in die Datenbank.
Das Szenario kann folgendermaßen beschrieben werden:

Der MDA empfängt eine E–mail und leitet diese an den betreffenden Nutzer, bzw. eine Datei, auf die der Nutzer Zugriff hat, weiter. An dieser Stelle werden die Einstellungen des Mail–Verteilers für den Nutzer so konfiguriert, daß die E–mails nicht wie üblich in einen

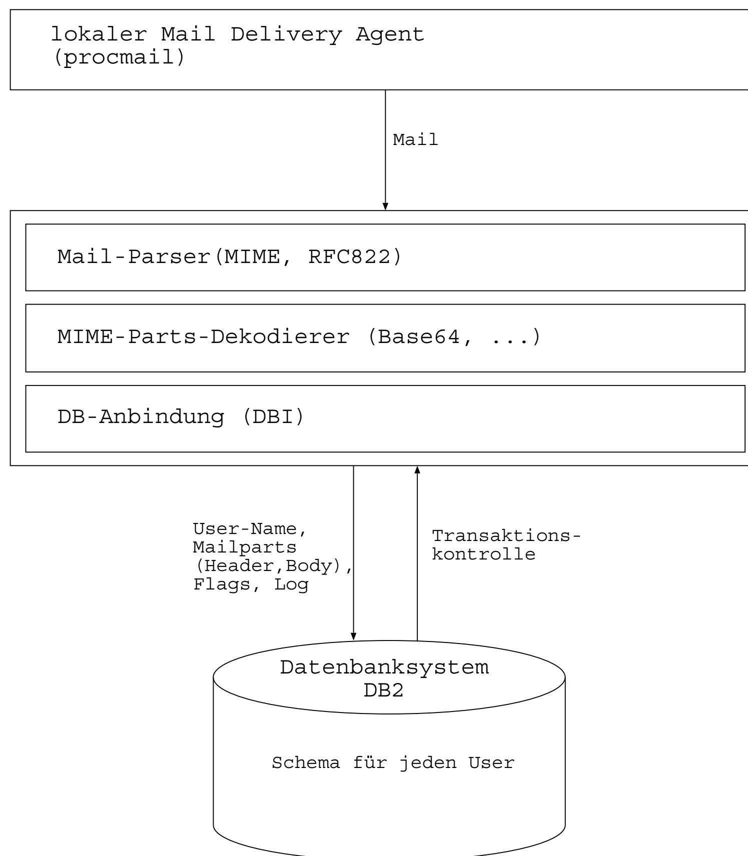

Abbildung 3: Architekturskizze des Parsing–Vorgangs

Mail–Folder geschrieben werden, sondern an ein Programm übergeben werden, welches die E–Mails weiterverarbeitet. Dieses Programm muß in erster Linie jede E–mail syntaktisch analysieren und sie so aufbereiten, daß sie in die Datenbank–Struktur paßt. Dazu müssen die Header–Informationen, der RFC 822–Mail–Body [Cro82] und eventuell vorliegende Attachments (MIME–Bodies, [FB96a]) in Datenbank–konforme Attribut–Wert–Paare umgewandelt werden. Bei Attachments ist es zusätzlich möglich, die kodierten Informationen (kodiert wegen der Kompatibilität zum RFC 822–Standard) zu dekodieren, um dadurch den Zugriff auf die Inhalte der kodierten Mail–Teile zu unterstützen.

2.1.2 Der Mail–Verteiler *procmail*

procmail als Mail–Verteiler am Fachbereich Informatik der Universität Rostock kann für jeden Nutzer lokal eingerichtet werden. Dazu muß eine Datei `.procmailrc` im Home–Verzeichnis des Nutzers angelegt werden. Hier gibt es dann eine Vielzahl von Einstellungsmöglichkeiten, wie in [Aal99] beschrieben, von denen wir nur einige wenige benötigen werden.

Was muß *procmail* leisten? Als erstes muß von *procmail* für jede eingehende E–mail ein Programm gestartet werden, dem die eingegangene E–mail übergeben wird. Weiterhin sollte es möglich sein, eine Warteschlange für eingehende E–mails aufzubauen. Dies ist nötig, da ansonsten bei sehr großen Mengen von E–mails Probleme mit Effizienz und Verfügbarkeit der E–mails

riskiert würden. Ein solches Verhalten von *procmail* wird erreicht, indem die Datei `.procmailrc` mit der Option `w` eingestellt wird, so daß *procmail* auf das Ende des aufgerufenen Programms wartet. Mit der zusätzlichen Angabe von `: local.lock` wird eine temporäre Sperre erzeugt, die weitere *procmail*–Aufrufe verhindert. Ist das Programm abgearbeitet und *procmail* fertig, so wird die Sperre wieder entfernt und die nächsten *procmail*–Aufrufe können anlaufen.

2.2 Implementierung und Nutzung

In diesem Unterabschnitt soll dargestellt werden, wie die zuvor beschriebene Aufgabe implementationstechnisch umgesetzt wurde. Dazu wurde im Wesentlichen *Perl* (s. [SSP99, Sri97]) zur syntaktischen Analyse der E–mails, der *DB2*–Kommandozeileninterpreter (CLI) zum Ansprechen der Datenbank und ein Shell–Skript zur Ausführung der Datenbank–Befehle verwendet.

2.2.1 Parsing mit *Perl*

Jede E–mail wird über die Standard–Eingabe an das Parser–Programm übergeben. Es ist möglich, als Parameter den Mail–Folder anzugeben, in dem die E–mail eingetragen werden soll. Wird kein Parameter angegeben, so wird die E–mail per default in den nach [Cri96] für IMAP4 geforderten INBOX–Folder eingetragen.

Die syntaktische Analyse von E–mails ist in *Perl* relativ simpel, da es *Perl*–Module bzw. –Bibliotheken gibt, die ein Mail–Parsing unterstützen. Es wurde daher ein solches Modul zur syntaktischen Analyse von MIME–Mails benutzt [Ery99b]. Für diese Bibliothek benötigt man noch die Module `MailTools` [Bar98] für Basis–Mails, `IO-stringy` [Ery99a] zur Unterstützung von Input und Output und `MIME-Base64` [Aas98] zur Kodierung und Dekodierung binärer Attachments. An dieser Stelle muß gesagt werden, daß der Parser das Dekodieren von Binär–Teilen von E–mails im Grunde beherrscht. Da das Gesamt–System aber aus Vereinfachungsgründen diese Funktionalität momentan nicht unterstützen kann, mußte der Parser gezwungen werden, E–mails nicht automatisch zu dekodieren. Dazu bedurfte es eines kleinen Eingriffes in die Bibliothek `ParserBase.pm`. Dort wurde die Methode `parse_data` angewiesen, keine Dekodierung durchzuführen.

Das *Perl*–Programm `mimeparser` (s. [KSW99]) nutzt nun die genannten Bibliotheken, um E–mails in einzelne Bestandteile zu zerlegen und diese dann in SQL–Befehle einzubetten. Die SQL–Befehle werden danach als *DB2*–Interpreter–Aufrufe in ein Shell–Skript geschrieben (ein Beispiel–Skript findet sich in Anhang B). Dieses Skript ist nötig, da nicht davon ausgegangen werden kann, auf dem Rechner, auf dem der Mail–Verteiler arbeitet und auf welchem damit auch der Parser gestartet wird, einen Datenbank–Client zur Verfügung zu haben. Aus diesem Grund wird das Shell–Skript dynamisch generiert und per Remote–Shell (Möglichkeit, auf entfernten Rechnern Programme auszuführen) auf einem Rechner mit Datenbank–Client (in der aktuellen Infrastruktur des Lehrstuhls Datenbank– und Informationssysteme des Fachbereichs Informatik der Universität Rostock der Rechner `hades`) gestartet. Damit bei dem Remote–Shell–Aufruf keine Passwort–Abfrage erfolgt, muß im Home–Verzeichnis des Nutzers eine `.rhosts`–Datei existieren, in der die Rechner eingetragen sind, die an der Remote–Verbindung beteiligt sind.

Als erstes wird die E–mail im Programm durch Aufruf entsprechender Funktionen der Bibliotheken syntaktisch analysiert. Dabei erhält man ein Objekt "Mail" und kann auf die Bestandteile zugreifen, indem man bestimmte Methoden aufruft:

- Zunächst werden die Header–Informationen extrahiert und in Variablen gespeichert. Es werden natürlich nur solche Header–Informationen betrachtet, die auch in der Datenbank

definiert sind. Es gibt nun Header–Informationen, die problemlos in die Datenbank eingefügt werden können. Das sind diejenigen, die nicht als Character Large Object (CLOB) in der Datenbank deklariert sind und keine nicht–konformen Zeichen (`'` und `"` können vom *DB2*–Interpreter nicht korrekt verarbeitet werden) beinhalten. Diese Attribute werden aufbereitet (das letzte "new_line"–Zeichen wird entfernt) und in einer Attributliste gesammelt. Die gesammelten Attribute werden dann aus dem Header gelöscht.

- Im Datenbank–Schema vorgesehene Header–Attribute, die in der E–mail nicht belegt waren, werden mit definierten Nullwerten belegt. Dies ist notwendig, da es sonst zu Inkompatibilitäten mit dem E–mail–Datenbank–Server (s. Abschnitt 3) und dem E–mail–Datenbank–Client (s. Abschnitt 4) kommt, die Probleme mit undefinierten Nullwerten haben. Zwei sehr wichtige Attribute sind der sogenannte "Content-Type" und die Statusflags. Aus dem "Content-Type" wird der Objekt–Typ und das zugehörige OID–Präfix des jeweiligen Mail–Teiles ermittelt. In der ersten Hierarchie–Ebene wird unabhängig vom angegebenen "Content-Type" immer der Objekttyp "mail" gebildet. Folgende Zuordnung wurde getroffen (Angaben des "Content-Type"–Feldes als reguläre Ausdrücke):

Content-Type		Objekttyp	OID--Präfix
^([Mm][Ee][Ss][Ss][Aa][Gg][Ee])	→	Mail	'rfc'
^([Tt][Ee][Xx][Tt])	→	Text	'txt'
^([Ii][Mm][Aa][Gg][Ee])	→	Image	'img'
^([Aa][Uu][Dd][Ii][Oo])	→	Audio	'aud'
^([Vv][Ii][Dd][Ee][Oo])	→	Video	'vid'
^([Aa][Pp][Pp][Ll][Ii][Cc][Aa][Tt][Ii][Oo][Nn])	→	Appl	'app'
^([Mm][Uu][Ll][Tt][Ii][Pp][Aa][Rr][Tt])	→	Mime	'mim'
sonst	→	Appl	'app'

  In der letzten Zeile werden die nicht–standardisierten Typen aufgefangen und allgemein auf den "Application"–Typ abgebildet. Dadurch können alle E–mails verlustfrei in der Datenbank gespeichert werden. Das zweite interessante Mail–Attribut ist das "Statusflag". Wenn ein solches "Statusflag" definiert ist und einen Wert enthält, wird es in dieser Form als Mailflag in die Datenbank aufgenommen. Wenn kein solches Mail–Attribut existiert, wird in der Datenbank nur das New–Flag (N) gesetzt. Weitere Angaben zu den Flags finden sich in Anhang A.

- Die "Content"–Attribute (Attribute, die direkt zum MIME–Header gehören) werden nach dem Auslesen nicht gelöscht, sondern zusätzlich ins Datenbank–Attribut "con_desc" geschrieben, da dies dem E–mail–Datenbank–Server und dem E–mail–Datenbank–Client das Auslesen der E–mails wesentlich erleichtert.

- Weiterhin gibt es Header–Informationen, die nicht als eigenständige Attribute in der Datenbank definiert sind. Diese gehören in den optionalen Header. Der optionale Header war ursprünglich zweigeteilt gedacht, und zwar als optionaler Header nach RFC 822 und als optionaler Header ("con_desc") nach MIME. Das Problem ist, daß eigentlich die Zugehörigkeit der Attribute zu dem einen oder anderen optionalen Header vom Parser nicht entscheidbar ist. Deshalb wird momentan der optionale Header komplett in das "con_desc"–Feld geschrieben.

- Der optionale Header ist, wie auch "Body" und "Received", ein Attribut, das für den *DB2*–Interpreter nicht–konforme Zeichen enthalten kann und als CLOB definiert wurde. Auch

"Subject" gehört in diese Klasse von Attributen, da auch hier nicht–konforme Zeichen vorkommen können. Diese Attribut–Klasse muß gesondert in die Datenbank geschrieben werden, da sie beim normalen Datenbank–"INSERT"–Befehl nicht ohne Probleme in die Datenbank aufgenommen werden. Die Attribute, die als CLOB in der Datenbank definiert wurden, werden daher in Dateien ausgelagert. "Subject" als Attribut mit nicht–konformen Zeichen wird so präpariert, daß es in die Datenbank geschrieben werden kann. Wie solche Attribute in die Datenbank eingetragen werden, wird nachfolgend beschrieben.

- Sobald alle erreichbaren Informationen aus der E–mail extrahiert sind, wird versucht, Attachments aus dem Mail–Body zu extrahieren. Dazu wird der Mail–Body per *Perl*–Methode zerlegt und — falls weitere Teile existieren — auch diese nacheinander syntaktisch analysiert. Dabei muß die Hierarchie und die Reihenfolge der Teile und deren Einzel–Informationen beachtet werden. Da die vorherigen bzw. übergeordneten Teile noch nicht in der Datenbank enthalten sind, muß also eine Art Baum der E–mail erstellt werden, um alle Teile eindeutig identifizieren und später auch wieder zusammensetzen zu können.

2.2.2 Einfügen von E–mails in die Datenbank

Nachdem eine E–mail in ihre Einzelteile zerlegt und soweit präpariert wurde, steht dem Einbringen der Daten in die Datenbank kaum noch etwas im Wege. Wie schon erwähnt, wird die Datenbank per Shell–Skript (siehe Anhang B für ein generiertes Beispiel–Skript) gefüllt. Aus den extrahierten Informationen, die nun als Attribut–Wert–Paare vorliegen, werden dazu SQL–Anfragen generiert, die durch das Shell–Skript ausgeführt werden.

Begonnen wird das Skript mit dem Setzen der *DB2*–Umgebungsvariablen. Danach wird in ein temporäres Arbeitsverzeichnis gewechselt, in dem die notwendigen Dateien gelagert werden.

Im Anschluß daran wird die Datenbank geöffnet. Hierbei ist es wichtig, daß keine Autorisierung erforderlich sein darf, da ansonsten der Nutzer bei jeder eingehenden E–Mail zugegen sein müßte. Hierin liegt auch begründet, warum die Remote–Verbindung autorisierungsfrei sein muß.

Der nächste Schritt, das Ausschalten des automatischen Commit, ist sehr wichtig, da sonst das Einfügen einer kompletten E–mail nicht als eine Transaktion angesehen würde. Dieser Umstand ist deshalb so wichtig, da bei Problemen keine unvollständigen E–mails in die Datenbank geschrieben werden sollen.

Die nächsten Schritte wiederholen sich zumeist für jeden Mail–Teil:

Als erstes muß für jeden Mail–Teil eine OID generiert werden. Diese OID wie auch alle weiteren OIDs ("ObjectIDentifier" zur Identifikation zusammengehöriger E–mail–Bestandteile in der Datenbank; s. Unterabschnitt 1.4) und UIDs ("Unique IDentifier" zur Identifikation einer E–mail durch IMAP4; s. Unterabschnitt 1.3) werden in der Tabelle "next_oid" bzw. als "uid_next" in der Tabelle "mailfolder" vorgehalten. Es existiert für jeden Tabellentyp ein solcher Eintrag. Nach dem Auslesen muß der OID–Zähler in der "next_oid"–Tabelle inkrementiert werden.

Die nutzergenerierte OID für einen Mail–Teil wird für mehrere spätere Einsätze benötigt und muß deshalb zwischengespeichert werden und im gesamten Skript eindeutig durch eine Variable erreichbar sein. Dieser Wert wird benötigt, um zum einen den aktuellen Mail–Teil zu identifizieren, um bei eventuellen Kind–Teilen der Mail–Teile–Hierarchie die "parent"–Referenz zu belegen und um die Teile in der Import–Tabelle eindeutig dem Mail–Teil zuordnen zu können.

Die Identifizierung durch die OID ist auch wichtig, um die E–mails dem richtigen Mail–Folder zuzuordnen. Dies geschieht nur in der ersten Hierarchie–Ebene.

Für die Zuordnung von E–mails zu Mail–Foldern werden eine OID für diese Datenbank–Tabelle und eine UID benötigt, die die E–mail innerhalb des Mail–Folders eindeutig identifiziert.

Für den Import wird eine Datei verwendet, in der alle Daten der zu importierenden Attribute stehen. Durch die Angabe "MODIFIED BY LOBSINFILE" in der Import–Anweisung werden Large Objects (LOBS) nicht in die generierte Datei geschrieben, sondern stehen in einer eigenen Datei und werden nur mit dem Dateinamen in der generierten Datei angegeben. Da die OID des jeweiligen Mail–Teils auch benötigt wird, kann die Erstellung der Datei erst im Shell–Skript erfolgen. Die OID muß beim Import deshalb genannt werden, damit beim anschließenden Update auch das richtige Tupel in der Mail–Teile–Tabelle aktualisiert wird. Bevor ein UPDATE–Kommando durchgeführt werden kann, muß natürlich zuerst einmal die jeweilige Mail–Teile–Tabelle mittels "INSERT INTO ..." mit der OID und allen einfachen Attributen gefüllt werden. Nach dem Update wird der Import–Eintrag wieder gelöscht. Nach diesem Schema werden sämtliche Teile einer E–mail in die Datenbank eingetragen.

Zum Schluß wird ein explizites Commit durchgeführt, die Datenbank–Verbindung geschlossen und die temporären Dateien und Verzeichnisse gelöscht.

2.2.3 Die Benutzung des Parsers

Um den Parser zu benutzen, müssen einige Voraussetzungen erfüllt und einige Dinge vorbereitet bzw. installiert sein. Als erstes muß der Nutzer die Möglichkeit des autorisierten Zugangs zur Datenbank, zu dem Rechner mit Datenbank–Anbindung und zu dem Rechner, der die E–mails verteilt, haben. Weiterhin muß es möglich sein, eine Remote–Verbindung ohne Passwort–Abfrage zwischen Mail–Verteiler und Datenbank–Client aufzubauen. Dazu muß eine `.rhosts`–Datei mit folgenden Einträgen angelegt werden.

```
<Rechnername Mail-Verteiler> <Nutzername>
<Rechnername Datenbank-Client> <Nutzername>
```

Momentan ist es auch unabdingbar, daß das Home–Verzeichnis des Nutzers auf beiden Rechnern gemountet ist (z.B. mittels NFS; Network FileSystem: Möglichkeit, über das Netzwerk Partitionen von fremden Rechnern zu mounten). Da die temporären Dateien im Home–Verzeichnis des Nutzers angelegt werden, muß auch genügend Speicherplatz vorhanden sein. Dies ist keine Einschränkung, denn auch ohne den Mail–Server zu benutzen, würden die E–mails bei Speicherplatzmangel nicht zugestellt werden. Weiterhin muß die Mail–Verteiler–Software (*procmail*) so konfiguriert werden, daß eingehende E–mails zur Verarbeitung an den Parser geschickt werden. Die nötigen Eintragungen in der Datei `.procmailrc` im Home–Verzeichnis des Nutzers, die den Parser unter der angegebenen Adresse ansprechen, lauten folgendermaßen:

```
:0 w: local.lock
| <Nutzer-Home-Verzeichnis>/ksws/parser/mimeparser
```

Als letztes muß nun noch die Datenbank eingerichtet werden. Dazu wird erst das Schema generiert und dann noch einige Initialisierungen vorgenommen. Das Schema wird einfach per

```
db2 -tf <Schema-Datei>
```

angelegt. Genauso wird das Initialisierungs–Skript gestartet.

2.3 Probleme und Ausblick

Mit den beschriebenen Mitteln wurde ein lauffähiges Programm entwickelt, das einfache RFC 822–E–Mails und beliebig stark strukturierte MIME–E–mails syntaktisch analysiert und in die Datenbank importiert.

Bekannte Probleme und ihre Gründe Die gravierendsten Mängel des Systems bestehen bei der Sicherheit. Beispielsweise wird bei den Parsern, die vom E–mail–Datenbank–Server und dem E–mail–Datenbank–Client eingesetzt werden, der Nutzername und das Passwort im Programm verwendet. Das Problem ist, daß sich der Nutzer für die Nutzung des IMAP4–Protokolls autorisieren muß. Dabei wird das Passwort unkodiert in den beschriebenen Programmen verwendet. Die Angriffsmöglichkeiten müssen hier als enorm eingeschätzt werden.

Die Laufsicherheit des Programms ist im Allgemeinen gewährleistet. Vor allem kann garantiert werden, daß E–mails beliebiger Schachtelungstiefe und beliebiger Zusammensetzung vollständig in die Datenbank eingebracht werden können. Schwierigkeiten können jedoch noch bei unvorhergesehenen "Spezialmails" von "Spezialsoftware" drohen, die am Rande der Möglichkeiten der Mail–Spezifikation liegen.

Die vorgeschlagene Architektur eines E–Mail–Parsers in der genannten Umgebung ist nicht die einzig mögliche Lösung. Eine gute Lösung ist auch in der folgenden Form vorstellbar: Der Mail–Verteiler startet per Remote–Shell ein Skript auf einem Datenbank–Client–Rechner und übergibt "remote" die gesamte E–mail an das Skript. Der Vorteil dieser Lösung wäre, daß hier keine NFS–fähige Netzstruktur mehr benötigt würde. Ein Nachteil wäre allerdings, daß sich alles auf dem Datenbank–Rechner abspielt und die verteilte Ausführung wegfällt.

Die angesprochene Aufteilung des optionalen Headers in RFC 822–Header und MIME–Header macht Sinn, da damit weitere Arbeitsschritte vereinfacht werden. Eine Möglichkeit, die beiden Arten optionaler Header–Informationen zu unterscheiden, wäre, alle mit "Content-" beginnenden Attribute als MIME–Attribute und alle übrigen als RFC 822–Attribute anzusehen. Dies könnte einfach im Parser realisiert werden und auch einen beträchtlichen Nutzen für den E–mail–Datenbank–Server und den E–mail–Datenbank–Client mit sich bringen.

Bei Verwendung einer neueren Version als *DB2 UDB 5.2.0 (Fixpack 6 – Sep. 98)* kann sauberer mit den OIDs umgegangen werden, da nutzergenerierte OID als datenbanktheoretisch unsauber betrachtet werden müssen. Außerdem könnten mit *DB2*–Extendern (z.B. Text–Extender, Image–Extender etc.) Zugriffsmechanismen auf die einzelnen Mail–Teile angeboten werden, die eine bessere Anfrageunterstützung bei der Nutzung von E–mail–Funktionalität mit sich bringen würden. Dazu müßte allerdings die Dekodierung im Parser wieder eingeschaltet, die Datenbank angepaßt und als schwierigstes Detail vom E–mail–Datenbank–Server ein dahingehend erweitertes IMAP4++–Protokoll unterstützt werden.

Sollte in einer neuen Version von *DB2* auch der Import–Bug behoben sein, so würde die zusätzliche Tabelle "import" nicht mehr benötigt.

Ein Problem, welches sich beim Import zeigte, ist, daß am Ende jedes Imports die Datenbank ein automatisches Commit durchführt. Dies ist in der verwendeten Version von *DB2* nicht zu unterbinden. *DB2* kann offensichtlich mit sehr großen Objekten transaktionstechnisch nicht lange umgehen. Dies verhindert natürlich das angestrebte Ziel, eine komplette E–mail in einer einzigen Transaktion in die Datenbank zu überführen.

Mögliche Erweiterungen des E–mail–Parsers Um die Verarbeitung der E–mails zu beschleunigen, müßte die momentane Beschränkung der Warteschlange bei *procmail* aufgehoben

werden. Daraus ergäbe sich jedoch auch die Notwendigkeit, die Sicherheit für den parallelen Betrieb der Software zu erhöhen. Ein weiteres Problem besteht in diesem Zusammenhang darin, daß die gesamte E–mail beim Parsing durch die Software in den Speicher geladen wird. Bei sehr vielen großen E–mails im parallelen Betrieb könnte es hier zu Engpässen bei den Ressourcen kommen.

Für die Verwendung anderer DBMS wird es neben einer Anpassung des Datenbank–Schemas nötig sein, die Umgebungsvariablen anzupassen, die SQL–Anfragen zu überarbeiten und damit die Möglichkeiten des neuen DBMS und dessen Befehls–Interpreters auszunutzen.

Momentan bietet *Perl* keine Datenbank–Schnittstelle (z.B. DBI) zur verwendeten *DB2*–Version. Gäbe es ein solches *Perl*–Modul, so könnte die in Unterabschnitt 1.5 vorgestellte Architektur wesentlich einfacher und zudem effizienter aufgebaut werden. Man müßte dazu nicht mehr relativ umständlich den Befehlsinterpreter bemühen.

Zunächst war geplant, alle Einsatzgebiete des Parsers in einem Programm zu integrieren. Aufgrund der Vielfalt der unterschiedlichen Anforderungen aus den drei Einsatzgebieten wurde dieser Versuch jedoch aufgegeben. Die größten Probleme waren hier die Übergabe der Nutzernamen und Passwörter und die Bereitstellung der temporären Arbeitspfade. Trotzdem sollte es prinzipiell möglich sein, alle drei Einsatzgebiete in ein Programm zu integrieren.

Insgesamt kann festgestellt werden, daß einige Möglichkeiten zur Erweiterung und zur Modifikation des E–mail–Parsers bestehen, die Software aber in jedem Fall die gestellte Aufgabe einer prototypischen Umsetzung des Problems einer syntaktischen Analyse und des Imports von E–mails in eine Datenbank gerecht wird und dabei stabil genug funktioniert, um als Grundlage des Gesamtsystems eingesetzt zu werden.

3 Der E–mail–Datenbank–Server

Der E–mail–Datenbank–Server basiert auf der IMAP4–Referenz–Implementierung der University of Washington [Cri98].

Die Dateien müssen entpackt werden und dann auf einem Rechner mit freiem `/tmp`-Verzeichnis (in der aktuellen Infrastruktur des Fachbereichs Informatik der Universität Rostock also nicht auf `hokkaido`) kompiliert werden. Unter *Sun Solaris* lautet der Befehl `make sol`. Das Programm funktioniert aber auch unter *GNU/Linux*. Im Unterverzeichnis `imapd` liegt dann der IMAP–Dämon `imapd`. Dieser kann direkt gestartet werden, und erlaubt so ein Arbeiten auf Kommandozeile im "PREAUTH"–Modus (dieser entspricht dem "Authenticated state" aus Unterabschnitt 1.3; es ist also ohne Angabe von Login und Passwort möglich, auf Mail–Foldern zu arbeiten). Alternativ kann das Programm auch von einem System–Administrator als daemon registriert werden, um so automatisch gestartet zu werden, sobald ein Client eine Verbindung zum IMAP4–Port 143 aufbaut. In diesem Fall läuft das Programm unter den Administrator–Rechten und verlangt daher die Angabe eines *Unix*–Accounts und eines Passworts.

Die Umsetzung des E–mail–Datenbank–Servers sollte auf der IMAP4–Referenz-Implementierung aufbauen, wobei dieser soweit umgeschrieben werden mußte, daß die E–mails nicht mehr wie üblich aus Dateien ausgelesen werden, sondern eine Datenbank verwendet wird. Implementiert wurde auch für diesen Server der Zugriff auf *DB2*, um die durch den E–mail–Parser (s. Abschnitt 2) in die Datenbank importierten E–mails als Datenbestand nutzen zu können. Der E–mail–Datenbank–Server liest die E–mails nach Bedarf (je nach Wunsch des E–mail–Datenbank–Clients aus Abschnitt 4 oder jedem beliebigen anderen IMAP4–fähigen Mail–Client)

aus der Datenbank aus, legt Mail–Folder an, benennt Mail–Folder um, löscht Mail–Folder, ändert E–mail–Flags, kopiert E–mails oder löscht E–mails.

Der *DB2*–Server läuft (in der aktuellen Infrastruktur des Lehrstuhls Datenbank– und Informationssysteme des Fachbereichs Informatik der Universität Rostock) auf dem Rechner `hades`. Vor der Benutzung der Datenbank muß zunächst `source ~db2inst1/sqllib/db2cshrc` (bzw. `db2profile`, abhängig von der verwendeten Shell) ausgeführt und das Skript zum Anlegen der Tabellen in der Datenbank ausgeführt werden (sofern dies noch nicht gemäß der Beschreibung zum Parser in Unterabschnitt 2.2 geschehen ist). Sowohl die Kompilation, als auch die Ausführung des geänderten Servers muß derzeit auf `hades` stattfinden.

Die vollständige Beschreibung der Implementierung des E–mail–Datenbank–Servers findet sich in [KWR99].

3.1 Das Treiber–Prinzip

Die Sourcen des Standard–IMAP–Servers erlauben die Kapselung aller Aktionen zum Laden/ Speichern von E–mails. Dazu wird eine Liste aller Mail–Treiber gehalten, die immer, wenn ein Client einen IMAP4–Befehl schickt, nach einem passenden Treiber durchsucht wird. Derjenige Treiber, welcher die Aufgabe erfüllen kann, wird aufgerufen. Auf diese Weise unterstützt der Standard–Server gleichzeitig lokale E–mails und news–Gruppen.

Aufgrund dieser Ausgangs–Situation war es für den E–mail–Datenbank–Server lediglich notwendig, einen neuen Treiber zu entwickeln. Auf Änderungen am eigentlichen Hauptprogramm konnte weitestgehend verzichtet werden. Mail–Treiber müssen als `treibername.c` und `treibername.h` existieren und im Makefile der jeweiligen Plattform, d.h. im aktuellen Fall unter Verwendung der Software unter *Sun Solaris* in `src/osdep/unix/Makefile`, unter der Umgebungsvariable `DEFAULTDRIVERS` oder `EXTRADRIVERS`, registriert werden. Standardimplementierungen finden sich z.B. in `src/osdep/unix/unix.c`; die Implementierung des E–mail–Datenbank–Servers liegt in `src/osdep/unix/db2mail.sqc`.

Bei dem E–mail–Datenbank–Server handelt sich um einen *C*–Quelltext, der Embedded–SQL–Befehle enthält und vor der Kompilierung vom Präprozessor in `db2mail.c` umgewandelt wird. Bei der vorliegenden Implementierung des E–mail–Datenbank–Servers wurden sämtliche anderen Treiber “auskommentiert”, d.h. aus den Umgebungsvariablen gestrichen, da sie nur stören würden (alle E–mails sollen in der Datenbank gespeichert werden; der Mail–Folder “INBOX” würde jedoch ansonsten von einem anderen Treiber höherer Priorität verwaltet werden). `db2mail.sqc` enthält die Definition einer Struktur vom Typ “DRIVER”. Dort sind Name, Flags und Zeiger auf die unterstützten Funktionen angegeben.

Sobald ein Client einen IMAP4–Befehl zum Selektieren eines Mail–Folders an den IMAP–Server schickt, wird vom IMAP–Kern (definiert in `c-client/mail.[ch]`) die Liste der Mail–Treiber durchsucht und die Treiber jeweils befragt, ob sie den jeweiligen Mail–Folder unterstützen. Dabei wird auch die neu implementierte Funktion `db2mail_valid(char* name)` aufgerufen. Diese prüft mit einer Datenbank–Anfrage, ob der jeweilige Mail–Folder existiert, und liefert im positiven Fall einen Zeiger auf `db2maildriver` als Rückgabewert. Anschließend verwendet der IMAP–Kern diesen Treiber bei der Abarbeitung der vom Client gesendeten Befehle. Für weitere Details zu den implementierten Funktionen im *DB2*–Mail–Treiber wird auf den Quellcode verwiesen [KSW99].

3.2 Ideen für Optimierungen

Während der Entwicklung des E–mail–Datenbank–Servers wurden einige Probleme identifiziert, deren Behebung einige Geschwindigkeitssteigerungen bringen würde.

3.2.1 Verbindungsauf– und –abbau

Bei jeder Aktion (Abarbeitung eines Client–Befehls) wird die Verbindung zur Datenbank neu aufgebaut und hinterher wieder geschlossen. Dies ist insofern notwendig, als ein Absturz bzw. eine unvorhergesehene Beendigung des Serverprogramms ansonsten einen Datenbank–Prozess hinterlassen würde.

Die Datenbank–Zugriffe könnten wesentlich beschleunigt werden, indem die Datenbank-Verbindung nur beim Start des Programms hergestellt und erst bei Beendigung desselben getrennt würde. Speziell neuere Versionen von *Netscape mail* schicken sehr viele "LIST"–Befehle (zur Erkundung, welche Mail–Folder existieren) an den Server und bauen mehrere IMAP4–Verbindungen parallel auf, was mit der derzeitigen Implementierung einige Zeit in Anspruch nimmt.

3.2.2 Implementierung weiterer DRIVER–Methoden

Beim Schreiben eines IMAP–Mail–Treibers müssen nicht sämtliche Methoden implementiert werden. Beispielsweise braucht der Mail–Treiber nicht die Struktur einer E–mail selbst zu ermitteln, es reicht, Funktionen zur Bereitstellung des Mail–Kopfes und des Mail–Bodies einzubauen. Für die vorliegende Implementierung wären dies `db2mail_header` und `db2mail_text`.

Wenn ein Client z.B. die Struktur einer E–mail, bestimmte Attachments (Mail–Bodies) oder nur spezielle Zeilen des Mail–Kopfes anfordert, dann ruft der IMAP–Kern die beiden genannten Funktionen auf und verwendet den eingebauten Parser zur Extraktion der benötigten Teile. Allerdings ist diese Vorgehensweise keinesfalls effizient, zumal die E–mail–Bestandteile bereits voneinander getrennt in verschiedenen Attributen im Datenbank–Schema gespeichert sind. Die Beantwortung von Client–Anforderungen könnte deshalb schneller, bzw. mit einem deutlich geringeren logischen Aufwand vonstatten gehen. Die Datenbank–Lösung ist dafür prädestiniert, direkt die richtigen Daten zu selektieren (bzw. auf diese zu projizieren), anstatt den Weg zu gehen, jedesmal eine gesamte E–mail zu rekonstruieren, die anschließend ohnehin wieder auf die eigentlich interessierenden Anteile verkürzt werden muß. Deshalb wäre es für den E–mail–Datenbank–Server sinnvoll, einige der durch die "DRIVER"–Spezifikation unterstützten möglichen Funktionen zu implementieren. Denkbar und besonders sinnvoll erscheint die Implementierung der folgenden Treiber–Funktionen (die Reihenfolge gibt die Einschätzung des absteigenden, erreichbaren Gewinns — des Sinns — der Implementierung an):

db2mail_partial nur einen Teil der E–mail ausgeben, d.h. bestimmten Teil (Startindex → Endeindex) bzw. bestimmte Attachments

db2mail_structure nur die Struktur der E–mail ausgeben

db2mail_search eine Liste von E–mails ausgeben, die bestimmte Suchkriterien erfüllen

db2mail_overview für eine Sequenz von E–mails eine Übersicht ausgeben

db2mail_flag für eine Sequenz von E–mails Flags ändern

db2mail_sort anhand eines Kriteriums die E–mails eines Mail–Folders sortieren, d.h. neue UIDs zuordnen

3.2.3 Änderung des Schemas zur Unterstützung hierarchischer Mail–Folder

In der aktuellen Implementierung des E–mail–Datenbank–Servers wird die Hierarchie der Mail–Folder ausschließlich über die Namen der Mail–Folder verwaltet, untergeordnete Mail–Folder sind alle Mail–Folder, die etwa dem Ausdruck "Mail–Folder'Separatorzeichen'Zeichenkette" entsprechen. Dies macht Operationen wie das Umbenennen eines Mail–Folders, der "links" in der Hierarchie steht, sehr aufwendig. Dies gilt insbesondere, wenn eine große Anzahl von Mail–Foldern in einer Hierarchie liegt. Es erscheint überlegenswert, eine Verkettung mit Hilfe von Referenzen als alternative Darstellungsform in Betracht zu ziehen, da diese Vorgehensweise vermutlich deutlich besser geeignet wäre.

3.2.4 Vereinfachung der Authentifizierung

Wird der IMAP–Server durch den System–Administrator gestartet, so meldet sich das Programm zunächst beim Betriebssystem und dann bei *DB2* mit dem zuvor vom Client übertragenen Benutzernamen und Passwort an. Dies bedeutet, daß alle Nutzer des E–mail–Datenbank–Servers auch einen *Unix*–Zugang und einen Eintrag in der Nutzerverwaltung von *DB2* benötigen. Dies wäre bei einem Server, der von einer großen Anzahl von Nutzern für die Nutzung von E–mail–Funktionalität verwendet wird, wie z.B. einem Rechner in einem Universitäts–Rechenzentrum, zu aufwendig. Denkbar wäre die Entwicklung eines eigenen Authentifizierungsmoduls, das direkt auf *DB2* zugreift. Da jeder Nutzer sein eigenes Schema in der gemeinsamen Datenbank verwendet, muß überlegt werden, ob dies auch ohne separates Anlegen eines *Unix*–Benutzers erstellt werden könnte.

3.3 Änderungen an den Originalquellen

Durch das Treiberkonzept des IMAP–Servers waren nur geringfügige Änderungen an den Programmteilen außerhalb des neu geschaffenen Moduls notwendig. Da für die Anmeldung beim Datenbanksystem Nutzername und Passwort benötigt werden, wurde die Authentifizierungsroutine dahingehend geändert, daß sie das Passwort in einer globalen Variable speichert. Dies führte dazu, daß sich andere, hier nicht verwendete Programme der Distribution nicht mehr übersetzen ließen, was aber leicht durch Hinzufügen der Variable zu den betroffenen Dateien behoben werden könnte.

3.4 Probleme

In der vorliegenden Implementierung des E–mail–Datenbank–Servers gibt es Probleme mit Zeilenumbrüchen und dem Attribut, das die Länge von E–mails beinhaltet.

3.4.1 Zeilenumbrüche

Beim Testen des E–mail–Datenbank–Servers mittels verschiedener Clients fiel auf, daß z.B. *Netscape* unbedingt "CR/LF" als Zeilenumbruch erwartet, d.h. "\r\n". Sind diese Umbrüche nicht bzw. nur "\n" vorhanden, dann bricht *Netscape* beim Lesen der Kopfzeilen (Header–Informationen) ab.

Der implementierte Server fügt daher diese Zeilenumbrüche nun explizit bei der Ausgabe der Kopfzeilen ein, wodurch *Netscape* die E–mails dann akzeptiert. Da die E–mail–Header in der Datenbank als Attribut–Wert–Paare ohne diese Form der Zeilenumbrüche gespeichert sind, kann es Probleme geben, wenn eine E–mail mittels *Netscape* ausgegeben wird. Da die Benutzung dieses Zeilenumbruchs so auch in den RFCs spezifiziert ist, muß der Zeilenumbruch bei der Ausgabe in `db2mail_text` angefügt werden.

3.4.2 Die Länge von E–mails

Wenn Clients die Länge einer E–mail ausgeben, so wird in der aktuellen Version lediglich der Wert aus "con_length" der E–mail ausgegeben. Dieser ist aber oft nicht angegeben, daher wird ein frei gewählter Standardwert (z.Zt.: 2000) verwendet. Hier müssen eventuell der Parser oder der Server explizit die Länge der E–mail ermitteln und in die Tabelle "mime" (s. Anhang A) eintragen, damit die verwendeten Clients nicht aufgrund abweichender E–mail–Länge Warnungen versenden. Die Funktionalität des Gesamtsystems wird durch den verwendeten Standardwert nicht eingeschränkt.

4 Der E–mail–Datenbank–Client

An den E–mail–Datenbank–Client wurden die folgenden Anforderungen gestellt:

- Mail–Verwaltung in einer Datenbank (nicht in den üblichen Mail–Foldern)
- Unterstützung des IMAP4–Protokolls (und evtl. eigenen Erweiterungen)
- Arbeit als offline– bzw. disconnected–Client.
  Der Client soll möglichst autonom arbeiten und nur zum Empfangen neuer E–mails bzw. zum Datenabgleich mit dem Mail–Server in Verbindung treten müssen (Datenreplikation und Synchronisation)

Die vollständige Beschreibung der Implementierung des E–mail–Datenbank–Clients findet sich in [NMS99].

Nachfolgend werden einige Aspekte der Entwicklung des E–mail–Datenbank–Clients diskutiert. Unterabschnitt 4.1 widmet sich der Auswahl der verwendeten Programmiersprache, bevor Unterabschnitt 4.2 detailliert auf die Implementierung der einzelnen Komponenten des E–mail–Datenbank–Clients eingeht und in Unterabschnitt 4.3 schließlich die implementierte grafische Oberfläche beschrieben wird. Abschließend behandelt Unterabschnitt 4.4 die Verwendung des E–mail–Datenbank–Clients im online–Betrieb.

4.1 Die Wahl der Programmiersprache

Als erstes wurden die möglichen Programmiersprachen und –werkzeuge untersucht. In die engere Auswahl kamen schließlich *Java*, *Perl* und *Python*. *C/C++* schied für die Programmierung eines Mail–Client–Prototypen aufgrund des doch erheblichen Programmieraufwands aus. Die genannten Programmiersprachen bieten allesamt Bibliotheken für das Bearbeiten von E–mails und für die IMAP4–Kommunikation. Außerdem gibt es keine Probleme mit Speicherlecks oder "Dangling Pointern", was die Fehlersuche erheblich beschleunigt, so daß die eigentliche Funktionalität des Programms in den Mittelpunkt der Arbeiten gestellt werden kann.

Java bietet zwar schon einen Beispiel–Mail–Client, IMAP4–Bibliotheken und sogar eine JDBC–Anbindung zur Datenbank, dennoch fiel die Entscheidung zugunsten der neuen Skript–Sprachen, die mit ihrem Interpreter–/Vorcompiler–Konzept ein wesentlich rascheres Prototyping und Debugging zulassen und in ihrer Mächtigkeit den herkömmlichen Programmiersprachen in nichts nachstehen. Das Gegenteil ist sogar der Fall: meistens sind ihre Fähigkeiten in der String–Verarbeitung (z.B. Parsing) und die angebotenen Datentypen weitaus fortgeschrittener.

Letztendlich fiel die Wahl auf *Python* [HM99, Lut96, WvRA96, vLF97], welches neben Bibliotheken für das Bearbeiten von E–mails und die IMAP4–Kommunikation auch noch eine *Tk*–Anbindung beinhaltet, welche für die Oberflächen–Programmierung natürlich von großem Nutzen war. Einziger Nachteil an *Python* (allerdings nur gegenüber JAVA und *C*) ist die fehlende Datenbank–Anbindung, z.B. per SQL–Schnittstelle. Diese Anbindung wurde mit Hilfe des Kommandozeileninterpreters (CLI) der Datenbank und passenden Shell–Skripten implementiert. Dieser Umweg führt zu einem drastischen Performance–Einbruch, da die Datenbank–Verbindung ständig neu aufgebaut und wieder abgebaut werden muß. Allerdings bietet dieses Verfahren wiederum den Vorteil der leichten Programmierbarkeit, guter Wartbarkeit, hoher Fehlertoleranz (keine Abstürze) und vor allem einer guten Portierbarkeit auf andere DBMS.

4.2 Die Implementierung

Dieser Unterabschnitt beschreibt den Datenbank–Zugriff, führt Besonderheiten der Implementierung auf und beschreibt abschließend bestehende Probleme und mögliche Lösungen.

4.2.1 Der Datenbank–Zugriff

Es gibt eine "Tabular Databases SIG" (Special Interest Group; Zusammenschluß von an einem speziellen Thema Interessierter. Für *Python* gibt es SIGs für Bild–Bearbeitung, XML–Bearbeitung, Datenbank–Anbindung und weitere), die sich mit dem Zugriff auf relationale Datenbanken aus *Python* heraus beschäftigt. Es wurde eine Standard API für den Datenbank–Zugriff definiert, welche es in einem Modul zu implementieren gilt. Für eine Reihe von verbreiteten Systemen wie *Informix*, *mySQL*, *SyBase* und *Oracle* existieren bereits derartige Module. Ebenso existiert ein ODBC–Modul für *Windows*.

Leider existiert jedoch keine Bibliothek für den Zugriff auf die im Rahmen der Implementierung des E–mail–Datenbank–Clients zu verwendende Datenbank *DB2 UDB* [Cha98, IBM98]. Aus diesem Grund mußte ein anderer Weg für den Datenbank–Zugriff gewählt werden. Eine Möglichkeit wäre gewesen, eine neue *Python*–Bibliothek in *C* zu implementieren, welche die Kommunikation mit einer Datenbank übernimmt, also die API der SIG implementiert. Da in Unterabschnitt 4.1 aber gerade gegen eine Programmierung in *C* entschieden wurde und auch die Anbindung an ein weiteres DBMS evtl. noch realisiert werden sollte, wurde ein einfacherer und zugleich offenerer Weg ausgewählt.

Python bietet, wie die meisten Programmiersprachen auch, die Möglichkeit, Programme auf Betriebssystem–Ebene aufzurufen (`system()-call`). Da jede Datenbank auch die Möglichkeit der Bedienung über einen Kommandozeileninterpreter (CLI) bietet, lag die Idee nahe, einfach diesen CLI mit den entsprechenden Parametern aufzurufen. Da der CLI seine Ergebnisse in einem nicht immer brauchbaren Format ausgibt und das vor allem auch noch auf die Konsole, mußten diese Aufrufe allerdings noch gekapselt werden. Ein weiterer Grund für die Kapselung ist der, daß eine Datenbank immer erst geöffnet werden muß, bevor auf sie zugegriffen werden kann. Datenbank–Verbindungen bestehen immer nur mit einem entsprechenden Prozess. Da bei

einem `system()-call` jedesmal ein neuer Prozess generiert wird, was automatisch zum Schließen der alten Verbindung und zu einer Fehlermeldung bei der neuen Anfrage führt, ist ein direkter Aufruf des CLI von *Python* aus nicht möglich.

Die einfachste Lösung für beide Probleme sind kleine Shell–Skripte. Ein Shell–Skript läuft als ein Prozess und hat dadurch ständig Zugriff auf eine einmal geöffnete Datenbank (allerdings muß diese Verbindung bei jedem Aufruf des Skripts neu hergestellt werden). Außerdem wird gleichzeitig die Ausgabe in eine Datei umgeleitet und mit Hilfe von *Unix*–Standardtools so formatiert, daß sie einfach wieder in *Python*–Datenstrukturen eingelesen werden kann.

Der Zugriff auf die Daten in der Datenbank läuft daher also im Wesentlichen wie folgt ab:

1. Es gibt eine Reihe von Standard–Zugriffsmethoden zu den Daten in der Datenbank, die in gewöhnlichem *Python*–Code geschrieben wurden. Dies sind Methoden zum Auflisten aller vorhandenen Mail–Folder oder aller E–mails innerhalb eines Mail–Folders, zum Anlegen und Löschen von Mail–Foldern, sowie Methoden zur Bearbeitung einzelner E–mails. So können die Flags einer E–mail gesetzt, E–mails mit/ohne Attachments ausgegeben, E–mails verschoben und kopiert werden. Diese Funktionen stellen die äußere Schnittstelle für die Kommunikation eines Mail–Clients mit der Datenbank dar. Alle diese Grundfunktionen sind in der Bibliothek `mblib.py` definiert.

2. Jeder dieser *Python*–Funktionen ist genau ein Shell–Skript, meist mit gleichem Namen, zugeordnet. Im einfachsten Fall ruft also die *Python*–Funktion nur das entsprechende Skript mit den Parametern über den `os.system()-call` auf. Im etwas komplizierteren Fall müssen die Daten noch entsprechend in eine *Python*–Struktur eingelesen werden, z.B. wenn eine Liste zurückgegeben werden soll, oder aber noch Parameter geändert oder bereinigt werden müssen. Für die Behandlung von Parametern gibt es noch eine Reihe nützlicher Funktionen, die in `tools.py` definiert sind und z.B. bestimmte Flags in einem Flag–String setzen bzw. löschen, oder aber die Umwandlung von den E–mail–internen Flags auf IMAP4–Protokoll–konforme Flag–Bezeichnungen vornehmen.

3. In jedem Shell–Skript wird zuerst die entsprechende Datenbank (hier "mailclnt") mit dem jeweiligen Nutzernamen und Passwort geöffnet. Die Passwort–Abfrage muß der Client übernehmen und das richtige Passwort als "PASS"–Parameter in seinem Environment speichern. Als nächstes werden dann die nötigen Daten mittels einer oder mehrerer SQL–Anfragen in System–Dateien extrahiert. Danach werden diese Dateien formatiert. Bei Aufrufen, die keine Daten zurückliefern, sondern nur Daten ändern, fällt dieser Teil natürlich weg, und es wird nur der entsprechende SQL–Aufruf abgesetzt. Als vorletzte Aufgabe wird dann noch eine entsprechende Zeile mit den nötigen Informationen in die Log–Tabelle geschrieben. Zum Schluß kann die Datenbank wieder geschlossen werden, bei einem erneuten Aufruf des Skripts stünde diese Verbindung sowieso nicht mehr zur Verfügung. Um sauber zu programmieren und nicht zu viele Datenbank–Clients zu verbrauchen, wird also jede Verbindung wieder geschlossen. Damit ist der Zugriff auf die Datenbank abgeschlossen.

4. Falls Daten aus der Datenbank extrahiert wurden, dann stehen sie anschließend formatiert in einer entsprechenden System–Datei und können einfach mittels *Python*-Systemfunktionen eingelesen werden. Falls nicht, endet die Funktion mit der Rückgabe des Fehler–Codes des Shell–Skripts.

5. Das Einfügen von Daten (E–mails) in die Datenbank erfolgt mittels des in Abschnitt 2 beschriebenen E–mail–Parsers. Dieser Parser ist in *Perl* geschrieben und arbeitet auch mit einer Datenbank–Anbindung über Shell–Skripte. Die Schnittstelle dieses Werkzeugs enthält einen Parameter, den Namen des Ziel–Mail–Folders. Die E–mail selbst wird in eine Datei geschrieben, die über eine Umlenkung der Standard–Eingabe an den Parser übergeben wird. Der Parser wird mittels `os.system()` und dem entsprechenden Parameter aufgerufen. Der Rückgabewert ist der Fehler–Code dieses Aufrufs. Danach ist die E–mail in die Datenbank eingetragen.

4.2.2 Besonderheiten der Implementierung

Bei der Implementierung aller Funktionen wurde in erster Linie Wert auf die Vervollständigung ihrer Funktionalität gelegt. Es ergibt sich daher eine Einschränkung bei der Laufsicherheit der Bibliotheken. Alle Funktionen sichern eine absturzfreie Arbeit zu, solange alle Eingabeparameter korrekt sind und keine Probleme mit dem Datenbank–Zugriff auftreten. Anderenfalls erfolgt zwar kein Absturz, aber die Fehler werden nicht abgefangen und als Fehlermeldung an die aufrufende Funktion zurückgeliefert. Dies führt unter Umständen zu unschönen Fehlerausschriften auf der Konsole und einer Fehlerfortführung in der aufrufenden Funktion.

Bei der Client–Implementierung achtet die Oberfläche darauf, nur korrekte Daten zu übermitteln. Es bleibt aber weiterhin das Problem des Datenbank–Zugriffs. Leider kommt es bei dem verwendeten System (*DB2*) ab und an zu Störfällen, welche eine korrekte Datenübermittlung vereiteln. In diesen Fällen können dann evtl. Updates verloren gehen, bzw. die Darstellungskomponente erhält keine Ergebnisse.

Da es sich bei dem E–mail–Datenbank–Client nur um einen Prototypen zur Demonstration handelt, sind diese Einschränkungen nicht entscheidend. Bei einer Erweiterung zu einer richtigen Anwendung muß dies allerdings berücksichtigt werden (neben den im folgenden aufgeführten Problemen).

4.2.3 Probleme und Lösungen

Das einzige offensichtliche Problem der beschriebenen Lösung der Datenbank–Anbindung ist der Performance–Verlust durch das Öffnen der Datenbank bei jedem Funktionsaufruf. Obwohl nachfolgende Aufrufe schneller ablaufen, da die Datenbank–Clients nicht gleich wieder vernichtet werden, sondern für neue Anfragen zur Verfügung stehen, wäre doch eine ständig geöffnete Verbindung wünschenswert. Diese Art der Geschwindigkeitssteigerung wurde auch schon bei "FAST–cgi" praktiziert. Momentan bietet der E–mail–Datenbank–Client jedoch keine funktionsfähige Lösung für dieses Problem an.

Die beste Alternative stellt vermutlich die Implementierung der Datenbank–API für *DB2 UDB* dar. Wahlweise könnte das ganze System auch auf ein anderes DBMS portiert werden, für das bereits eine solche Implementation verfügbar ist. Die Portierung sollte ohne große Schwierigkeiten realisierbar sein, solange alle vom Datenbank–Schema in Anhang A verwendeten Datentypen und Konstruktoren verfügbar sind.

4.3 Die grafische Oberfläche

Im folgenden wird zunächst die Funktionalität der grafischen Oberfläche beschrieben, bevor Erweiterungsmöglichkeiten aufgeführt werden.

4.3.1 Beschreibung der Funktionalität

Die Oberfläche des E–mail–Datenbank–Clients besteht aus einem Menü, einer Informationsbox und einer "Toolbar" (s. Abbildung 4). Letztere dient dem schnellen Zugriff auf die wichtigsten Funktionen. Über das Menü sind sämtliche Funktionen erreichbar. Mittels der Funktionen "Folders", "Mails" und "Attachments" werden entsprechende Listen angezeigt und der Zugriff auf spezielle Funktionen der jeweiligen Objekte ermöglicht. Dabei wird zum Einen mit den Funktionen der Datei `mblib.py` und zum Anderen mit *Unix*–Programmen wie *mutt*, *joe*, *less* u.a. gearbeitet. Eine Portierung des Clients auf andere Systeme erfordert somit eine Anpassung der aufgerufenen Anwendungssoftware.

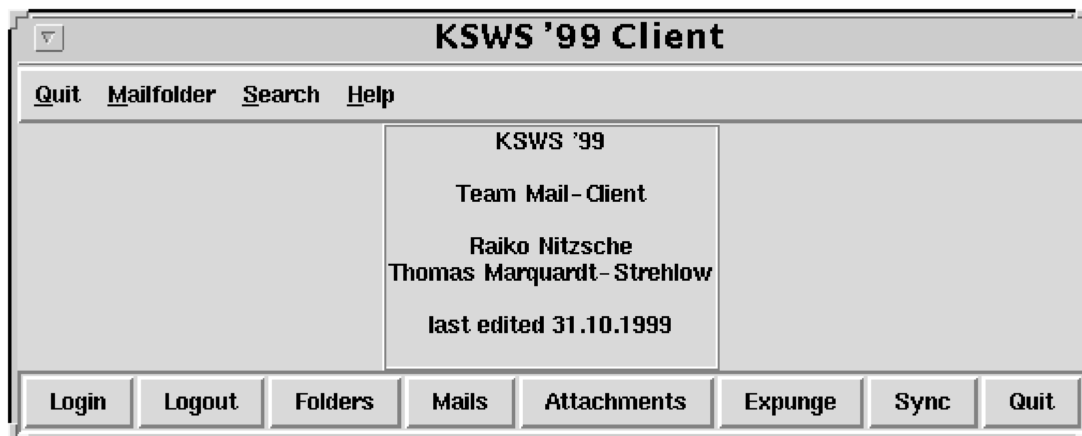

Abbildung 4: Startfenster des E–mail–Datenbank-Clients

Da der Client mittels *Python–Tk* implementiert wurde und dieses in der aktuellen Infrastruktur am Fachbereich Informatik der Universität Rostock nur unter *Linux* auf dem Rechner laxos zur Verfügung stand, mußte eine zusätzliche Autorisierung eingeführt werden. Diese könnte im Falle eines Programmstarts auf den meisten *Unix*–Rechnern sicherlich entfallen. Allerdings ergibt sich durch das Login der Vorteil des möglichen Nutzerwechsels. Leider ist dies jedoch nicht mit einem Mehrbenutzer–Betrieb vergleichbar, da lediglich ein implizites Logout des alten Nutzers und ein Login des neuen Nutzers erfolgen.

Mittels der "Logout"–Funktion werden sämtliche *DB2*–Zugriffe ohne ein erneutes vorheriges "Login" unterbunden. Vorher wird ein mögliches serverseitiges Löschen der clientseitig mit dem "Delete"–Flag markierten E–mails abgefragt. Diese Funktion des IMAP4–Kommandos "Expunge" ist auch direkt per Menü und Toolbar zugreifbar. Mittels "Sync" wird die volle Synchronisations–Funktionalität, die in Abschnitt 5 beschrieben wird, ausgeführt. Das Programm kann mit "Quit" unter Bestätigung der zugehörigen Nachfrage oder durch Schließen des Fensters beendet werden.

Für Dialoge bietet *Python–Tk* einige Bibliotheken an, wie z.B. `tkSimpleDialog`. Zur Passwort-Eingabe wurde die Klasse `tkSimpleDialog` zur Klasse `tkHiddenDialog` modifiziert. Sämtliche Nachrichten beendeter Operationen benutzen die Klasse `tkMessageBox`. Als Geometrie–Manager wurde der *Packer* benutzt — alternativ wären auch *Placer* u.a. denkbar.

Über die entsprechenden Menüpunkte und Buttons der Toolbar ist der Zugriff auf die Liste aller Mail–Folder, E–mails eines Mail–Folders oder Attachments einer E–mail eines Mail–Folders realisiert.

Mailfolder Unter dem Menüpunkt "Mailfolder" (Abbildung 5) sind die Funktionen betreffend der Mail–Folder, einzelner E–mails und Attachments gestaffelt verfügbar. Zu den Funktionen auf Mail–Foldern gehören die Anzeige aller Mail–Folder, das Anlegen und Löschen eines Mail–Folders sowie das Öffnen eines Mail–Folders, welches die Anzeige aller E–mails des Mail–Folders nach sich zieht.

Abbildung 5: Menüpunkt Mailfolder

Das Anlegen neuer Mail–Folder geschieht über die Funktion `create_mb` aus `mblib.py`. Vorhandene Mail–Folder werden über die Funktion `drop_mb` aus `mblib.py` entfernt (Abbildung 6).

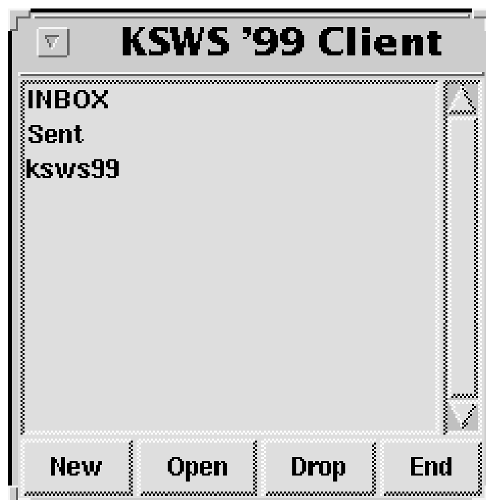

Abbildung 6: Mail–Folder–Liste

Mails Der Submenüpunkt "Mail" (Abbildung 7) enthält alle Operationen auf und mit E–mails. Zu diesen gehören das Erstellen und Versenden einer neuen E–mail, das Beantworten oder Weitersenden erhaltener E–mails, das Bewegen und Duplizieren von E–mails zwischen Mail–Foldern sowie die Ansicht, das Speichern und das Löschen von E–mails.

Das Erstellen und das Versenden neuer E–mails erfolgt in der vorliegenden Client-Implementierung unter Nutzung des Programms *mutt*, wobei die E–mail jeweils mittels "Fcc:" unter `~/.smt/ksws99` gespeichert wird, um nach dem Versenden mittels des IMAP4–Kommandos "APPEND" in der Datenbank an den vorher angegebenen Mail–Folder angehängt zu werden. Dazu dient die Funktion `app_mail` aus `mblib.py`.

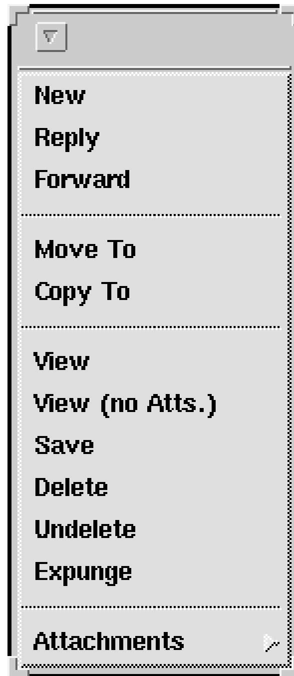

Abbildung 7: Submenüpunkt Mail

Bei “Reply” und “Forward” (s. Abbildung 8) wird nach der Auswahl einer E–mail diese aus der Datenbank exportiert und mittels *mutt* dem Nutzer zur Bearbeitung zur Verfügung gestellt. An dieser Stelle ist auch die Verwendung beliebiger anderer Mail–Clients möglich. Im Wesentlichen werden die Editor–Funktionalität und die Übergabe der erstellten E–mail an *sendmail* genutzt. Der Vorteil dieser Lösung besteht in der Ausnutzung der Attachment–Behandlung des Clients *mutt*. Nach erfolgreichem Versand der E–mail wird das “R”– bzw. “F”–Flag der alten E–mail gesetzt und eine Kopie an den vorher angegebenen Mail–Folder angehängt. Zum Setzen der Flags dienen die Funktionen `reply` und `forward` der Bibliothek `mblib.py`.

Weitere Funktionen auf E–mails sind die Speicherung einer E–mail, die Ansicht der gesamten E–mail und die Ansicht ohne Attachments. Die jeweilige E–mail wird hier nach `/tmp/mail_cnt` exportiert und von dort aus entweder gespeichert oder aber mittels *mutt* oder *less* aufgerufen. Bis auf die Funktion “View (no Atts)” sind alle Kommandos direkt über Buttons der Toolbar aufrufbar.

Attachments Über den Subsubmenüpunkt “Attachments” (Abbildung 9) kann auf die Funktionen für Attachments einer E–mail zugegriffen werden. Dazu gehören die Funktion des Auflistens aller Attachments einer E–mail sowie die Ansicht und das Speichern eines speziellen Attachments.

Die Funktionalitäten “Search” und “Help” sind zwar schon vorgesehen, aber noch nicht implementiert.

Die Attachments einer E–mail können aufgelistet werden und mittels Zugriff über die entsprechende Listennummer entweder mit einem beliebigen Programm angeschaut oder aber gespeichert werden (Abbildung 10). Dabei wird über die Liste der E–mails eines Mail–Folders und

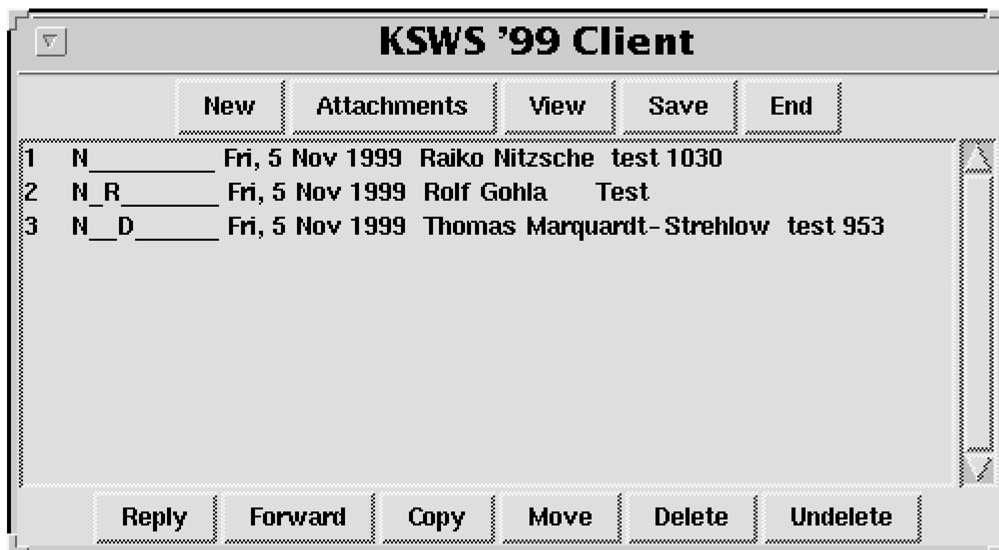

Abbildung 8: Mail–Liste

Abbildung 9: Subsubmenüpunkt Attachments

die Liste der Attachments einer E–mail die entsprechende Attachment–ID bestimmt, diese nach `/tmp/attm_cnt` exportiert und von dort aus weiterbearbeitet.

Fehlerbehandlung Aufgrund des hohen Programmieraufwands wird von der vorliegenden Implementierung nur eine minimale Fehlerbehandlung durchgeführt, so daß zumindest Oberflächen–eigene Fehlerquellen, wie z.B. abgebrochene Dialoge, berücksichtigt werden. Nicht überprüft werden hingegen die Existenz von Mail–Folder–Namen und Dateinamen, die systemspezifische Umgebung u.ä.

4.3.2 Erweiterungsmöglichkeiten

Die Client–Datei `client.py` besteht aus der Klasse "Hello". Die Bibliotheken `mblib.py` und `i4lib.py` stellen die Schnittstelle für Aufrufe der *DB2*–Funktionen bereit. Alle Fenster des Clients erben die Eigenschaften der Konstruktor–Funktion "__init__". Hier tritt das Problem auf, daß Titel, Icon–Name und andere Eigenschaften weiterer Fenster nicht neu definiert werden, was die Benutzerfreundlichkeit deutlich erhöhen würde. So könnten Fenster, wie z.B. die Listen, entweder den Namen des Nutzers oder Namen von Mail–Foldern bzw. E–mails als Titel übergeben bekommen.

Auch die Übergabe ausgewählter Mail–Folder, E–mails oder Attachments an Funktionen wie Öffnen, Reply u.a. wäre denkbar. Die Realisierung solcher Parameterübergaben scheiterte bisher

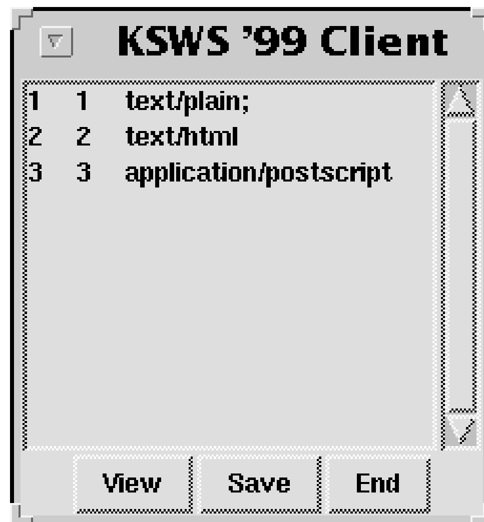

Abbildung 10: Attachment–Liste

lediglich am Programmieraufwand. Eine Erweiterung der Listen um eine Update–Funktion wäre ebenfalls denkbar. Zusätzlich könnten auch noch Operationen auf Flags angeboten werden — besonders für die nutzerdefinierten Flags.

4.4 Der Client im online–Betrieb

Neben der gebräuchlichen Art des Einsatzes als offline–Mail–Client mittels IMAP4 ist es außerdem möglich, direkt auf die Mail–Server–Datenbank zuzugreifen (s. Unterabschnitt 1.3). Wenn sich der Mail–Client auf einem Rechner mit ständiger Verbindung zum Mail–Server befindet, wäre es wenig sinnvoll, ständig per IMAP4 eine Synchronisation vorzunehmen. Andere Mail–Systeme, wie z.B. *Lotus Notes* [DS98], bieten die Möglichkeit, neben dem Zugriff auf die replizierte lokale Datenbank, im online–Betrieb auch auf die Server–Datenbank zuzugreifen.

Da das Datenbank–Schema von Server und Client das gleiche ist, funktionieren alle Datenbank-Zugriffsfunktionen auch für die Server–Datenbank. Alles was zu tun bleibt, ist die Einrichtung eines lokalen Datenbank–Clients, der die Verbindung zwischen dem Datenbank–Server und dem lokalen Rechner herstellt.

Wenn darüber hinaus noch der Name der Datenbank variabel, also vom Nutzer veränderbar, definiert wird, ist somit auch der Zugriff auf mehrere lokale oder entfernte Datenbanken möglich, d.h. mittels eines Mail–Clients ließen sich auf diese Weise mehrere Mail–Accounts verwalten. Dieses Prinzip von verschiedenen lokalen und serverbasierten Datenbanken wird z.B. von *Lotus Notes* eingesetzt.

5 Synchronisation von Client und Server

Eine grundlegende Forderung an den Mail–Client ist die Fähigkeit, seinen Datenbank–Inhalt mit dem des Mail–Servers abzugleichen. Diese Funktion ist notwendig, um disconnected, also ohne Verbindung zum Mail–Server (s. Unterabschnitt 1.3), arbeiten zu können. Während der Client im online–Betrieb ständig mit dem Server verbunden ist und die Mail–Folder auf dem Server direkt manipuliert, arbeitet der Client im offline– und disconnected–Betrieb dagegen selbständig

auf lokalen Kopien der Daten (Replikaten). Von Zeit zu Zeit müssen daher die Datenbestände von Server und Client abgeglichen werden. Diese Synchronisation erfolgt über die Verwendung normaler IMAP4–Kommandos.

Der E–mail–Datenbank–Client soll im disconnected–Betrieb arbeiten, um so z.B. auf einem Laptop oder auf einer Workstation zu Hause, die nicht ständig mit dem Internet verbunden sind, lauffähig zu sein. Dazu ist es natürlich notwendig, daß alle Aktionen sowohl auf Client– als auch auf der Server–Seite protokolliert werden, um sie bei gegebener Verbindung auf der Gegenseite nachholen zu können.

Die vollständige Beschreibung der Implementierung der Synchronisation von Client und Server findet sich in [KWR99, NMS99].

Im Folgenden beschreibt Unterabschnitt 5.1 den Aufbau und die Inhalte der Log–Dateien. Unterabschnitt 5.2 beschäftigt sich anschließend mit der Arbeitsweise der Synchronisation, bevor in Unterabschnitt 5.3 Probleme und Lösungsansätze diskutiert werden.

5.1 Die Log–Dateien

Vom E–mail–Datenbank–Client und dem E–mail–Datenbank–Server werden alle auf der jeweiligen Datenbank durchgeführten Aktionen in einer Log–Datei protokolliert. Sowohl Client als auch Server verfügen für diesen Zweck über eine Tabelle "logtable" (s. Anhang A), die zusätzlich in das Datenbank–Schema eingefügt wurde, um diese Protokoll–Einträge aufzunehmen.

In die Tabelle "logtable" schreiben dazu alle Aktionen, die Änderungsoperationen auf der Datenbank durchführen, einen Eintrag mit den dazugehörigen Daten. Die Tabelle ist folgendermaßen aufgebaut:

timestamp	**action**	**msg-id**	**oid**	**target**	**t_uid**	**source**	**s_uid**	**flags**

Die Einträge in der Tabelle haben folgende Bedeutung:

Timestamp ist der Zeitpunkt, zu dem die Aktion durchgeführt wurde. Serverseitig wird dieser Wert benötigt, um die richtigen Einträge aus der angeforderten Log–Datei zur Übertragung auszuwählen, da jeder Client nur die Einträge seit seiner letzten Synchronisation benötigt.

Action enthält das Schlüsselwort für die protokollierte Aktion. Als Schlüsselworte sind folgende Strings zugelassen:

E–mail–Aktionen:

- `APPEND` Anfügen einer E–mail an einen Mail–Folder
- `MOVE` Verschieben einer E–mail in einen anderen Mail–Folder
- `COPY` Kopieren einer E–mail in einen Mail–Folder
- `DELETE` Löschen einer E–mail aus einem bestimmten Mail–Folder (nicht aus der Datenbank)
- `FLAG_S` Ändern der System–Flags einer E–mail (in der "mail"–Tabelle)
- `FLAG_M` Ändern der Standard–Flags einer E–mail (in der "foldercontent"–Tabelle)

- `FLAG_U` Ändern der nutzerdefinierten Flags einer E–mail
  (in der "foldercontent"–Tabelle)

Mail–Folder–Aktionen:

- `CRTFOLDER` Anlegen eines Mail–Folders
- `DELFOLDER` Löschen eines Mail–Folders
- `RENFOLDER` Umbenennen eines Mail–Folders
- `FLAG_F` Ändern der Flags eines Mail–Folders

Msg-Id ist die Message–ID der bearbeiteten E–mail. Über die Msg-ID wird die E–mail eindeutig in unserem System identifiziert, damit Client und Server auch jeweils die gleiche E–mail ansprechen. Die UID des IMAP4–Protokolls (s. Unterabschnitt 1.3) ist dafür leider nicht geeignet, da Server und Client selbständig den UID–Zähler erhöhen. Bei längerer autonomer Arbeit laufen die UIDs daher auseinander und eine eindeutige Zuordnung wäre wieder nur über die Msg-ID möglich. Außerdem können sich die UIDs bei einer Neuordnung des Mail–Fodlers (z.B. Sortieren der E–mails in einem Mail–Folder) ändern, d.h. die Log–Datei–Einträge würden evtl. ungültige UIDs enthalten.

OID ist die physikalische Objektidentität einer E–mail in der "mail"–Tabelle. Dieser Wert wird zum einen als Mail–UID im virtuellen "rootmail"–Mail–Folder benötigt. Dieser Mail–Folder enthält alle E–mails, die physikalisch in der Datenbank enthalten sind, unabhängig von ihrer Zugehörigkeit zu tatsächlich existierenden Mail–Foldern ("rootmail"–View). In diesem Mail–Folder ist die Tabellen–OID einer E–mail gleichbedeutend der UID im IMAP4–System. Damit ist es möglich, eine E–mail zu übertragen, egal in welchem Mail–Folder sie sich zum Zeitpunkt der Synchronisation befindet.

Eine weitere Anwendung ergibt sich aus der Wartung der Log–Datei. Mit Hilfe der OID können Log–Datei–Einträge genau den betroffenen E–mails zugeordnet werden. Eine Anwendung wäre z.B. das Löschen aller Log–Datei–Einträge einer E–mail beim Entfernen selbiger aus der Datenbank (bis auf den "DELETE"–Eintrag selbst natürlich). So werden unnötige Log–Datei–Einträge entfernt und die "logtable" wird nicht so schnell zu groß.

Target bezeichnet den Mail–Folder, auf die sich die protokollierte Aktion bezieht. Es gibt daher mehrere Bedeutungen. Bei "CRTFOLDER" z.B. steht hier der Name des anzulegenden Mail–Folders. Als Regel gilt: Bei allen Aktionen, an denen nur ein Mail–Folder beteiligt ist, steht dessen Name in diesem Feld. Bei Aktionen mit zwei beteiligten Mail–Foldern steht hier der Zielort (wohin).

T_UID ist die UID der bearbeiteten E–mail im Ziel–Mail–Folder. Sie dient der einfacheren Identifizierung der E–mail z.B. beim Setzen von Flags. Hier wäre auch die Verwendung einer Kombination aus Msg-ID und Mail–Folder–Name möglich, was aber den benötigten Aufwand erhöhen würde.

Source bezeichnet, wie oben schon angedeutet, den Ausgangs–Mail–Folder (Quelle) bei binären Aktionen, wie z.B. "COPY" oder "MOVE". Bei allen anderen Aktionen ist dieses Feld leer.

S_UID enthält genau wie T_UID die aktuelle UID der bearbeiteten E–mail, nur in diesem Fall für den Mail–Folder, von dem die Aktion ausgeht (Quelle). Wie auch Source, ist dieses Feld nur bei binären Aktionen belegt.

Flags enthält die Flags für eine E–mail. Dieses Feld ist immer dann belegt, wenn die protokollierte Aktion in irgendeiner Weise den Flag–String einer E–mail schreibt. Am offensichtlichsten ist dies bei "FLAG_M", "FLAG_U" und "FLAG_S" der Fall. Aber auch Aktionen wie "APPEND" schreiben einen Flag–Wert. Im Falle von "CRTFOLDER" bezieht sich dieser Wert auf die Flags des erzeugten Mail–Folders.

5.2 Arbeitsweise der Synchronisation

Der E–mail–Datenbank–Server hält für die Synchronisation zwei virtuelle Mail–Folder bereit: "rootmail" und "logfile". "rootmail" bietet, wie bereits zuvor erwähnt, Zugriff auf alle physikalisch auf dem Server vorhandenen E–mails. Mittels "logfile" ist der Zugriff auf System–Informationen, wie die Log–Datei und den aktuellen Zeitstempel des Servers, möglich.

Virtuelle Mail–Folder liegen nicht physikalisch in der (Server–)Datenbank vor; trotzdem kann ein Client sie mittels "SELECT" anwählen und Operationen auf ihnen ausführen. Alle `db2mail`–Funktionen des Servers unterscheiden deshalb zwischen den beiden virtuellen Mail–Foldern und den in der Datenbank physikalisch existierenden Mail–Foldern.

Durch die beiden virtuellen Mail–Folder werden dem E–mail–Datenbank–Client die notwendigen Mittel zur Verfügung gestellt, um alle Log–Datei–Einträge ab einem bestimmten Zeitpunkt anzufordern. Dies ist für eine erfolgreiche Synchronisation unabdingbar.

Die Synchronisation der Datenbestände von Client und Server wird vollständig vom Client gesteuert vorgenommen. Als erstes wird vom Server dessen Log–Datei vom Zeitpunkt der letzten Synchronisation bis zum aktuellen Zeitpunkt angefordert. Dazu wird per "APPEND"–Kommando der Zeitstempel der letzten Synchronisation unter Verwendung des virtuellen Mail–Folders "logfile" an den Server übertragen.

```
1 append logfile () {26}
1999-11-03-15.52.49.516984
```

Der Server selektiert daraufhin die entsprechenden Log–Datei–Einträge aus seiner Datenbank und stellt diese virtuell als erste E–mail im "logfile"–Mail–Folder zum Abruf bereit. Mittels FETCH–Kommando werden dann diese Log–Datei–Einträge als zu übertragene E–mail vom Client angefordert.

```
1 select logfile
2 fetch 1 body[]
```

Genauer ist es so, daß immer wenn der "FETCH"–Befehl auf dem "logfile"–Mail–Folder ausgeführt wird, vom IMAP–Kern die Funktion `db2mail_text` aufgerufen wird, die dann eine E–mail mit den entsprechenden Log–Datei–Einträgen generiert.

Danach selektiert der Client aus seiner eigenen Datenbank die Log–Datei–Einträge und schreibt sie in eine lokale Datei.

Im nächsten Schritt wird nun die Server–Log–Datei inkrementell Schritt für Schritt abgearbeitet und somit alle Aktionen, die seit der letzten Synchronisation auf dem Server stattgefunden haben, auch auf die lokale Datenbank des Clients angewendet. Beim Auftreten eines "APPEND"–Eintrags in der Log–Datei wird die dazugehörige E–mail via IMAP4–Protokoll aus dem virtuellen Mail–Folder "rootmail" geladen und in die Client–Datenbank eingetragen. Alle diese Aktionen, die der Synchronisation dienen, erzeugen auf dem Client keine Log–Einträge, da

sie ja bereits auf Server–Seite ausgeführt wurden und es nicht sinnvoll ist, sie bei der nächsten Synchronisation erneut auf dem Server durchzuführen.

Danach wird die Log–Datei des Clients abgearbeitet. Wieder wird Schritt für Schritt jede Datenbank–Aktion in einen IMAP4–Aufruf umgesetzt und an den Server geschickt, der diese dann auf seinem Datenbestand ausführt. Eine neu angefügte E–mail auf Client–Seite, z.B. ein Draft oder eine E–mail im "Sent"–Mail–Folder, wird einfach mittels dem IMAP4–Befehl "APPEND" an den entsprechenden Server–Mail–Folder angefügt. Bei diesem Vorgang protokolliert der Server allerdings weiterhin alle Aktionen in seiner Log–Datei, da diese Änderungen auch auf weiteren Clients ausgeführt werden müssen, wenn diese sich mit dem Server synchronisieren wollen.

Kein Client benötigt bei jeder Synchronisation sämtliche vorhandenen Log–Datei–Einträge, die auf dem Server gespeichert sind, sondern jeder Client benötigt immer nur diejenigen Einträge, die seit seiner letzten Synchronisation protokolliert wurden. Es gibt daher die Möglichkeit, den Zeitstempel des Servers an den Client zu übertragen. Zum Abschluß der Synchronisation fordert der Client den aktuellen Zeitstempel des Servers an, der immer als zweite E–mail des virtuellen Mail–Folders "logfile" bereitsteht, und daher mit einem einfachen "FETCH"–Kommando ermittelt werden kann:

```
1 select logfile
2 fetch 2 body[]
```

Der so übertragene Zeitstempel des Servers wird auf dem Client in ein internes Register eingetragen (Tabelle "lastrepl"). Mit diesem neuen Zeitstempel wird dann die nächste Synchronisation durchgeführt, indem dieser zu Beginn der nächsten erfolgenden Synchronisation wiederum dem Server zur Selektion der zu übertragenden Log–Datei–Einträge mitgeteilt wird, ...

Es ist wichtig, daß der Zeitstempel des Servers und nicht ein Zeitstempel des Clients gespeichert wird, da nicht immer sichergestellt werden kann, daß die Systemzeit auf beiden Seiten genau gleich ist. Auf diese Weise wird somit verhindert, daß bei der nächsten Synchronisation eigene Aktionen (des Clients) erneut auf diesem ausgeführt werden.

5.3 Probleme und Lösungen

Bei der in Unterabschnitt 5.2 beschriebenen Art der Synchronisation von Client und Server treten einige Probleme auf, die im Folgenden beschrieben werden. Gegebenenfalls wird gleich eine Lösungsvariante vorgeschlagen.

Ein wichtiges, hier nicht behandeltes Problem ist das Sperren des Servers während der Synchronisation. Greift ein weiterer Client während der Synchronisation auf den Server zu oder wird eine neue E–mail eingetragen, dann gehen diese Einträge dem gerade synchronisierenden Client verloren, da dieser ja erst nach abgeschlossener Synchronisation den Zeitstempel des Servers anfordert. Vorher kann dieser Zeitstempel jedoch nicht angefordert werden, da ansonsten bei einer nachfolgenden Synchronisation der Client sein eigenes Update erneut "vorgesetzt" bekäme. Folglich darf also auf dem Server keine Vermischung von Log–Datei–Einträgen einer Synchronisation mit anderen Log–Datei–Einträgen stattfinden.

Ein weiteres Problem ist das unendliche Anwachsen der Log–Datei–Einträge. Da die Clients anonym (nicht registriert) sind, kann der Server nicht entscheiden, welche Log–Datei–Einträge nicht mehr benötigt werden, z.B. weil sich alle Clients bis zu einem gewissen Zeitpunkt schon synchronisiert haben. Außerdem kann jederzeit ein neuer Client hinzukommen, der den aktuellen

Zustand von Mail–Foldern benötigt. Beide Problemfälle sind allerdings mit kleinen Erweiterungen des vorgestellten Systems schnell zu lösen.

Der einfachere Fall ist das Hinzukommen eines neuen Clients. Hier kann leicht eine Log–Datei generiert werden, die den aktuellen Datenbank–Zustand widerspiegelt. Da der Client noch keine Daten besitzt, können alle Änderungseinträge für bestehende Daten ignoriert werden. Die generierte Log–Datei enthält dann nur Einträge der Art "CRTFOLDER" und "APPEND".

Der kompliziertere Fall ist das Herausfinden des ältesten Synchronisations–Zeitpunktes. Hierzu ist es notwendig, daß der Server alle Clients kennt. Zum einen kann sich jeder Client einen eindeutigen Schlüssel generieren, mit dem er sich beim Server zum Synchronisieren anmeldet. Einfacher ist es jedoch, diesen Schlüssel vom Server generieren zu lassen, was zudem auf jeden Fall die Eindeutigkeit garantiert. Zu unserem Datenbank–Schema müßte dazu eine weitere Tabelle mit zwei Attributen hinzugefügt werden:

client-ID	timestamp

Der Server speichert in dieser Tabelle dann alle angemeldeten Clients mit ihrem letzten Synchronisations–Zeitpunkt. Die Clients haben daher jeweils nur einen Eintrag. Die Client-ID kann z.B. einfach eine Zahl sein. Mit Hilfe dieser Information kann nun der Server den ältesten Synchronisations–Zeitpunkt feststellen und alle Log–Datei–Einträge bis zu diesem Zeitpunkt löschen. Zusätzlich werden durch dieses Vorgehen noch weitere Optimierungen möglich. Liegen bei einer E–mail z.B. "APPEND", diverse Änderungsaktionen und schließlich "DELETE" alle innerhalb der Zeitspanne nach der letzten Synchronisation, so kann der Server alle zu dieser E–mail gehörenden Log–Datei–Einträge vor der Übertragung zu dem entsprechenden Client entfernen. Dies kann die Synchronisations–Dauer unter Umständen erheblich verkürzen.

Meldet sich ein Client zum ersten Mal zur Synchronisation, sendet er eine "Dummy-ID", z.B. 0. Der Server gibt bei der Bestätigung zur Synchronisation dann die neue ID für diesen Client aus und generiert außerdem eine spezielle Log–Datei, die zuvor bereits skizziert wurde. Mit diesem Prinzip können theoretisch beliebig viele Clients von einem Server bedient werden.

Zur Sicherheit kann noch eine Zeitschranke vergeben werden, die angibt, nach welcher Zeit ein Client aus der Liste entfernt wird. Dieser Client würde dann bei einer neuen Kontaktaufnahme eine neue ID und dazu den aktuellen Datenbank–Zustand bekommen. Auf diese Weise kann sichergestellt werden, daß ein Client, der sich nur einmal synchronisiert hat, nicht das ganze System ausbremst. Allerdings muß in diesem Szenario natürlich darauf geachtet werden, daß gelöschte Client-IDs nicht erneut vergeben werden dürfen, damit es nicht zu Überschneidungen zwischen mehreren Clients kommt.

Um all diese Funktionen zu ermöglichen, müßte ein komplettes Synchronisations–Protokoll implementiert werden, das mindestens folgende Punkte umfaßt:

- Anmeldung zur Synchronisation eines Clients beim Server mit einer Client–Identifikation
- der Server sendet die Client-ID und "OK", wenn er nicht schon einen anderen Client bedient
  (die Client-ID ist die ID, mit der sich der Client angemeldet hat bzw. die ihm neu zugewiesen wurde)
- der Client führt die Synchronisation durch; der Server sperrt solange alle anderen Zugriffe (dies umfaßt die Übertragung der Log–Datei, die Ausführung der IMAP4–Operationen und das Lesen des Server–Zeitstempels)

- der Client meldet sich beim Server ab; der Server aktualisiert den Zeitstempel in der Client–Tabelle und gibt Zugriffe wieder frei

Die anderen Probleme betreffen jeweils die erweiterten Flags in unserem Datenbank–Schema.

Die System–Flags sind jeder physikalischen E–mail genau einmal zugeordnet, also unabhängig von dem Mail–Folder, in dem diese steht. Änderungen der Flags werden zwar protokolliert, können aber nur in Richtung Server → Client übertragen werden, da hierbei die komplette Log–Datei geschickt wird. In der Richtung Client → Server wird das IMAP–Protokoll zur Synchronisation benutzt, welches keine entsprechende Funktion dafür bietet. Eine Lösung wäre die Festlegung spezieller Flag–Worte, die dann vom Server interpretiert werden müßten, um sie jeweils in den richtigen Attributen zu speichern. Diese speziellen Flag–Worte könnten dann allerdings nicht mehr für die normale E–mail verwendet werden.

Eine elegantere Lösung wäre es, ein Setzen der Mail–Flags im "rootmail"–Mail–Folder als Setzen der System–Flags zu interpretieren. Damit blieben alle Worte frei verfügbar. Die einfachste Lösung, die hier auch verwirklicht wurde, ist die, daß System–Flags nur für systeminterne und Mail–Daten–abhängige Informationen genutzt werden und nicht auf den Client bzw. Server übertragen werden müssen. Zur Zeit werden die System–Flags nicht benutzt, sie stehen lediglich für neue Features zur Verfügung.

Ein ähnliches Problem ergibt sich mit den Nutzer–Flags. Da auf IMAP4–Ebene keine Trennung zwischen Standard–Flags und nutzerdefinierten Flags gemacht wird, bleibt nur die Interpretation der Werte. Allerdings gibt es keine Vorschrift, wie so ein Nutzer–Flag aussieht und wie es im "u_flags"–Attribut gespeichert werden soll. Die Richtung Server → Client ist wieder sehr einfach, da hier der entsprechende Flag–String an den Client geschickt wird, der ihn nur in die Datenbank eintragen muß. Für dieses Problem gibt es momentan keine Lösungsvorschläge. Wie auch bei den System–Flags werden die Nutzer–Flags einfach ignoriert. Sie stehen den entsprechenden Systemen daher zur freien Verfügung.

6 Zusammenfassung und Ausblick

Wenn E–mails als semistrukturierte Daten gesehen werden, bietet es sich für ihre Verwaltung sicherlich an, eine Datenbank–Lösung zumindest in Betracht zu ziehen. Auch weitere Probleme bei der Verwaltung von E–mails, wie die in herkömmlichen Mail–Systemen nur unzureichend verfügbare Anfrageunterstützung und die Frage nach der Behandlung von Replikaten, sind im Datenbankbereich erforscht, so daß sich eine umfassende Datenbank–Lösung geradezu anbietet.

Eine erste prototypische Implementierung einer E–mail–Verwaltung auf Basis des objektrelationalen DBMS *DB2* wurde im Sommersemester 1999 im Rahmen einer KSWS-Lehrveranstaltung des Lehrstuhls Datenbank– und Informationssysteme am Fachbereich Informatik der Universität Rostock durch studentische Projekte realisiert. Die Implementierungen bauen auf verschiedenen, frei verfügbaren Quellcode–Bibliotheken zu IMAP4 auf und nutzen unterschiedlichste Programmier– und Skript–Sprachen: So wurden neben einem Mail–Parser in *Perl* ein IMAP4–Mail–Server in *C* unter Nutzung von Embedded–SQL sowie ein IMAP4–Mail–Client in *Python* umgesetzt. Sowohl *Perl* als auch *Python* verwenden dabei den Kommandozeileninterpreter (CLI) von *DB2*. Eine rudimentäre Oberfläche für einen Mail–Client wurde mittels *Tk* realisiert. Die notwendige Synchronisation von Client und Server, die grundlegende Teile des Problems der Replikat–Verwaltung löst, wird durch Log–Dateien auf Client– und Server–Seite unterstützt. Bei Aufbau einer Verbindung zwischen Client und Server wird dazu zunächst die

Log–Datei des Servers zum Client übertragen, bevor alle protokollierten Änderungen inkrementell auf der jeweils anderen Plattform nachgeholt werden.

Aufgrund des gewählten heterogenen Programmier–Ansatzes, speziell der Verwendung unterschiedlicher Skript–Sprachen, der teilweise erforderlichen Übertragung von Daten zwischen verschiedenen Rechnern und der Übergabe von Daten unter Verwendung von Dateien erreicht der geschaffene Prototyp nicht die Performance, die bei der ausschließlichen Verwendung von nur einer Programmiersprache sicherlich erreicht werden könnte. Ausgehend von der grundlegenden Betrachtung des Prototypen als Machbarkeitsstudie sind unsere Erwartungen jedoch erfüllt worden. Durch die Investition von mehr Ressourcen, als im beschränkten Rahmen einer Lehrveranstaltung verfügbar sind, wird potentiell eine E–mail–Verwaltung auf Basis objektrelationaler Datenbank–Technologie möglich, die den Nutzer von E–mail–Funktionalität in bisher nicht erreichter Weise — gerade auch im mobilen Umfeld — unterstützt. Der in diesem Artikel beschriebene Prototyp zeigt damit, daß die in [KH99] skizzierte Vision der Verschmelzung von E–mail– und Datenbank–Funktionalität *emailDB* eine realistische Möglichkeit zur Lösung wesentlicher Probleme mobiler Nutzer von E–mail–Funktionalität darstellt.

Weitergehende Untersuchungen zur gezielten Replikation von Teil–Datenbeständen, sowie eine parallel dazu erfolgende prototypische Implementierung einer E–mail–Verwaltung auf Basis des objektrelationalen DBMS *Informix* bilden die beiden Schwerpunkte einer — unmittelbar im Anschluß an die beendete KSWS–Lehrveranstaltung des Sommersemesters 1999 gestarteten — KSWS–Lehrveranstaltung im Wintersemester 1999/2000. Darüber hinaus wird die zuvor entwickelte Oberfläche des Mail–Clients in diesem Rahmen vervollständigt und erweitert.

Literatur

[Aal99] Aalto, J.: *Procmail-Hints*, April 1999. URL: http://www.procmail.org/jari/pm-tips.txt.

[Aas98] Aas, G.: *Perl-Modul: MIME-Base64-2.11*, April 1998. URL: http://www.cpan.org/modules/MIME-Base64-2.11.tar.gz.

[Bar98] Barr, G.: *Perl-Modul: MailTools-1.13*, 1998. URL: http://www.cpan.org/modules/MailTools-1.13.tar.gz.

[BS99] Bruder, I.; Schneider, L.: *Komplexe Software–Systeme — Electronic–Mail–Verwaltung auf objektrelationalen Datenbanksystemen, Gruppe: Parser.* Lehrstuhl Datenbank– und Informationssysteme, Fachbereich Informatik, Universität Rostock, November 1999. Bericht zur KSWS–Veranstaltung im Sommersemester 1999, URL: http://wwwdb.informatik.uni-rostock.de/Lehre/Vorlesungen/skripte/ksws-1999/parser.ps.gz.

[Cha98] Chamberlin, D.: *A Complete Guide to DB2 Universal Database.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1998.

[Cri96] Crispin, M.: *RFC 2060: Internet Message Access Protocol — Version 4rev1*, Dezember 1996. URL: http://www.imap.org/docs/rfc2060.html.

[Cri98] Crispin, M.: *IMAP4rev1/c-client Development Environment*, September 1998. University of Washington, Computing & Communications, Anonymous FTP Server, ftp://ftp.cac.washington.edu/mail/imap.tar.Z, permanently updated.

[Cro82] Crocker, D.: *RFC 822: Standard for the Format of ARPA Internet Text Messages*, August 1982. URL: http://www.isi.edu/in-notes/rfc822.txt.

[DS98] Dierker, M.; Sander, M.: *Lotus Notes 4.6 und Domino — Integration von Groupware und Internet.* Addison–Wesley, Bonn, 1998.

[Ery99a] Eryq: *Perl-Modul: IO-stringy-1.207*, 1999. URL: http://www.cpan.org/modules/IO-stringy-1.207.tar.gz.

[Ery99b] Eryq: *Perl-Modul: MIMETool-4.124*, 1999. URL: http://www.cpan.org/modules/MIME-tools-4.124.tar.gz.

[FB96a] Freed, N.; Borenstein, N.: *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*, November 1996. URL: http://www.ietf.org/rfc/rfc2045.txt.

[FB96b] Freed, N.; Borenstein, N.: *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, November 1996. URL: http://www.ietf.org/rfc/rfc2046.txt.

[FB96c] Freed, N.; Borenstein, N.: *RFC 2049: Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*, November 1996. URL: http://www.ietf.org/rfc/rfc2049.txt.

[FKP96] Freed, N.; Klensin, J.; Postel, J.: *RFC 2048: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures*, November 1996. URL: `http://www.ietf.org/rfc/rfc2048.txt`.

[Gra95a] Gray, T.: *Comparing Two Approaches to Remote Mailbox Access: IMAP vs. POP*, September 1995. URL: `http://www.imap.org/imap.vs.pop.brief.html`.

[Gra95b] Gray, T.: *Message Access Paradigms and Protocols*, September 1995. URL: `http://www.imap.org/imap.vs.pop.html`.

[HM99] Himstedt, T.; Mätzel, K.: *Mit Python programmieren : Einführung und Anwendung skriptorientierter Programmierung.* dpunkt–Verlag, Heidelberg, 1999.

[IBM98] IBM: *DB2 Universal Database*, 1998. Online-Bücher, <DB2-Installation>/sqllib/doc/de/html/index.htm.

[KFR+95] Klensin, J.; Freed, N.; Rose, M.; Stefferud, E.; Crocker, D.: *RFC 1869: SMTP Service Extensions*, November 1995. URL: `http://www.ietf.org/rfc/rfc1869.txt`.

[KH99] Kröger, J.; Heuer, A.: Verwaltung von E–mails in Datenbanken? — emailDB! In: *11. Workshop "Grundlagen von Datenbanken", Jenaer Schriften zur Mathematik und Informatik, Nr. 99/16*, S. 62–66. Friedrich–Schiller–Universität Jena, Mai 1999.

[KSW99] KSWS'99: *Komplexe Software–Systeme — Electronic–Mail–Verwaltung auf objektrelationalen Datenbanksystemen.* Lehrstuhl Datenbank– und Informationssysteme, Fachbereich Informatik, Universität Rostock, November 1999. Quellcode der Implementierung zur KSWS–Veranstaltung im Sommersemester 1999, URL: `http://wwwdb.informatik.uni-rostock.de/Lehre/Vorlesungen/skripte/ksws-1999/ksws-IMAP.tar.gz`.

[KWR99] Knorr, S.; Waschk, G.; Rzehak, M.: *Komplexe Software–Systeme Electronic–Mail Verwaltung auf objektrelationalen Datenbank–Systemen — Entwicklung eines IMAP4rev1–kompatiblen Servers mit Anbindung an DB2.* Lehrstuhl Datenbank– und Informationssysteme, Fachbereich Informatik, Universität Rostock, November 1999. Bericht zur KSWS–Veranstaltung im Sommersemester 1999, URL: `http://wwwdb.informatik.uni-rostock.de/Lehre/Vorlesungen/skripte/ksws-1999/server.ps.gz`.

[Lut96] Lutz, M.: *Programming Python.* O'Reilly Ass., New York, NY, 1996.

[Moo96] Moore, K.: *RFC 2047: Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non–ASCII Text*, November 1996. URL: `http://www.ietf.org/rfc/rfc2047.txt`.

[MR96] Myers, J.; Rose, M.: *RFC 1939: Post Office Protocol — Version 3*, Mai 1996. URL: `http://www.imap.org/docs/rfc1939.html`.

[NMS99] Nitzsche, R.; Marquardt-Strehlow, T.: *Mailclient mit Datenbankunterstützung und Serversynchronisation über IMAP4.* Lehrstuhl Datenbank– und Informationssysteme, Fachbereich Informatik, Universität Rostock, November 1999. Bericht zur KSWS–Veranstaltung im Sommersemester 1999, URL: `http://wwwdb.informatik.uni-rostock.de/Lehre/Vorlesungen/skripte/ksws-1999/client.ps.gz`.

[Pos82] Postel, J.: *RFC 821: Simple Mail Transfer Protocol*, August 1982. URL: `http://www.isi.edu/in-notes/rfc821.txt`.

[SH99] Saake, G.; Heuer, A.: *Datenbanken: Implementierungstechniken.* MITP–Verlag, Bonn, 1999.

[Sri97] Srinivasam, S.: *Advanced Perl Programming — Foundations and Techniques for Perl Application Developers.* O'Reilly Ass., Cambridge, 1997.

[SSP99] Siever, E.; Spainhour, S.; Patwardhan, N.: *Perl in a Nutshell — A Desktop Quick Reference.* O'Reilly Ass., Cambridge, 1999.

[vLF97] Löwis, M. v.; Fischbeck, N.: *Das Python–Buch — Referenz der objektorientierten Skriptsprache für GUIs und Netzwerke.* Addison–Wesley, Bonn, 1997.

[WvRA96] Watters, A.; Rossum, G. v.; Ahlstrom, J.: *Internet Programming with Python.* M&T Books, Sebastopol, CA, first edition. Auflage, 1996.

A Der Datenbank–Entwurf

In Anhang A.1 wird das Datenbank–Schema eingeführt und erläutert, bevor es in Anhang A.2 anhand von SQL–Anweisungen dargestellt wird.

A.1 Erläuterung des Schemas

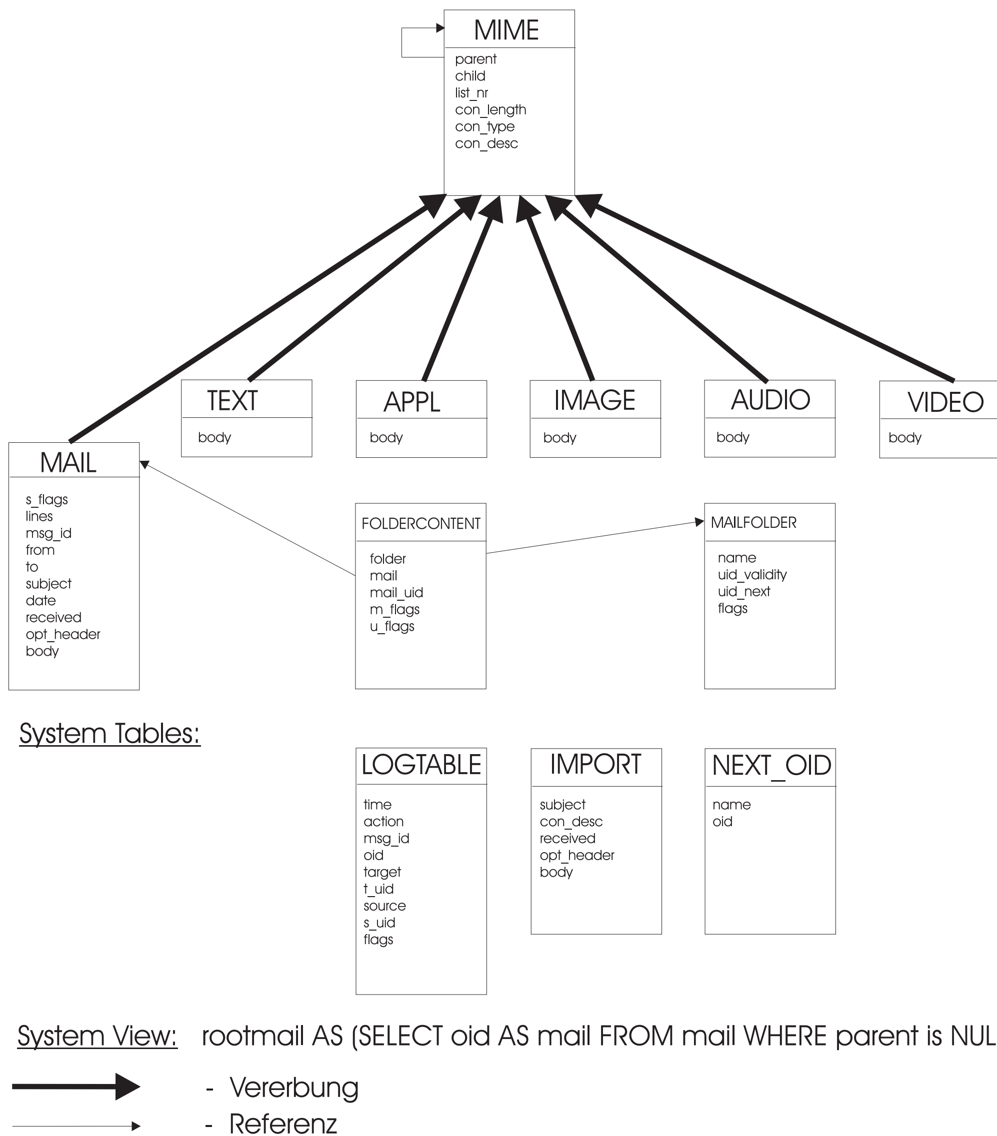

Abbildung 11: Das Datenbank–Schema

Wie im Datenbank–Schema zu sehen ist, haben wir eine logische Trennung der E–mail in Header– und Body–Bestandteile (MIME–Teile) vorgenommen. Aus Effizienz– und Sicherheitsgründen werden alle Tabellen für jeden Nutzer separat angelegt. Einzelheiten zu Datentypen finden sich in Anhang A.2.

Die Header–Informationen werden in der Tabelle "mail" aufgenommen. Alle Attribute dieser Tabelle entsprechen E–mail–Bestandteilen gemäß RFC 822 [Cro82]. In der Tabelle werden die E–mail–Attribute "Lines", "Message-ID", "From", "To", "Subject", "Date", "Received" und "Body" gespeichert. Die "Received"–Zeilen einer E–mail werden in einem CLOB gespeichert, da dieser Teil ziemlich groß werden kann. Falls die E–mail eine nach RFC 822 definierte E–mail ohne MIME–Bestandteile [FB96a, FB96b, Moo96, FKP96, FB96c] ist, so wird ihr "Body", also die Text–Nachricht, ebenfalls in einem CLOB in dieser Tabelle gespeichert. Andernfalls ist die E–mail eine MIME–Mail, und die Body–Bestandteile werden als MIME–Bestandteile in nachfolgend beschriebenen Tabellen gespeichert. Die Begrenzung der CLOB–Speichergröße auf ein Megabyte beruht auf der Tatsache, daß der Mail–Server am Fachbereich Informatik der Universität Rostock zur Zeit nur E–mails von kleiner gleich einem Megabyte weiterreicht. Unter "opt_header" werden alle restlichen und die optionalen Header–Bestandteile, wie "X-"beliebige Erweiterungen und auch andere, Nicht–RFC 822–Bestandteile gespeichert, damit diese Informationen nicht verloren gehen. Alle E–mails können daher vollständig in der Datenbank gespeichert werden. Zusätzlich werden in dieser Tabelle vom System generierte Flags ("s_flags") gespeichert.

In der Tabelle "mime" werden die MIME–Attribute ("Content-"-Attribute) sowie Informationen über die Reihenfolge und Hierarchie der MIME–Bestandteile gespeichert. Die Reihenfolge innerhalb einer Hierarchie–Ebene bestimmt das Attribut "list_nr". Die Hierarchie kann durch die Attribute "child" und "parent" zurückgewonnen werden. In "child" steht dabei die Anzahl der untergeordneten Mail–Teile. Umgekehrt referenziert das Attribut "parent" die MIME–Bestandteile der höheren Hierarchie.

In den Tabellen "text", "appl", "image", "audio" und "video" werden die entsprechenden MIME–Nachrichten bzw. Body–Bestandteile gespeichert. In der vorliegenden Implementation sind die MIME–Bestandteile nicht dekodiert, sondern werden im "Base64"–Format abgespeichert. Bei einer dekodierten Speicherung ist es möglich, die in der Datenbank *DB2* implementierten Extender für Text etc. zu verwenden. Zu beachten ist aber, daß bei dekodierter Speicherung der IMAP4–Server bei jeder Anfrage die MIME–Bestandteile wieder kodiert an die Mail–Clients weiterreichen muß, um IMAP4–kompatibel zu bleiben, und somit zusätzlich belastet wird. In der Tabelle "appl" werden neben den MIME–Bodies vom Typ "application" auch alle noch nicht definierten und nicht identifizierbaren MIME–Bestandteile der E–mails gespeichert. Für die Beschränkung der Größe der einzelnen MIME–Bestandteile auf ein Megabyte ist ebenfalls die am Fachbereich Informatik der Universität Rostock derzeit bestehende Mail–Server–Einstellung ausschlaggebend.

In der Tabelle "mailfolder" werden die Namen und Attribute der benutzten Mail–Folder gespeichert. Über die Tabelle "foldercontent" wird die Zuordnung von E–mails zu dem bzw. den Mail–Foldern festgelegt. Außerdem werden im Attribut "m_flags" die standardisierten Mailflags gespeichert.

Zu den bisher aufgeführten Tabellen werden noch die folgenden Systemtabellen benötigt:

Die "import"–Tabelle wird zum Einfügen der CLOBs benötigt und wurde bei der Dokumentation des Parsers in Abschnitt 2 erläutert. Die "next_oid"–Tabelle dient dem Generieren der Mail–OIDs bzw. dem Speichern der nächsten OID und wurde in Unterabschnitt 1.4 erläutert. Die "logtable"–Tabelle wird zur Steuerung von Synchronisation und Replikation genutzt und wurde daher in Abschnitt 5 genauer beschrieben.

Über die Sicht "rootmail" werden alle "rootmail"s ausgegeben. Eine "rootmail" ist dabei eine E–mail, welche in der Hierarchie–Ebene ganz oben steht, d.h. ihr "parent"–Attribut ist leer. Eine "rootmail" kann zwar andere E–mails enthalten, ist aber niemals selbst in einer E–mail enthalten.

A.2 Das Schema als SQL–Anweisungen

```
-- Create the MIME types --


-- OID string: "mim" --
CREATE TYPE mime_t AS
       (
       parent       REF(mime_t),
       -- child: = number of children
       child        INTEGER,
       list_nr      INTEGER,
       con_length   INTEGER,
       con_type     VARCHAR(128),
       con_desc     CLOB(1k)
       )

-- OID string: "rfc" --
CREATE TYPE mail_t UNDER mime_t AS
       (
       -- system flags (replicated, just headers, ...)
       s_flags      CHAR(10),
       lines        INTEGER,
       msg_id       VARCHAR(254),
       from         VARCHAR(254),
       to           VARCHAR(254),
       subject      VARCHAR(254),
       date         VARCHAR(128),
       received     CLOB(1k),
       opt_header   CLOB(1k),
       -- max size of a mail
       body         CLOB(1M)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;

-- OID string: "txt" --
CREATE TYPE text_t UNDER mime_t AS
       (
       body         CLOB(1M)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;

-- OID string: "app" --
CREATE TYPE appl_t UNDER mime_t AS
       (
       body         CLOB(1M)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;
```

```
-- OID string: "img" --
CREATE TYPE image_t UNDER mime_t AS
       (
       body         CLOB(1M)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;

-- OID string: "aud" --
CREATE TYPE audio_t UNDER mime_t AS
       (
       body         CLOB(1M)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;

-- OID string: "vid" --
CREATE TYPE video_t UNDER mime_t AS
       (
       body         CLOB(1M)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;


-- Create system types --


-- OID string: "f" --
CREATE TYPE mailfolder_t AS
       (
       name         VARCHAR(254),
       uid_validity BIGINT,
       uid_next     BIGINT,
       -- flags are defined as follows:
       -- Byte      Value Meaning
       -- 0         N Noinferiors flag
       -- 1         M Marked
       -- 2         U Unmarked
       -- 3         S Subscribed
       -- 4         Separator
       flags        CHAR(5)
       )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;

-- OID string: "fc" --
CREATE TYPE foldercontent_t AS
       (
       folder       REF(mailfolder_t),
       mail         REF(mail_t),
       mail_uid     BIGINT,
       -- standard mail flags:
       -- flags are defined as follows:
       -- flag      Meaning
       -- N         new mail
       -- O         old mail
```

```
        -- R          replied mail
        -- D          deleted mail
        -- F          forwarded mail
        -- M          mime mail
        m_flags       CHAR(10),
        -- user flags: __________
        u_flags       CHAR(10)
        )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;



-- Create administration types --


-- needed for a workaround for the "IMPORT" command
CREATE TYPE import_t AS
        (
        subject       VARCHAR(254),
        con_desc      CLOB(1K),
        received      CLOB(1K),
        opt_header    CLOB(1K),
        body          CLOB(1M)
        )
WITHOUT COMPARISONS NOT FINAL MODE DB2SQL;



-- Create the MIME tables --


CREATE TABLE mime OF mime_t (REF IS oid USER GENERATED,
        parent WITH OPTIONS SCOPE mime);

CREATE TABLE mail OF mail_t UNDER mime INHERIT SELECT PRIVILEGES;
CREATE TABLE text OF text_t UNDER mime INHERIT SELECT PRIVILEGES;
CREATE TABLE appl OF appl_t UNDER mime INHERIT SELECT PRIVILEGES;
CREATE TABLE image OF image_t UNDER mime INHERIT SELECT PRIVILEGES;
CREATE TABLE audio OF audio_t UNDER mime INHERIT SELECT PRIVILEGES;
CREATE TABLE video OF video_t UNDER mime INHERIT SELECT PRIVILEGES;



-- Create system tables and views --


CREATE TABLE mailfolder OF mailfolder_t (REF IS oid USER GENERATED);

CREATE TABLE foldercontent OF foldercontent _t (REF IS oid USER GENERATED,
        folder WITH OPTIONS SCOPE mailfolder, mail WITH OPTIONS SCOPE mail);

CREATE VIEW rootmail AS (select oid AS mail FROM mail WHERE parent is NULL);
```

```
-- Create administration tables --


CREATE TABLE import OF import_t (REF IS oid USER GENERATED);

CREATE TABLE next_oid
       (
       name         CHAR(20),
       -- VALUES for name: mime, mail, text, appl, image, audio,
       -- video, mailfolder, foldercontent, uid_validity
       oid          BIGINT
       );

CREATE TABLE logtable
       (
       time         TIMESTAMP,
       action       CHAR(10),
       msg_id       VARCHAR(254),
       oid          VARCHAR(16) FOR BIT DATA,
       target       VARCHAR(254),
       t_uid        BIGINT,
       source       VARCHAR(254),
       s_uid        BIGINT,
       flags        CHAR(10)
       );
```

B Ein Beispiel Datenbank–Skript

Dieses Datenbank–Skript wurde automatisch beim Parsing einer E–mail generiert und wird nach dem Einfügen dieser E–mail in die Datenbank automatisch wieder gelöscht. Weitere Einzelheiten sind im Unterabschnitt 2.2 bei der Beschreibung des Einfügevorgangs von E–mails in die Datenbank erläutert.

```
#! /bin/sh
```

`###` Setzen der *DB2*–Umgebung.

```
# Default DB2 product directory
DB2DIR="/opt/IBMdb2/V5.0"

#-----------------------------------------------------------------------
# DB2INSTANCE [Default null, values: Any valid instance name]
# Specifies the instance that is active by default.
#-----------------------------------------------------------------------
DB2INSTANCE=db2inst1
export DB2INSTANCE

INSTHOME=/export/home/db2inst1

#-----------------------------------------------------------------------
# Add the directories:
# INSTHOME/sqllib/bin - database executables
# INSTHOME/sqllib/adm - sysadm executables
# INSTHOME/sqllib/misc - miscellaneous utilities
# to the user's PATH.
#-----------------------------------------------------------------------
PATH=$PATH:$INSTHOME/sqllib/bin:$INSTHOME/sqllib/adm
PATH=$PATH:$INSTHOME/sqllib/misc
export PATH
if [ -f /export/home/db2inst1/db2tx/db2txprofile ]; then
. /export/home/db2inst1/db2tx/db2txprofile
fi
```

`###` Setzen des Arbeitspfades.

```
cd ~/.smt/temp/.19990708205313.A530@snoopy./
```

`###` Mit der Datenbank verbinden.

```
db2 connect to ksws99 > dblog 2>&1
```

```
### Ausschalten des automatischen Commits.

db2 "UPDATE COMMAND OPTIONS USING c OFF" >> dblog 2>&1


### OID für E–Mail–Teil auslesen und nächste OID setzen.

db2 "SELECT oid FROM next_oid WHERE name = 'mail'" > next_oid
oid=`awk 'NR == 4  print $1 ' next_oid`
oid1="mail_t('rfc"$oid"')"
db2 "UPDATE next_oid SET oid = oid + 1 WHERE name = 'mail'" >> dblog 2>&1


### OID für folder_content-Eintrag, nur bei Hierarchie–Ebene eins.

db2 "SELECT oid FROM next_oid WHERE name = 'foldercontent'" > next_fc
fcoid=`awk 'NR == 4  print $1 ' next_fc`
db2 "UPDATE next_oid SET oid = oid + 1 WHERE name = 'foldercontent'" >> dblog 2>&1


### UID für E–Mail in Mail–Folder, nur bei Hierarchie–Ebene eins.

db2 "SELECT uid_next FROM mailfolder WHERE name = 'INBOX'" > next_uid
uid=`awk 'NR == 4  print $1 ' next_uid`
db2 "UPDATE mailfolder SET uid_next = uid_next + 1 WHERE name = 'INBOX'" >> dblog
2>&1


### Trage E–Mail in Tabelle folder_content ein.

db2 "INSERT INTO foldercontent ( oid, mail, mail_uid, m_flags, u_flags ) VALUES
( foldercontent_t('fc$fcoid'), $oid1, $uid, '_OR_____', '________' )" >> dblog 2>&1
db2 "UPDATE foldercontent SET folder = ( SELECT oid FROM mailfolder WHERE name =
'INBOX' ) WHERE oid = foldercontent_t('fc$fcoid')" >> dblog 2>&1


### Trage alle einfachen Attribute des Mail–Teiles in die Datenbank ein.

db2 "INSERT INTO mail (oid, list_nr,con_type,lines,msg_id,from,to,date,con_length,
opt_header, child) VALUES ($oid1, 1,'multipart/mixed; boundary=liOOAslEiF7prFVr', 148,
'<19990708205313.A530@snoy>','<ilr@@>','<ilr@snoy.hole.de>',
'Thu, 8 Jul 1999 20:53:13 +0200',4172,'', 2)" >> dblog 2>&1
```

Schreibe alle Attributwerte oder Dateinamen (bei CLOBs), die importiert werden müssen,
in die Import–Datei und führe das IMPORT mit anschließendem DELETE durch.

```
echo '"rfc'$oid'", "ksws-'"'"'test'"'"'", "received.1.1", "opthead.1.1", "bodyf.1.1"'
>imp.del.1.1

db2 "IMPORT FROM imp.del.1.1 OF DEL MODIFIED BY LOBSINFILE INSERT INTO import (oid,
subject, received, con_desc, body )" >> dblog 2>&1
db2 "UPDATE mail AS m SET ( subject, received, con_desc, body) = (SELECT subject,
received, con_desc, body FROM import WHERE VARCHAR(m.oid) = VARCHAR(oid))
WHERE VARCHAR( m.oid) in ( SELECT VARCHAR( oid) FROM import)" >> dblog 2>&1
db2 "DELETE FROM import WHERE VARCHAR(oid) = VARCHAR($oid1) " >> dblog 2>&1
```

Schreibe Log–Datei.

```
db2 "INSERT INTO logtable (time, action, msg_id, oid, target, t_uid, source, s_uid,
flags) VALUES ( CURRENT TIMESTAMP, 'APPEND', '<19990708205313.A530@snoy>', 'rfc$oid',
'INBOX', $uid, '', 0, '_OR______' ) " >> dblog 2>&1
```

Nun die gleichen Schritte für das erste Attachment.

```
db2 "SELECT oid FROM next_oid WHERE name = 'text'" > next_oid
oid=‘awk 'NR == 4  print $1 ' next_oid‘
oid2="text_t('txt"$oid"')"
db2 "UPDATE next_oid SET oid = oid + 1 WHERE name = 'text'" >> dblog 2>&1

db2 "INSERT INTO text (oid, list_nr,con_type,parent, child) VALUES ($oid2, 1,
'text/plain; charset=us-ascii',$oid1, 0)" >> dblog 2>&1

echo '"txt'$oid'", "opthead.2.1.1", "bodyf.2.1.1"' >imp.del.2.1.1

db2 "IMPORT FROM imp.del.2.1.1 OF DEL MODIFIED BY LOBSINFILE INSERT INTO import (oid ,
con_desc, body )" >> dblog 2>&1
db2 "UPDATE text AS m SET ( con_desc, body) = (SELECT con_desc, body FROM import
WHERE VARCHAR(m.oid) = VARCHAR(oid)) WHERE VARCHAR( m.oid) in ( SELECT VARCHAR( oid)
FROM import)" >> dblog 2>&1
db2 "DELETE FROM import WHERE VARCHAR(oid) = VARCHAR($oid2) " >> dblog 2>&1
```

```
### Und die Schritte für das zweite Attachment.

db2 "SELECT oid FROM next_oid WHERE name = 'text'" > next_oid
oid=‘awk 'NR == 4  print $1 ' next_oid‘
oid2="text_t('txt"$oid"')"
db2 "UPDATE next_oid SET oid = oid + 1 WHERE name = 'text'" >> dblog 2>&1

db2 "INSERT INTO text (oid, list_nr,con_type,parent, child) VALUES ($oid2, 2,
'text/plain; charset=us-ascii',$oid1, 0)" >> dblog 2>&1

echo '"txt'$oid'", "opthead.2.1.2", "bodyf.2.1.2"' >imp.del.2.1.2

db2 "IMPORT FROM imp.del.2.1.2 OF DEL MODIFIED BY LOBSINFILE INSERT INTO import (oid ,
con_desc, body )" >> dblog 2>&1
db2 "UPDATE text AS m SET ( con_desc, body) = (SELECT con_desc, body FROM import
WHERE VARCHAR(m.oid) = VARCHAR(oid)) WHERE VARCHAR( m.oid) in ( SELECT VARCHAR( oid)
FROM import)" >> dblog 2>&1
db2 "DELETE FROM import WHERE VARCHAR(oid) = VARCHAR($oid2) " >> dblog 2>&1


### Commit durchführen.

db2 commit >> dblog 2>&1


### Die Datenbank–Verbindung schließen.

db2 disconnect ksws99 >> dblog 2>&1
```