Entfernter Postzugriff auf RPC-Basis

Diplomarbeit

Technische Universität Chemnitz-Zwickau
Fachbereich Informatik

eingereicht von Thomas Schwotzer
geboren am 10.7.1969 in Frankfurt(Oder)

Betreuer: Prof. Hübner

Chemnitz, den 24.5.1994

Zusammenfassung

Die Arbeit untersucht einen entfernten Mailzugang auf Basis drei unterschiedlicher RPC-Implementierungen (ROSE, DCE-RPC, ONC-RPC)

Selbstständigkeitserklärung

Ich, Thomas Schwotzer, erkläre hiermit, die vorliegende Arbeit selbstständig angefertigt zu haben. Sie wurde ohne Hilfe anderer Personen und nur unter Verwendung der aufgeführten Quellen erstellt.

Chemnitz, den 24.5.1994 Thomas Schwotzer

Inhaltsverzeichnis

Selbstständigkeitserklärung **1**

1 Remote Procedure Call **3**
1.1 Verteilte Anwendungen . . . 3
1.2 Das RPC-Konzept . . . 4
1.2.1 Datenformate . . . 6
1.2.2 Netzwerkfehler . . . 7
1.2.3 Aufrufsemantik . . . 7

2 Elektronische Post **10**
2.1 Message Handling System (MHS) . . . 10
2.1.1 Nachricht . . . 11
2.2 Zugangsprotokolle . . . 12
2.2.1 POP3 . . . 13
2.2.2 IMAP2 . . . 16
2.2.3 Zusammenfassung . . . 20
2.2.4 Entwurfsziele/Voraussetzungen . . . 21
2.2.5 Das Protokoll für RPC . . . 21
2.2.6 Diskussion . . . 24

3 DCE-RPC **27**
3.1 Schnittstellenbeschreibung . . . 27
3.1.1 Konstante . . . 28
3.1.2 Pointer . . . 28
3.1.3 Arrays . . . 29
3.1.4 Weitere Typen . . . 29
3.1.5 Prozedurdeklaration . . . 30
3.1.6 Pipe . . . 31
3.2 Werkzeug . . . 32
3.3 Einordnung in das OSI-Referenzmodell . . . 33
3.4 Binding . . . 33
3.4.1 In der Anwendung . . . 36
3.5 Authentication / Security . . . 38
3.5.1 Server . . . 39
3.5.2 Client . . . 40
3.6 Authorization . . . 41
3.7 Fehlerbehandlung . . . 41
3.8 Speichermanagement . . . 42
3.8.1 Client . . . 42
3.8.2 Server . . . 43
3.9 Aufrufsemantik . . . 44

4 ONC-RPC **47**
4.1 Schnittstellenbeschreibung . . . 47
4.1.1 Flags . . . 47
4.1.2 Nachrichtennummern . . . 48
4.1.3 Indikatoren . . . 48
4.1.4 Kommandos . . . 49
4.1.5 Programm . . . 50
4.2 Datenkonvertierung mit XDR . . . 50
4.3 Werkzeug rpcgen . . . 51
4.4 Das RPC Protokoll . . . 52
4.5 Binden . . . 53
4.5.1 In der Anwendung . . . 54
4.5.2 Der Internet Superserver . . . 54
4.6 Authentication . . . 54
4.6.1 UNIX - Authentication . . . 55
4.6.2 DES - Authentication . . . 56
4.7 Security / Authorization . . . 57
4.8 Speichermanagement . . . 58
4.8.1 Clientstub . . . 58
4.8.2 Server . . . 59
4.8.3 Pointer . . . 59
4.9 Fehlerbehandlung . . . 59
4.9.1 Client . . . 60
4.9.2 Server . . . 60
4.10 Aufrufsemantik . . . 60
4.10.1 TCP . . . 60
4.10.2 UDP . . . 60
4.10.3 Asynchroner RPC . . . 61
4.10.4 Batching . . . 61

5 ROSE **62**
5.1 OSI . . . 62
5.1.1 Anwendungsschicht . . . 62
5.2 Das ROSE-Protokoll . . . 64
5.2.1 ROSE Primitives . . . 64
5.2.2 Abbildung der Kommandos auf ROSE-Primitives . . . 66
5.3 Schnittstellenbeschreibung . . . 66
5.3.1 Application Context . . . 67
5.3.2 Operationen . . . 68
5.3.3 Flags . . . 70
5.3.4 Nachrichtennummern . . . 70
5.3.5 Indikatoren . . . 70
5.4 Datenkonvertierung . . . 70
5.5 Werkzeuge . . . 71
5.5.1 ISODE . . . 71
5.6 Binding . . . 71
5.6.1 ISODE . . . 73
5.7 Authentication / Security . . . 75
5.8 Speichermanagement in ISODE . . . 75
5.8.1 Client . . . 75
5.8.2 Server . . . 75
5.9 Fehlerbehandlung in ISODE . . . 76
5.9.1 Server . . . 76
5.9.2 Client . . . 76

5.10 Aufrufsemantik 77

6 Zusammenfassung **78**
6.1 Ausblick 79

A Schnittstellenbeschreibung DCE **80**

B Schnittstellenbeschreibung ONC **83**

C Schnittstellenbeschreibung ROSE **87**

Kapitel 1

Remote Procedure Call

1.1 Verteilte Anwendungen

Eine Anwendung ist verteilt, wenn die zugehörigen Prozesse auf mehreren Hosts im Netzwerk laufen. Dies kann aus unterschiedlichen Gründen sinnvoll sein.

- Eine von mehreren Prozessen gemeinsam genutzte Ressource ist nur einmal vorhanden und wird von einem Server verwaltet. So werden Daten oft nur einmal (zentral) gehalten, um Inkonsistenzen und Redundanz und damit erhöhten Speicherbedarf zu vermeiden. Ein Beispiel ist ein Datenbankserver, aber auch verteilte Verzeichnisdienste wie X.500 oder NIS[1]. Solche Ressourcen können auch „von Natur aus“ nur einmal vorhanden sein, wie z.B. ein spezielles Gerät (Drucker) oder ein Host mit hoher Rechenleistung.
- Zur Lastverteilung werden mehrere Hosts zur Ermittlung eines Ergebnisses benutzt. So können parallelisierbare Berechnungen gut verteilt werden.
- Sollen Daten über ein LAN hinaus übertragen werden, wird oft die Methode *store-and-foreward* benutzt. Kann der Absender den Empfänger nicht direkt erreichen, sendet er die Daten zunächst an einen „näher am Empfänger“ liegenden Host. Dieser versucht seinerseits die Nachricht an den Empfänger zu übermitteln, usw. Das System zur Versendung elektronische Post ist ein solches System.

Mit der Verteilung entstehen allerdings Probleme, die es in einem einzelnen Prozeß nicht gibt.

- Die von einem Prozeß angebotenen Funktionen und Datentypen sind zu beschreiben (**Schnittstellenbeschreibung**). In einem *einzelnen* Prozeß geschieht dies bereits mit der Funktionsdeklaration. Falsche Parameter können bei den meisten Sprachen bereits im Übersetzungsprozeß festgestellt werden.
- Um die Arbeit der Prozesse zu koordinieren, ist der Ablaufes der Kommunikation festzulegen (**Protokoll**).
- Die Kodierung der übermittelten Daten muß den beteiligten Prozessen bekannt sein. Besonders interessant wird dieser Aspekt bei der Kommunikation zwischen Hosts unterschiedlicher Architekturen. Die interne Darstellung von Datentypen kann verschieden sein (*big endian*, *little endian*).

[1] dabei werden zwar auch oft zusätzlich Replikate erzeugt, so daß wieder Redundanz entsteht, es wird aber im allgemeinen vermieden die Daten auf auf jedem beteiligten Rechner zu halten.

- Zwischen den Prozessen muß zu Beginn ein Kommunikationskanal aufgebaut werden (**Verbindungsaufbau**).
- Der rufende Prozeß muß vor dem Verbindungsaufbau die Adresse des zu rufenden Prozesses ermitteln. Dieser Vorgang ist Teil des **Bindens**.
- In einem Netzwerk besteht die Gefahr des Abhörens von Datenpaketen. Bei sensiblen Anwendungen sind Methoden der Verschlüsselung zu benutzen (**Security**).
- Auf einem einzelnen Host können die Zugriffsrechte auf Ressourcen vom Betriebssystem verwaltet werden. Unter UNIX werden diese Rechte für einzelne Nutzer oder Nutzergruppen vergeben. Bei der Nutzung von Ressourcen im Netzwerk besteht der Wunsch, diese Rechte unabhängig davon zu halten, von welchem Host aus auf sie zugegriffen wird, d.h auch unabhängig von den Rechten des Nutzers auf diesem speziellen Host. Die Verwaltung muß an zentraler Stelle im Netzwerk stattfinden (**Authorization**).
- Es besteht die Gefahr, daß sich Prozesse für andere ausgeben. Um dieser Gefahr zu begegnen, sind Methoden zur Verifikation von Identitäten zu finden (**Authentication**).

Zur Programmierung von verteilten Anwendungen können verschiedene Programmierschnittstellen benutzt werden. Verbreitet ist das Konzept der Berkley-Sockets, TLI und verschiedene RPC-Implementierungen.

Um die oben genannten Probleme zu bearbeiten, werden aber weitere Werkzeuge und Dienste benötigt, um z.B. Möglichkeiten zur Datenverschlüsselung anzubieten oder das Binden eines Clients an einen Server zu unterstützen. Ob solche Dienste verfügbar sind, erscheint ebenso als ein Bewertungskriterium wie die Performance einer API und wie sehr der Entwickler von den Unsicherheiten der unterliegenden Protokolle befreit wird.

Ein weiterer – nicht technischer – Aspekt ist die Verbreitung und Verfügbarkeit einer solchen Schnittstelle und ob deren Spezifikation offengelegt ist, wieweit es sich also um ein *offenes System* handelt.

Implementierungen von verteilten Systemen folgen sehr oft dem Client-Server-Modell. Im weiteren soll deshalb unter **Client** der rufende Prozeß verstanden werden, dieser läuft entsprechend auf einem **Clienthost** ab. Analog ist der **Server** der Prozeß, der die gerufenen Dienste zur Verfügung stellt, er läuft auf einem **Serverhost**.

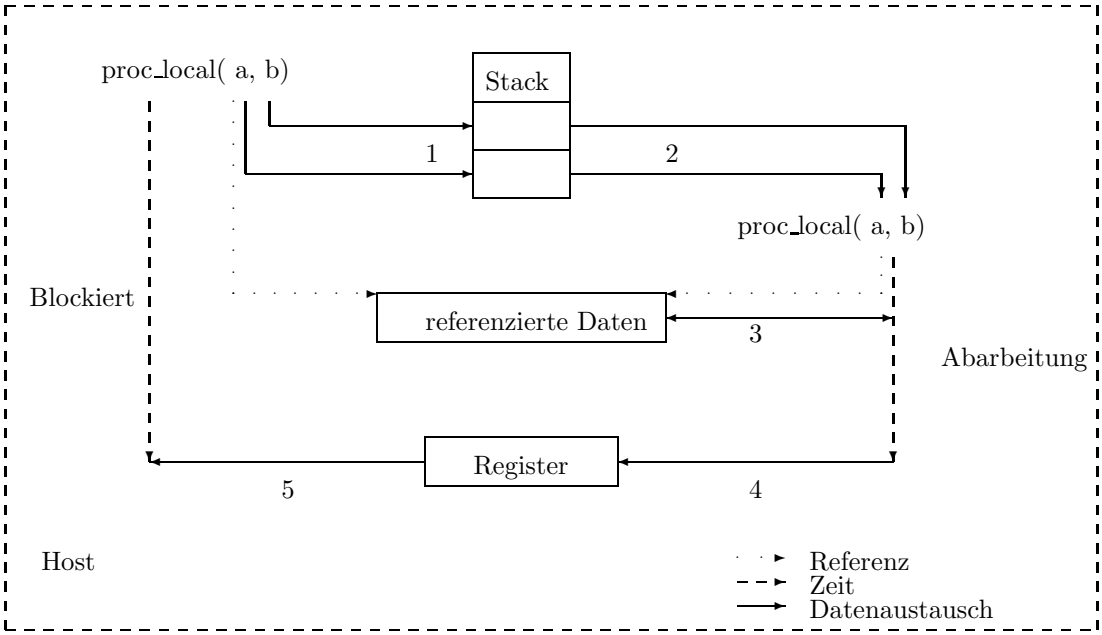
1.2 Das RPC-Konzept

Im folgenden soll ein kurzer Überblick über das Konzept RPC gegeben werden, der im wesentlichen der Begriffsklärung dienen soll. Umfangreichere Diskussionen finden sich u.a. in [Tan2] aber auch in [Blo].

Nach [Tan2] geht die Idee für RPC auf einen Artikel von Birell und Nelson aus dem Jahre 1984 zurück. Ähnlich einem lokalen Prozeduraufruf soll es möglich sein, eine Prozedur auf einem anderen Host aufzurufen.

Die rufende Prozedur legt die Übergabeparamater in einem gemeinsamen Speicher (Stack) ab, die rufende Prozedur wird in ihrer Ausführung angehalten. Nun entnimmt die gerufene dem Stack die Parameter und kommt zur Ausführung. Am Ende werden die Rückgabeparameter ebenfalls in einen gemeinsamen Speicherbereich (oft auch ein Register) geschrieben, danach erhält die rufende Prozedur die Steuerung zurück und entnimmt die Prozedurergebnisse

Es gibt verschiedene Möglichkeiten zur Parameterübergabe. Zum einen kann der Parameter selbst (*call by value*), zum anderen kann ein Verweis auf ihn (*call by reference*) übergeben werden. Im ersten Fall kann eindeutig zwischen Eingabe- und Ausgabewerten unterschieden werden, während im zweiten Fall eine Manipulation der referenzierten Daten aus der gerufenen Prozedur heraus möglich ist. Allein anhand des Aufrufes ist nicht entscheidbar, ob die Daten der Eingabe, der Ausgabe oder beidem dienen, siehe Abbildung 1.1.



1. Die rufende Prozedur legt die Parameter im Stack ab (a sei eine Referenz auf einen Speicherbereich).
2. Die Steuerung wurde an die gerufene Prozedur übergeben, die Parameter werden dem Stack entnommen.
3. Die Prozedur wird ausgeführt, auf die referenzierten Daten kann zugegriffen werden.
4. Die Arbeit ist beendet, die Rückgabewerte werden abgelegt.
5. Die rufende Prozedur erhält die Steuerung zurück.

Abbildung 1.1: Lokaler Prozeduraufruf

Ein wesentlicher Unterschied besteht im RPC darin, daß es für die rufende und gerufene Prozedur keinen gemeinsamen Speicher gibt. Die Parameter müssen von einem Adreßraum in den anderen übertragen werden. Dazu dienen die **Stubs**. Der Clientstub enthält für jede entfernte Prozedur eine gleichnamige mit der gleichen Signatur. Das Clientprogramm ruft die Stubroutine mit den Eingabeparametern. Diese werden mit einem Identifikator für die Prozedur in ein Datenpaket abgelegt und zum Server übertragen. Der Serverstub entnimmt dem Paket die Werte, stellt anhand des Identifikator fest, welche Prozedur gerufen werden und ruft sie auf. Nach der Abarbeitung erhält der Serverstub die Ergebnisse. Sie werden in einem Datenpaket an den Client zurückgesandt, siehe Abbildung 1.2.

Der eben beschriebene Vorgang ist der pure entfernte Prozeduraufruf. Die untersuchten Implementierungen des Konzeptes RPC umfassen aber nicht nur eine bloße API. Es existieren weitere Komponenten, die es ermöglichen, die im vorherigen Abschnitt genannten Probleme der verteilten Anwendungen zu bearbeiten. So gibt es in jedem Fall ein Beschreibungsmittel für die Schnittstelle, die ein (Server-)Prozeß anbietet und Komponenten, die die Daten in einer geräteunabhängigen Form übertragen bzw. bei DCE die unterschiedlichen Darstellungen konvertieren. Vor allem im DCE ist RPC fest in das gesamte System integriert und somit auch eng verbunden mit dem Verzeichnisdienst oder dem Security Service. In den folgenden Kapiteln wird darauf näher eingegangen.

Somit steht RPC nur im engeren Sinne für den bloßen Prozeduraufruf, im weiteren Sinne umfaßt RPC eine Anzahl von Komponenten.

So sehr RPC auch an den lokalen Prozeduraufruf angelehnt wurde, gilt es doch, einige wesentliche Unterschiede zu beachten:

- Sämtliche Parameter müssen zwischen den Prozeduren ausgetauscht werden, ein Ruf *call by reference* macht bei RPC keinen Sinn[2].
- Client und Server können intern unterschiedliche Datenformate benutzen. Die Stubs übernehmen die Konvertierungen.
- Im Gegensatz zum lokalen Prozeduraufruf kommt das benutzte Netzwerk als Fehlerquelle hinzu. Dadurch verliert das Konzept an der erwünschten Transparenz [Tan2].
- Vor dem Aufruf einer Prozedur muß erst die Adresse des Servers ermittelt und ein Kommunikationskanal aufgebaut werden.

1.2.1 Datenformate

Zur Konvertierung unterschiedlicher Datenformate gibt es grundsätzlich zwei Vorgehensweisen:

- Es wird ein von den Hosts unabhängiges Format definiert. Bei jedem Ruf werden die Parameter von der internen Darstellung in dieses Format umgewandelt und übertragen. Auf der Empfängerseite vollzieht sich der umgekehrte Vorgang. Der Vorteil dieses Vorgehens ist es, nur ein gemeinsames Datenformat zu besitzen, in das die jeweilige interne Darstellungen überführt wird. Der Nachteil besteht darin, daß auch bei gleichartigen Hosts zweimal eine Konvertierung der Datenformate stattfindet.

- Diesem Nachteil hilft ein anderes Vorgehen ab. Dieses wird auch als *receiver makes it right* bezeichnet. Der Absender wandelt seine Daten nicht in eine andere Darstellung um, teilt dem Empfänger lediglich das benutzte Format mit. Dieser muß das Format kennen und gegebenenfalls in das eigene umwandeln. Bei gleichen internen Darstellungen findet keine Konvertierung statt. Nachteilig ist, daß der Empfänger alle möglichen Eingabeformate kennen muß.

[2] in Schnittstellenbeschreibungen kann eine Referenz zwar vereinbart werden doch übernehmen hierbei nur die unterliegenden Funktionen das vollständige Übertragen; werden diese Daten sowohl hin- ,als auch zurückkopiert wird von *call by copy/restore* gesprochen[Tan2]

1.2.2 Netzwerkfehler

Bei einem lokalen Prozeduraufruf laufen rufende und gerufene Prozedur auf einem Host, in einem Prozeß. Die Übergabe von Daten erfolgt über den Hauptspeicher. Ausnahmesituationen, wie Hardwarefehler oder *exceptions* betreffen in gleichem Maße rufende wie gerufene Prozedur.

Bei einem entfernten Prozeduraufruf tritt ein potentiell unsicheres Kommunikationsmedium zwischen Client und Server. Damit können Fehler entstehen, die bei einem lokalen Prozeduraufruf unbekannt sind.

- Daten gehen verloren
- Daten werden dupliziert
- ein beteiligter Prozeß fällt aus

In [Tan2] und [Ros1] werden Möglichkeiten aufgezeigt, diesen Problemen zu begegnen, an dieser Stelle sollen nur einige Ideen genannt werden.

Gegen den Verlust von Daten kann mit Quittungen und Timeouts vorgegangen werden. Der Sender erwartet für jedes abgeschickte Paket eine Quittung. Erreicht sie ihn in einer gewissen Zeit nicht, erfolgt die Übertragung erneut. Gehen allerdings die Quittungen verloren, entstehen Duplikate.

Diese können erkannt werden, indem jeder Ruf mit einer eindeutigen Nummer versehen wird (z.B. *invokation ID* in ROSE). Führt der Empfänger Buch über die bereits erhaltenen Rufe, kann er Duplikate einfach verwerfen.

Der Ausfall eines Prozesses ist schwerer zu bearbeiten. Fällt ein Client aus, so konsumiert der laufende Server zumindest Ressourcen. In [Tan2] werden unterschiedliche Konzepte vorgestellt, wie mit verwaisten Servern verfahren werden kann.

Der Client kann den Ausfall *seines* Servers zumindest vermuten, wenn jede Prozedur einen Rückgabewert liefert[3]. Es lassen sich allerdings RPC benutzen, die keine Werte zurückgeben. In diesem Fall ist nicht entscheidbar, ob eine Prozedur erfolgreich ausgeführt wurde. In [Ros1] wird diesbezüglich auf die Bedeutung von **Totality** hingewiesen. Totality heißt, daß die Ergebnisse bei einem erfolgreichen Beenden der Prozedur, bei einem Fehler in der Prozedur und bei einem Netzwerkfehler definiert und unterscheidbar sind.

„This is an important feature of any reliable system. “ M.T. Rose [Ros1][4]

Das Erfüllen dieser Forderung führt die RPC Semantik näher an die Semantik lokaler Aufrufe, denn auch im lokalen Fall ist das Beenden einer Prozedur (mit oder ohne Rückgabewerte), an der Rückgabe der Steuerung an die rufende Prozedur zu erkennen. Implizit ist damit bekannt, daß die Prozedur beendet wurde.

Fehler in der Anwendung lassen sich durch geeignete Rückgabeparameter anzeigen, andere Fehler (*exceptions*) werden z.B. in UNIX über Signale weitergegeben. Totality ist somit bereits ohne weitere Eingriffe realisiert.

1.2.3 Aufrufsemantik

In Abhängigkeit davon, wie mit Netzwerkfehlern umgegangen wird, wird sichergestellt, wie oft ein Aufruf den Server erreicht, wie oft also die entfernte Prozedur ausgeführt wird. Dies wird als **Aufrufsemantik** bezeichnet, u.a. in [Ros1] werden drei unterschieden:

[3] nach einer gewissen Zeit und einer Anzahl von erfolglosen Kontaktversuchen kann angenommen werden, daß der Server nicht mehr läuft

[4] In den untersuchten Systemen wird Totality nicht erzwungen, im Gegenteil, in DCE wird mit der *maybe* Semantik direkt ein Weg angeboten, das zu umgehen.

genau einmal Jeder Ruf wird genau einmal ausgeführt. Das ist die Semantik des lokalen Prozeduraufrufes.

wenigstens einmal Es wird garantiert, daß ein Ruf mindestens einmal sein Ziel erreicht und die Prozedur ausgeführt wird. Für diese Semantik ist es erforderlich, daß die gerufene Prozedur **idempotent** ist. Idempotente Prozeduren geben bei gleichen Eingabeparametern stets die gleichen Ausgabewerte zurück, der Zustand des Servers ändert sich nicht. Zum Beispiel in [Mey] wird in diesem Zusammenhang zwischen Prozeduren und Funktionen unterschieden. Funktionen führen Berechnungen ohne Nebeneffekte aus, sie kommen damit dem mathematischen Begriff näher. Im Gegensatz dazu besitzen Prozeduren Nebeneffekte, sie sind nicht idempotent. Damit wird aber die Semantik einer Prozedur oder Funktion beschrieben, die ein Compiler nicht erkennen kann.

Im folgenden wird auf eine solche Unterscheidung verzichtet.

höchstens einmal Der Ruf wird höchstens einmal sein Ziel erreichen.

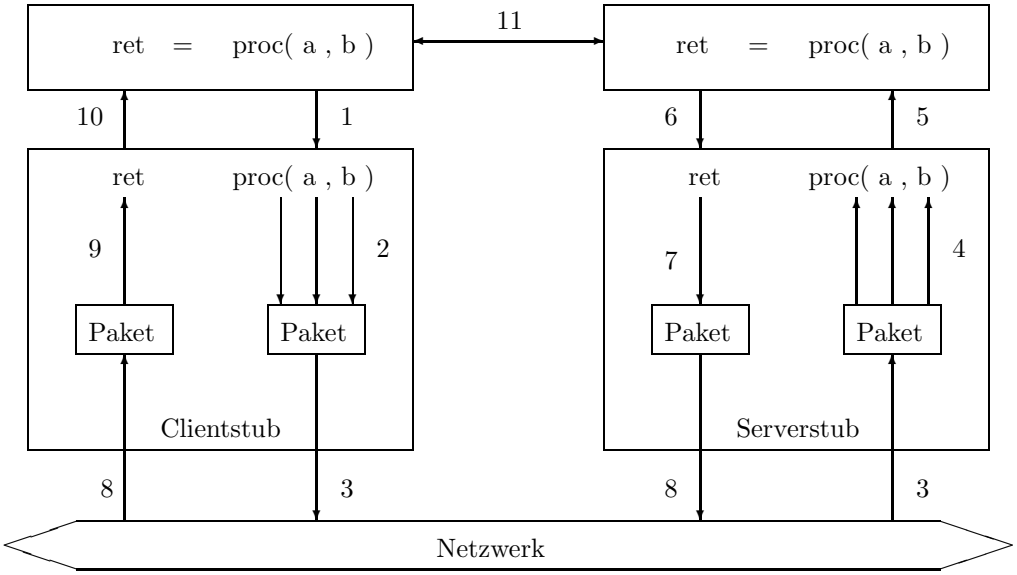
In [Ros1] werden ebenfalls Methoden zur Gewährleitung der einer Semantik angesprochen. Hier nur einige Ideen.

Die letzte Semantik ist am einfachsten umzusetzen. Ein Ruf wird nur einmal gesandt, er kommt an oder nicht.

Muß jeder Ruf quittiert werden, kann der Client über Timeouts gesteuert Rufe erneut absetzen, bis er eine Quittung erhält. Das führt zu *wenigstens einmal*.

Wird jeder Ruf zusätzlich mit einer ID versehen und führt der Server über bereits bearbeitete Rufe Buch, so kann er mehrfach gesendete Rufe zwar quittieren, aber nicht (wiederholt) bearbeiten. Das führt zu *genau einmal*.

Voraussetzung ist dabei stets, daß keines der beteiligten Systeme abstürzt.



1. Der Client ruft die (entfernte) Prozedur auf.
2. Der **Clientstub** stellt aus den Eingabedaten ein Datenpaket her (**Marshalling**).
3. Die Daten werden zum Serverprozeß übertragen.
4. Der **Serverstub** packt die Parameter aus (**Unmarshalling**)
5. Die (entfernte) Prozedur wird gerufen.
6. Die Prozedur gibt Ergebnisse zurück.
7. Die Ergebnisse werden zusammengepackt
8. und zurückgesendet.
9. Der Clientstub packt die Ergebnisse aus und
10. übergibt sie der rufenden Prozedur.
11. Zwischen Client und Server findet eine „virtuelle Kommunikation“ statt.

Abbildung 1.2: Entfernter Prozeduraufruf

Kapitel 2

Elektronische Post

„Electronic mail is by far the most popular of all the existing networkbased applications.“ M.T.Rose [Ros1]

Aus diesem Grund gibt es eine Fülle von Literatur, genannt werden soll hier [Ros1], [Ros2], das sich mit Konzepten im Internet befaßt oder auch [X.400].

Wie in anderen Bereichen der Kommunikation in *offenen Systemen* wurden mit OSI Standards und Modelle vorgeschlagen. Diese Standards sind allerdings nicht in dem Maße akzeptiert und verbreitet wie die Protokolle des Internet. Die Gründe sind unterschiedlicher Natur. M.T. Rose's Veröffentlichungen geben dazu detaillierte Hinweise. Dieser Zustand führt oft dazu, daß Begriffe und Modelle von OSI benutzt werden, um Protokolle des Internet einzuordnen. So wird auch im weiteren verfahren. Der folgende Abschnitt dient dazu, das MHS vorzustellen und einige im weiteren benutzte Begriffe einzuordnen. Ein ausführliche Beschreibung befindet sich z.B in [Ros1].

2.1 Message Handling System (MHS)

MHS dient dem Austausch von Nachrichten zwischen Nutzern, das sind Personen oder Prozesse.

Will ein Nutzer eine Nachricht übermitteln, tritt er mit dem MHS über einen **User Agent** (UA) in Verbindung. Der UA übergibt die Nachricht an das **Message Transfer System** (MTS) über einen lokalen **Message Transfer Agent** (MTA). Das MTS leitet die Nachricht weiter zu einem MTA, der den UA des Empfängers direkt erreichen kann. Die Weiterleitung kann über mehrere MTA erfolgen (*third-party delivery*). Der UA des Empfängers kann mit einem UA eine Nachricht bei seinem lokalen MTA beziehen. Um die UA nicht permanent verfügbar halten zu müssen, wurde ein weiteres Element eingeführt, das **Message Store** (MS). Es dient als temporärer Speicher zwischen dem MTA und dem UA. Der MTA verwaltet es lokal. Die Nachrichten werden nicht sofort an den UA übergeben, sondern im MS für ihn bereitsgestellt.

Beim Versenden von Nachrichten ist ein ähnlicher Weg möglich. Der UA plaziert die Nachrichten im MS, das MTA greift seinerseits darauf zu (*indirect submission*), siehe auch Abbildung 2.1.

Vier Protokolle werden in diesem System benutzt.

Messaging Protocol läuft zwischen zwei UA.

Relaying Protocol läuft zwischen zwei MTA.

Submission Protocol dient dem Übermitteln einer Nachricht vom UA zum MTA.

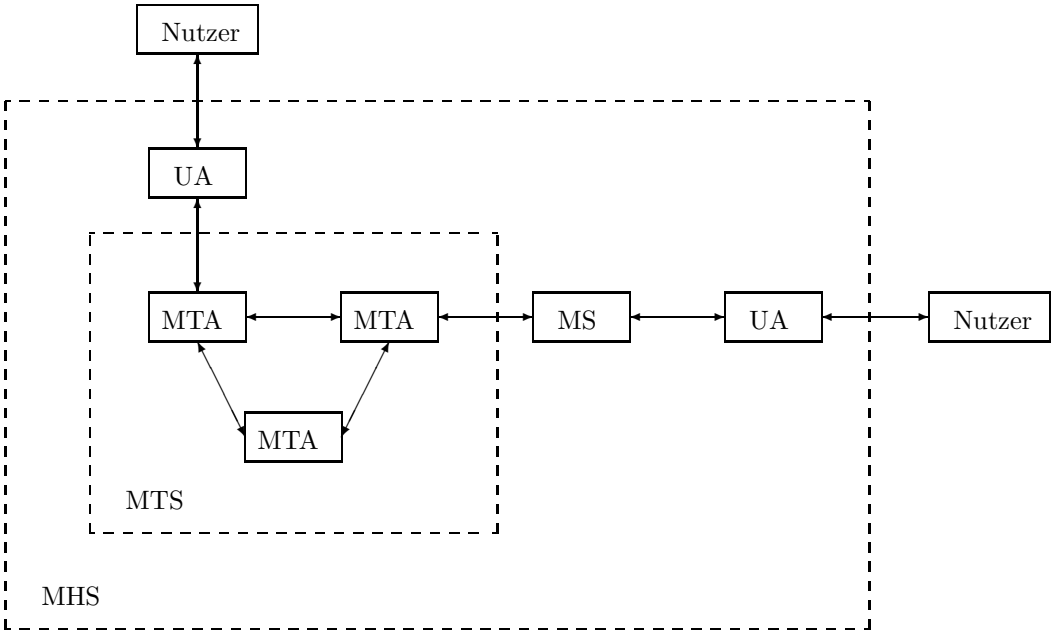


Abbildung 2.1: Das Mail Handling System (MHS)

Delivery Protocol dient der Ausgabe einer Nachricht von einem MTA an einen UA.

MTA und UA kommunizieren bei zwei Gelegenheiten. Will der Nutzer eine Nachricht versenden, benutzt er den UA, dieser wiederum übergibt die Nachricht über den *posting slot* eines MTA an das MTS. Zum Verteilen wird die Nachricht über den *delivery slot* eines MTA an einen UA übergeben, siehe auch Abbildung 2.2.

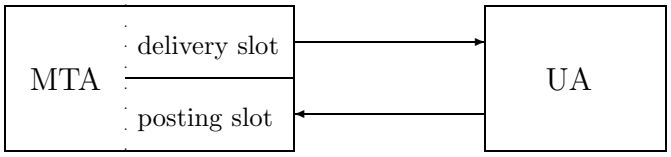


Abbildung 2.2: Zugriff des UA auf MTA

Das sind recht unterschiedliche Aufgaben. Entsprechend gibt es dazu unterschiedliche Protokolle. Ein UA der beide Funktionen bedient wird auch als **Split User Agent** bezeichnet.

2.1.1 Nachricht

Nach MHS besteht eine Nachricht aus zwei Teilen, der

Envelope enthält Informationen für das MHS, wie Adressen des *originator* und des *recipient*, der Inhalt des

Content ist die eigentliche Nachricht. Die innere Struktur des Content bleibt dem MHS verborgen.

Der Aufbau des Content ist ebenfalls Gegenstand eines OSI-Vorschlags.

Ein im Internet benutzter Standard wird in [822] beschrieben. Danach unterteilt sich der Content ebenfalls in zwei Teile, einen

Header, der Informationen über die Herkunft und das Ziel, über Zeiten und eventuell den Inhalt der Nachricht enthält und einen

Body, der Daten enthält, deren Struktur nicht weiter definiert werden.

Siehe Abbildung 2.3.

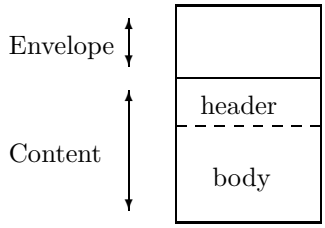


Abbildung 2.3: Aufbau einer Nachricht

Ziel der Arbeit ist es, ein Zugangsprotokoll zu implementieren.

2.2 Zugangsprotokolle

Eine Nachricht wird im MTS zu einem MTA transportiert, auf den der Nutzer mit einem UA zugreifen kann. Üblicherweise wird sie in die Mailbox des Empfängers eingetragen.

Wie gelangt nun der Nutzer an seine Nachrichten ? Das ist Aufgabe des UA. In Abhängigkeit davon, wo sich UA und MTA befinden, lassen sich drei Szenarien unterscheiden.

1. Die Mailbox und damit auch ein MTA und UA befinden sich auf dem Host des Nutzers.
2. Auf einem ausgezeichneten Host (Mail-Server) laufen sowohl der MTA, als auch der UA.
3. Ein Mail-Server verwaltet für jeden Nutzer eine Mailbox, der UA kann über ein Zugangsprotokoll auf sie zugreifen.

Vorteilhaft am ersten Fall ist, daß der Nutzer lokal auf seine Nachrichten zugreifen kann. Doch es existieren einige Nachteile.

- Die Hosts müssen ständig verfügbar sein, um eine Verbindungsaufnahme anderer MTA zu erlauben. Damit werden alle Nutzerhosts zusätzlich belastet.
- Der Verwaltungsaufwand für die Installation/Wartung und für eventuelle Backupdienste steigt. Der Nutzer muß stets den gleichen Host benutzen.

Die Verteilung der Daten wird mit der zweiten Herangehensweise vermieden. Die zusätzliche Last für den MTA trägt ebenso nur ein Host. Ein so ausgezeichneter Server kann ausreichend mit Ressourcen ausgerüstet werden, um die eingehenden Nachrichten zu halten (Speicherkapazität). Bei zentral gehaltenen Daten ist ebenso

die Organisation von Backupdiensten einfacher. Zum anderen muß nur der Server ständig verfügbar sein.

Um an die Nachrichten zu gelangen, muß der Nutzer den UA erreichen. Dazu kann er selbst zu dem Serverhost gehen, was nur in Ausnahmefällen eine befriedigende Lösung sein kann. Er kann den UA aber auch mittels telnet o.ä. entfernt steuern. Damit kann zwar die Mailbox manipuliert werden, die Daten sind zunächst aber nicht lokal verfügbar. Dazu müssen andere Dienste benutzt werden (ftp).

Durch den Einsatz eines Netzwerkfilesystems entfallen die letztgenannten Probleme aber auch.

Die dritte Strategie erscheint als die günstigste. Zum einen können die Nachrichten zentral gehalten werden, mit den damit verbundenen Vorteilen, zum anderen kann der Nutzer mittels eines UA seine Mailbox verwalten und auf die Nachrichten zugreifen und sie übertragen. Der Nutzer benötigt lediglich einen UA, der über ein geeignetes Zugangsprotokoll verfügt. Dieser UA wird sicherlich, im Gegensatz zu einem NFS, vergleichsweise wenige Ressourcen beanspruchen und geringe Ansprüche an das Betriebssystem stellen.

Im weiteren werden zwei im Einsatz befindliche Zugangsprotokolle diskutiert.

2.2.1 POP3

Die aktuelle Beschreibung des **Post Office Protocol - Version 3** erfolgt in [1460]. Dem Autor des Protokolls zufolge gab es zwei grundsätzliche Entwurfsziele[Ros2].

- Die Serverseite soll recht einfache Zugriffsmöglichkeiten auf die Mailbox[1] erlauben. Dadurch soll die Last auf dem Serverhost gering gehalten werden, der gleichzeitig mehrere Clients bedienen kann.
- Die Clientseite soll nur über eine minimale Intelligenz verfügen müssen. Das soll dem Cliententwickler die Möglichkeit geben, Implementierungen an die Leistungsfähigkeit des jeweiligen Hosts anzupassen.

Ein POP3 Server setzt als Transportprotokoll TCP voraus, er benutzt den Port 110. Wie üblich sendet der POP3-Client Requests, der Server antwortet mit Responses, die stets mit einem von zwei möglichen Indikatoren eingeleitet werden.

+OK zeigt an, daß das Kommando korrekt bearbeitet werden konnte.

-ERR bedeutet dem Client, daß ein Fehler auftrat.

Mit einem positiven Indikator folgen eventuell weitere Daten, mit einem Fehlerindikator wird eine lesbare[2] Fehlermeldung übertragen.

Die erste Aktion des Clients ist der Aufbau einer TCP - Verbindung. Ist diese hergestellt, sendet der Server eine positive Antwort mit einem Grußtext.

Mit diesem Gruß tritt das Protokoll in die erste der drei Phasen ein.

Authorization State

Der Nutzer muß sich mit einem Nutzername (`USER name`) identifizieren und mit einem Passwort (`PASS password`) authentifizieren.

Das Passwort wird unverschlüsselt übertragen. Dies birgt potentiell ein Sicherheitsrisiko. In [1460] wird ein anderer Mechanismus vorgeschlagen: Mit dem ersten

[1] Im RFC wird grundsätzlich von *maildrop* gesprochen. Diese entspricht einer Mailbox der bisherigen Betrachtungen. Es soll deshalb auch weiterhin nur ein Wort benutzt werden.

[2] Im weiteren soll darunter eine *für den Menschen* lesbare Meldung verstanden werden.

Gruß sendet der Server eine Zeichenkette, die für diesen Server auf diesem Host einmalig ist. Außerdem verfügen sowohl Client als auch Server über einen gemeinsamen Schlüssel. Der Client kodiert den Schlüssel mit der Zeichenkette und sendet das Ergebnis zum Server. Der Server vollzieht die gleiche Kodierung und vergleicht die Ergebnisse.

Mit der erfolgreichen Authentifizierung des Nutzer setzt der Server einen exklusiven Lock auf die Mailbox des Nutzers und teilt diese in die einzelnen Nachrichten auf. Den Nachrichten wird mit 1 beginnend eine Nummer zugewiesen.

Die nächste Phase wird erreicht.

Transaction State

Dem Client stehen sechs (optional sieben) Kommandos zu Benutzung der Mailbox zur Verfügung. Der Server hält in dieser Phase eine Variable (*highest number accessed*). In ihr wird die höchste Nummer einer Nachricht gespeichert, auf die bereits mit `RETR` zugegriffen wurde.

Dem Kommandonamen folgen eventuell Parameter. Optionale Parameter erscheinen in eckigen Klammern, `nr` bezeichnet jeweils die Nachrichtennummer auf die das Kommando angewandt werden soll. Die Nummer wurde vom Server vergeben.

Die folgenden Kommandos sind verfügbar.

`STAT` fordert eine *drop list* an. Diese enthält die Anzahl der Nachrichten in der Mailbox und die Größe der Box.

`LIST [nr]` fordert eine *scan list* an. Diese enthält für jede Nachricht deren Nummer und Größe auf dem Server.

`RETR nr` fordert eine gesamte Nachricht vom Server an.

`DELE nr` markiert eine Nachricht als gelöscht.

`LAST nr` fordert die *highest number accessed* an.

`RSET nr` setzt alle Löschmarken zurück und *highest number accessed* auf den Startwert.

`NOOP` manipuliert die Mailbox nicht, der Server liefert in jedem Fall ein positive Antwort.

`TOP nr n` ist optional zu implementieren. Es werden der Messageheader und die ersten n Zeilen der Nachricht angefordert.

Mit dem Kommando `QUIT` wird die die letzte Phase erreicht.

Update State

Alle als gelöscht markierte Nachrichten werden nun tatsächlich aus der Mailbox entfernt. Die Verbindung wird abgebaut.

Szenario

Das Szenario ist in [1460] zu finden.

```
S: <wait for connection on TCP port 110>
    ...
C: <open connection>
S:    +OK POP3 server ready <1896.697170952@dbc.mtview.ca.us>
```

```
C:    APOP mrose c4c9334bac560ecc979e58001b3e22fb
S:    +OK mrose's maildrop has 2 messages (320 octets)
C:    STAT
S:    +OK 2 320
C:    LIST
S:    +OK 2 messages (320 octets)
S:    1 120
S:    2 200
S:    .
C:    RETR 1
S:    +OK 120 octets
S:    <the POP3 server sends message 1>
S:    .
C:    DELE 1
S:    +OK message 1 deleted
C:    RETR 2
S:    +OK 200 octets
S:    <the POP3 server sends message 2>
S:    .
C:    DELE 2
S:    +OK message 2 deleted
C:    QUIT
S:    +OK dewey POP3 server signing off (maildrop empty)
C:  <close connection>
S:  <wait for next connection>
```

Vorteile

Das Entwurfsziel lautete Einfachheit, das bietet einige Vorteile.

- Ein Server muß nur recht einfache, wenig rechenintensive Funktionen ausführen. Die Last wird damit gering gehalten.
- Das Protokoll erzwingt keine große Intelligenz im Client. So liegt es beim Entwickler, die Clients an das jeweilige System anzupassen.
- Der Client kann sehr detailliert aussuchen, welche Informationen übertragen werden sollen. Es können Informationen über die Größe (`STAT`) der Mailbox und über die Größe der einzelnen Nachrichten (`LIST`) getrennt bezogen werden. Mit `TOP` wird der Header übertragen und eventuell ein Teil des Body. Durch eine gute Clientimplementierung kann die Netzwerklast gering gehalten werden.
- Mit `TOP` können Teile des Nachrichtenkörpers angefordert werden. Damit kann der Nutzer eventuell entscheiden, ob der gesamte Body benötigt wird. Unnütze Übertragungen könnten eingespart werden.
- Die Einfachheit eines Protokoll wird dessen Verbreitung erleichtern.

Nachteile

Durch die erreichte Einfachheit werden einige Konzepte nicht unterstützt.

- Nach der Authentifizierung wird eine Mailbox geöffnet, die der Server anhand des Nutzernamens ermittelt. Der Client hat keinen weiteren Einfluß auf diese Auswahl. Ein Konzept von *bulletin boards* ist mit POP nur durch einen gesonderten Nutzernamen mit einem eigenen Passwort möglich. Das erhöht den Verwaltungsaufwand.

- Der Server setzt einen exklusiven Lock auf die Mailbox. Das bedeutet, daß während einer Sitzung keine Nachrichten eingehen können.

- Der Server kapselt das Format der Nachricht nicht. Es wird das Format [822] unterstützt. Die Einordnung dieser Eigenschaft zu den Nachteilen ist allerdings recht willkürlich. Obwohl es ursprünglich zur Übertragung von reinen Textnachrichten entworfen wurde, hat es sich als flexibel für Erweiterungen erwiesen. Mit dem Konzept von *MIME* wird beispielsweise auch die Übertragung von Multimedia-Mail möglich[3].

- Die Mailbox verfügt über kein Gedächtnis. Es ist für den Client nicht möglich, Nachrichten zwischen zwei Sitzungen in irgendeiner Weise zu markieren. Es ist somit auch nicht möglich, zwischen neu eingetroffenen und bereits gesehenen Nachrichte zu unterscheiden. Denkbar ist, ein solches Gedächtnis in den Client zu implementieren. Dieser müßte lokal die gewünschten Vermerke halten. Der Nutzer wäre damit gezwungen, stets den gleichen UA zu benutzen. Dieses Vorgehen widerspricht aber grundlegend dem Ziel, den Nutzer unabhängig von einem speziellen Host zu machen.

- Die Kommandos unterstützen als Parameter nur eine einzelne Nachrichtennummer. So ist zum Beispiel für das Löschen von zehn Nachrichten zehnmal das Kommando `DELE` zu benutzen. Eine Möglichkeit dabei Zeit zu sparen, besteht darin, diese zehn Rufe an den Server zu schicken, ohne eine Antwort abzuwarten. In diesem Fall wird aber das synchrone Protokoll verlassen, die Kommunikation erfolgt asynchron[4].

- Das Kommando `STAT` erscheint redundant. Nach der Authentifizierung wird im allgemeinen eine *droping list* der gewählten Mailbox angefordert. Die Mailbox ist allerdings nach dem Kommando `PASS` bereits ausgewählt und geöffnet. Eine *droping list* ist also auch als zusätzliche Information zum Kommando `PASS` denkbar. In dem in [1460] genutzten Beispiel wird als zusätzlicher Text zu `PASS` auch genau eine *droping list* übergeben, um sie gleich darauf mit `STAT` wiederholt anzufordern, siehe Abschnitt 2.2.1. Die Informationen über die gesamte Mailbox erscheinen außerdem als so elementar, daß davon ausgegangen werden kann, daß diese bei jeder Sitzung vom Client angefordert werden.

- Da der Server von Last soweit als möglich verschont werden soll, gibt es keine Möglichkeit, Informationen über die Nachrichten außer deren Länge zu erhalten (*scan list*). Das kann auf Kosten der Netzlast gehen, wenn der Nutzer nur Nachrichten mit einem bestimmten Inhalt sucht. Mit diesem Protokoll müssen zunächst die Daten übertragen werden, um sie dann (lokal) zu untersuchen.

2.2.2 IMAP2

Eine Beschreibung des **Interactive Mail Access Protocols - Version 2** befindet sich in [1176].

[3] Das dabei genutzte Format definiert die Struktur des Körpers einer Nachricht, das Format der Gesamtnachricht bleibt konform zu [822].

[4] bei dem streambasierten Austausch von TCP mag dieses Vorgehen noch praktikabel sein, bei RPC ist die Wahl von synchronem Protokoll und asynchronem mit weitreichenden Folgen verbunden.

Beim Entwurf des Protokolls gab es zwei grundlegende Entwurfsziele:

- Der innere Aufbau einer Nachricht soll dem Client nicht bekannt sein müssen.
- Die Netzlast soll gering gehalten werden.

Um das erste Ziel zu erreichen, kapselt der Server die Nachrichten und stellt die Datenstruktur des *envelope* zur Verfügung[5]. Dieser enthält neben dem Body einige Headerinformationen.

Der Client kann diesen *envelope* vom Server in einer strukturierten Form beziehen[6]. Benutzt der Client tatsächlich nur diese Struktur, bleibt ihm eine Änderung des Nachrichtenformates im MHS verborgen.

Zur Verringerung der Netzlast bietet der Server ausgiebige Methoden zum Durchmustern der Nachrichten an. Der Nutzer ist somit in der Lage, bereits vor dem Übertragen der Nachrichten, diese in „interessante“ und „uninteressante“ zu unterscheiden. Der Server verwaltet eine Anzahl von Flags, mit denen Nachrichten gekennzeichnet werden können.

RECENT	Die Nachricht ist neu.
SEEN	Die Nachricht wurde bereits (in einer vorherigen Sitzung) übertragen.
ANSWERED	Nachricht wurde beantwortet.
DELETED	Die Nachricht ist als gelöscht markiert.
FLAGGED	Dieses Flag ist mit keiner Semantik belegt.

Client und Server tauschen drei verschiedene Arten von Daten aus.

Datum	Syntax
Request	`tag Kommandoname [Parameter]`
Response	`tag Indikator text`
Unsolicited Data	`* Kommandoname Daten`

Ein *tag* wird vom Client generiert und einem Request vorangestellt. Der Server antwortet mit einem Response und benutzt das *tag* des zugehörigen Request. Damit kann der Client ein Response eindeutig einem Request zuordnen. Mit den *unsolicited data* werden die eigentlichen Nutzdaten übertragen.

Der Grundgedanke ist, daß der Client einen lokalen Cache besitzt, den der Server nach Aufforderung mit aktuellen Daten füllt. Dazu sendet der Client ein Request, der Server überträgt daraufhin die angeforderten Daten als *unsolicited data*. Die Struktur der Daten ist dem Client durch den vorangestellten Kommandonamen bekannt, sie werden in den lokalen Cache eingetragen. Zum Schluß überträgt der Server einen Response und teilt so dem Client mit, daß er über die aktuellen Daten verfügt. Der Server kann Buch führen, welche Daten der Client bereits bezogen hat. Fordert der Client mehrfach die gleichen Daten an (die sich in der Zwischenzeit nicht geändert haben), so muß der Server diese nur beim erstenmal übertragen. Auf alle weitere Requests muß er nur mit einem Response antworten.

Es gibt drei verschiedene Arten von Indikatoren:

OK Der Request wurde erfolgreich bearbeitet, der Client besitzt die aktuellen Daten.

NO Bei der Abarbeitung des Kommandos trat ein Fehler auf.

[5] Er hat mit dem Envelope aus MHS nur den Namen gemeinsam.
[6] Es wird die aus LISP bekannte *propertylist* benutzt.

BAD Der Client hat sich nicht an das Protokoll gehalten, das Kommando wurde nicht bearbeitet.

Um eine IMAP2 - Sitzung zu beginnen, stellt der Client eine Verbindung zum Server her, dieser sendet mit *unsolicited data* einen Gruß.

Authentifizierung

Der Nutzer muß sich nun identifizieren. Hierzu dient das Kommando `LOGIN`. Mit ihm wird der Name und das Passwort übertragen.

Auswahl einer Mailbox

Der Server unterstützen mehr als eine Mailbox. Mit
`FIND MAILBOX pattern`
liefert der Server eine Liste aller die mit `pattern` matchen. Wildcards werden unterstützt. Der Name einer Mailbox ist frei wählbar, er muß kein Filename sein.

Der Server muß für jeden Nutzer eine Standardmailbox anbieten. Diese hat den Namen *INBOX*.

Mit
`SELECT Mailbox`
wird eine Box ausgewählt und geöffnet. Der Server sendet bei Erfolg Informationen über die Box[7]. Das Kommando kann in einer Sitzung beliebig oft angewendet werden, eine bereits geöffnete Mailbox oder ein geöffnetes Bulletin Board wird dabei geschlossen.

Mit `SELECT` liest der Server die Mailbox ein und weist den Nachrichten mit 1 beginnend Nummern zu.

Auswahl eines Bulletin Boards

Das Protokoll unterstützt explizit die Nutzung von Bulletin Boards. Es stehen analoge Kommandos wie für die Mailbox zur Verfügung. Mit `FIND BBOARDS` liefert der Server eine Liste der unterstützten Boards, mit `BBOARD` wird eines geöffnet.

Ein Standard-Bulletin-Board existiert nicht.

Zugriff

Sechs Kommandos ermöglichen den Zugriff auf die Nachrichten einer geöffneten Mailbox bzw. eines geöffneten Bulletin Boards. Die Nachrichten werden über eine *sequence* adressiert. Diese enthält eine oder mehrere Nachrichtennummern bzw. Bereiche von Nummern. Die Bereiche müssen in aufsteigender Folge angegeben werden. (eine gültige *sequence* kann sein: 1 3 5-7).

`FETCH sequence data:` Es werden Daten für den lokalen Cache angefordert. Mit `data` wird die Art der Daten spezifiziert. Dies kann sein:

- die gesamte Nachricht im Format[822] oder als *envelope*
- Teile der Nachricht: der Header im Format[822] oder der Body
- Informationen über die Nachricht: gesetzte Flags, Größe, Eingangsdatum

`STORE sequence action flags:` Flags können gesetzt, gelöscht oder hinzugefügt werden.

[7] Das sind die Zugriffsrechte, die Zahl aller und die Zahl der neuen Nachrichten.

`SEARCH criteria:` In der Mailbox oder dem Bulletin Board werden Nachrichten gesucht, die den Suchkriterien entsprechen. Es kann in Teilen oder in der gesamten Nachricht nach Worten gesucht werden, Zeiten könne verglichen werden und gesetzte Flags überprüft werden. Eine mögliches Kriterium ist beispielsweise: Nachricht enthält im Body ein bestimmtes Wort, ist vor einem angegebenen Datum eingegangen und das Flag *ANSWERED* ist nicht gesetzt. Es werden 24 Kriterien unterstützt, die teilweise miteinander kombiniert werden können.

`COPY sequence Mailbox:` Nachrichten werden in eine andere Mailbox kopiert.

`EXPUNGE:` Alle Nachrichten mit dem gesetzten Flag *DELETED* werden gelöscht.

`CHECK:` Die Mailbox wird erneut gelesen. Das Kommando entspricht einem `SELECT` auf die bereits geöffnete Box.

`NOOP:` Der Server antwortet mit einem OK.

Verbindungsabbau

Mit `LOGOUT` wird die Sitzung beendet.

Szenario

Das Szenario ist in [1176] zu finden.

```
    Client                          Server
    ------                          ------
                                {Wait for Connection}
{Open Connection}        -->
                              <-- * OK IMAP2 Server Ready
                                {Wait for command}
A001 LOGIN Fred Secret   -->
                              <-- A001 OK User Fred logged in
                                {Wait for command}
A002 SELECT INBOX        -->
                              <-- * FLAGS (Meeting Notice \Answered
                                          \Flagged \Deleted \Seen)
                              <-- * 19 EXISTS
                              <-- * 2 RECENT
                              <-- A0002 OK Select complete
                                {Wait for command}
A003 FETCH 1:19 ALL      -->
                              <-- * 1 Fetch (......)
                                      ...
                              <-- * 18 Fetch (......)
                              <-- * 19 Fetch (......)
                              <-- A003 OK Fetch complete
                                {Wait for command}
A004 FETCH 8 RFC822.TEXT -->
                              <-- * 8 Fetch (RFC822.TEXT {893}
                                      ...893 characters of text...
                              <-- )
                              <-- A004 OK Fetch complete
                                {Wait for command}
```

```
A005 STORE 8 +Flags \Deleted -->
                              <-- * 8 Store (Flags (\Deleted
                                                   \Seen))
                              <-- A005 OK Store complete
                                  {Wait for command}
A006 EXPUNGE               -->
                              <-- * 19 EXISTS
                              <-- * 8 EXPUNGE
                              <-- * 18 EXISTS
                              <-- A006 Expunge complete
                                  {Wait for command}
A007 LOGOUT                -->
                              <-- * BYE IMAP2 server quitting
                              <-- A007 OK Logout complete
{Close Connection}         --><-- {Close connection}
                                  {Go back to start}
```

Vorteile

- Durch die Auswahl einer Mailbox ist es möglich, flexiblere Anwendungen zu erstellen. Denkbar ist es, mehreren Nutzern den Zugriff auf eine Mailbox zu erlauben, während nur einem Schreiben, hier also das Löschen erlaubt wird.
- Die Mailbox bleibt während einer Sitzung verfügbar für Updates.
- Durch die Unterstützung von Flags kann der Nutzer die Nachrichten besser verwalten. So können sie auch über das Sitzungsende hinaus (z.B. als gelesen) markiert werden. Ebenso ist eine Unterscheidung zwischen neuen und alten Nachrichten möglich. Mit diesen Informationen können Nachrichten gezielter bezogen werden.
- Mit einem Kommando können über die *sequence* mehrere Nachrichten bearbeitet werden.

Nachteile

- Sehr viel Intelligenz liegt im Server. Mit den umfangreichen Methoden des Suchens bietet er damit auch Gelegenheit, den Serverhost stark zu belasten, vor allem wenn davon ausgegangen wird, daß gleichzeitig mehrere Clients zugreifen.
- Das Passwort wird mit `LOGIN` unverschlüsselt übertragen. Das birgt ein potentielles Sicherheitsrisiko.
- Obwohl als ein Entwurfsziel formuliert wurde, die Netzlast gering zu halten, ist ein Body nur im Ganzen übertragbar. Selbst eine eingeschränkte Methode wie `TOP` in POP3 wird nicht angeboten.

2.2.3 Zusammenfassung

POP3 ist ein einfaches Protokoll, es hält vor allem den Server von umfangreichen Aufgaben frei. Als Hauptmangel wird empfunden, daß der Nutzer außerstande ist, Informationen über den Inhalt von Nachrichten zu beziehen, ohne diese übertragen zu müssen.

IMAP2 bietet dabei weitere Unterstützung. Der große Nachteil ist die Benutzung eines asynchronen Protokolls. Prinzipiell ist das mit den RPC - Implementierungen realisierbar, doch birgt dies einen zusätzlichen (nicht notwendigen) Aufwand.

Es soll deshalb ein weiteres vorgeschlagen werden.

2.2.4 Entwurfsziele/Voraussetzungen

- Methoden zur Verschlüsselung von Daten sind zumeist Bestandteil der RPC - Implementierung, sie sollen nicht in das Protokoll Eingang finden.
- Mit der wachsenden Verbreitung von nichttextbasierten Nachrichtenformaten (Multimedia) werden die Nachrichtenkörper zunehmend größer. Das Protokoll soll Möglichkeiten zur Verringerung der Netzlast anbieten.
- Es wird das Format aus [822] benutzt. Es ist weit verbreitet und bietet mit Erweiterungen auch Dienste an, die über einfache Textnachrichten hinausgehen. Es wird davon ausgegangen, daß dieses Format auch in absehbarer Zukunft eine dominierende Stellung einnehmen wird.
- Die Verwaltung von Nutzern und deren Zugriffsrechte soll durch das Betriebssystem verwaltet werden.
- Software enthält im allgemeinen Fehler. Gerade Server die mit umfangreichen Rechten arbeiten, sind ein sensibler Punkt in einem System. Der benutzte Server soll mit den Rechten des Nutzers arbeiten, um diese Gefährdung gering zu halten.
- Der Server soll neben einer Standardmailbox auch frei wählbare benutzen können. Dadurch soll ein breiterer Einsatz möglich sein.
- Das Protokoll soll synchron arbeiten, um eine einfacher Umsetzung mit RPC zu erlauben.

2.2.5 Das Protokoll für RPC

In diesem Abschnitt wird nur das Protokoll vorgestellt. Die benutzten Datenformate sollen nur angedeutet werden, sie werden formal in den jeweiligen Schnittstellenbeschreibungen dargestellt.

Wie üblich in einem synchronen Protokoll sendet der Client jeweils einen Request, der Server antwortet mit einem Response. Jedem Response wird ein Indikator hinzugefügt, der den Erfolg der Abarbeitung beschreibt. Dieser Indikator kann sein: ok, no, bad. Die Semantik ist die der Indikatoren in IMAP2.

Jedem Client wird ein eigener Server zugeordnet, der mit den Rechten des Nutzers abläuft. Zu diesem Zweck ist ein Superserver notwendig, der nach einem erfolgreichem `LOGIN` den eigentlichen Server erzeugt. Der Superserver bedient nur das Kommando `LOGIN`. Bei allen anderen Kommandos wird ein Protokollfehler festgestellt (bad).

Der Client baut eine Verbindung zum Superserver auf. Danach hat er das Kommando `LOGIN` zu benutzen, um den Nutzer mit Namen und Passwort anzumelden. Der Superserver sucht mit dem Namen einen Eintrag in der Passwortdatei und testet das Passwort. Danach instanziiert er den eigentlichen Server mit den Rechten des Nutzers. Er gibt die Adresse des Serverprozesses zurück. Der Client baut die Verbindung ab und eine neue zu *seinem* Server auf. Er benutzt wiederholt `LOGIN`. Der Server vergleicht, ob er mit den Rechten des gerade gemeldeten Nutzers läuft

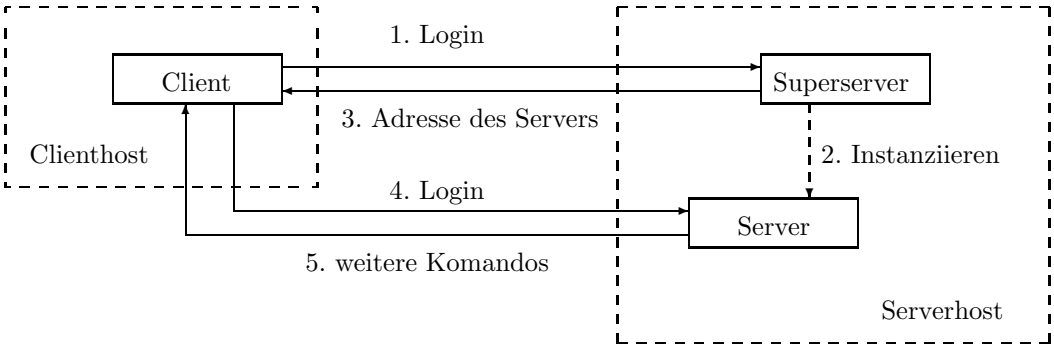


Abbildung 2.4: Die Komponenten des RPC-Protokolls

und gibt in dem Fall eine positive Antwort. Ansonsten erfolgt eine Fehlermeldung und der Server beendet die Arbeit.

Die weitere Kommunikation findet zwischen Client und Server statt.

Die Nachrichten werden über eine *sequence* adressiert. Sie hat die gleiche Struktur wie die *sequence* in IMAP2.

`LOGIN name passwort`

Argumente: Der Name des Nutzers auf dem Serverhost und das dortige Passwort.

Einschränkungen: Auf einem Server ist nur ein Login möglich. Der Superserver erzeugt mit jedem Login einen Server.

Rückgabewerte :

ok:	Adresse des Server (vom Superserver)
no/bad:	lesbare Fehlermeldung

Nach der Anmeldung muß der Client eine Mailbox auswählen. Als Standard wird *INBOX* unterstützt. Der Server sucht dazu ein File mit dem Namen des Nutzers auf dem Serverhost. Alle anderen Namen werden als Filenamen interpretiert. `SELECT` kann beliebig oft nach einem `LOGIN` angewandt werden. Die vorher selektierte Box wird geschlossen, die eventuellen Änderungen werden zurückgeschrieben.

`SELECT Mailbox`

Argumente: Der Name der Mailbox (Standard INBOX)

Einschränkungen: erst nach `LOGIN` möglich

Rückgabewerte :

ok:	Text zu dieser Box (z.B. Lese-/Schreibrechte verfügbar)
	Anzahl der Nachrichten
	Anzahl der neuen Nachrichten
no/bad:	lesbare Fehlermeldung

Informationen über die Nachrichten kann über eine *status list* bezogen werden. Ein Element in der Liste enthält den Header der Nachricht, die gesetzten Flags und die Größe des Body. Der Client ruft `STAT` im allgemeinen nach einem `SELECT`. Die Liste enthält Informationen der Nachrichten, die in dem angegebenen Bereich liegen.

`STAT von bis`

Argumente: Bereich der interessierenden Nachrichten

Einschränkungen: erst nach `LOGIN` und `SELECT` möglich

Rückgabewerte :

ok:	status list
no/bad:	lesbare Fehlermeldung

Jede Nachricht verfügt über Flags, die manipuliert werden können. Es werden die gleichen Flags wie bei IMAP2 unterstützt.

`STORE sequence action flags`

Argumente: Nachrichtennummern, eine Aktion (Flags setzen, rücksetzen oder hinzufügen), eine Liste von Flags

Einschränkungen: erst nach `LOGIN` und `SELECT` möglich

Rückgabewerte :

ok:	keine
no/bad:	lesbare Fehlermeldung

Der Client kann nach Zeichenketten in den Körpern der Nachrichten suchen lassen. Eine Möglichkeit, im Header entfernt nach Informationen zu suchen, existiert nicht.

`SEARCH string`

Argumente: Die gesuchte Zeichenkette

Einschränkungen: erst nach `LOGIN` und `SELECT` möglich

Rückgabewerte :

ok:	*sequence* der gefundenen Nachrichten
no/bad:	lesbare Fehlermeldung

Der Client kann Teile oder den gesamten Nachrichtenkörper anfordern.

`FETCHBODY nr von anz`

Argumente: Nachrichtennummer, Offset im Body und Anzahl der Bytes

Einschränkungen: erst nach `LOGIN` und `SELECT` möglich

Rückgabewerte :

ok:	der (Ausschnitt aus dem) Body
no/bad:	lesbare Fehlermeldung

Die Box wird mit dem Öffnen nur gelesen, aber nicht gesperrt. Während einer Sitzung können neue Nachrichten eintreffen. Der Client kann ein erneutes Lesen der Box anstoßen. Damit werden Änderungen sichtbar.

`CHECK`

Argumente: keine

Einschränkungen: erst nach `LOGIN` und `SELECT` möglich

Rückgabewerte :

ok:	Anzahl der Nachrichten Anzahl der neuen Nachrichten
no/bad:	lesbare Fehlermeldung

Das Löschen einer Nachricht erfolgt in zwei Schritten. Zuerst wird die das Flag *DELETED* gesetzt, ein physisches Löschen findet mit `EXPUNGE` statt.

`EXPUNGE`

Argumente: keine

Einschränkungen: erst nach `LOGIN` und `SELECT` möglich

Rückgabewerte :

ok:	Nummern der gelöschten Nachrichten
no/bad:	lesbare Fehlermeldung

Wird eine Nachricht gelöscht, die nicht die letzte war, so werden die Nummern der folgenden Nachrichten um eins verringert. Sie werden „noch oben geschoben“. Das geschieht nach jedem Löschen einer Nachricht erneut. Die zurückgegebenen Nachrichtennummern beziehen sich auch immer auf diese neuen Nummern. Beispiel: Eine Box enthalte fünf Nachrichten, die letzten drei seien als gelöscht markiert. Mit `EXPUNGE` wird zuerst die dritte Nachricht gelöscht, die vierte erhält die Nummer drei, die fünfte die Nummer vier. Nun wird die neue dritte Nachricht gelöscht usw. Zurückgegeben wird eine Liste der Form 3 3 3.

Die Sitzung wird mit `LOGOUT` beendet. Der Server schreibt die Mailbox bei Veränderungen zurück, sendet einen Grußtext und terminiert.

`LOGOUT`

Argumente: keine

Einschränkungen: erst nach `LOGIN` möglich

Rückgabewerte :

ok:	Abschiedsgruß
no/bad:	lesbare Fehlermeldung

2.2.6 Diskussion

Es wird davon ausgegangen, daß das Betriebssystem auf dem Serverhost, Nutzer und deren Passwörter verwaltet und Zugriffsrechte auf Files über Nutzeridentifikatoren steuern kann, wie es in UNIX üblich ist. Der Server nutzt diese Möglichkeiten und und vermeidet dadurch zusätzlichen Verwaltungsaufwand.

Der Nutzer kann nur auf eine Mailbox zugreifen, wenn er ein *account* auf dem Serverhost besitzt.

Es wird angenommen, daß sich eine Mailbox in einem File befindet. Die Zugriffsrechte auf eine Box sind somit durch die Zugriffsrechte auf Files steuerbar. Auf die persönliche Box sollte der Nutzer Lese- und Schreibrechte besitzen.

Die Bereitstellung von Bulletin Boards wird zwar nicht explizit unterstützt, ist aber einfach möglich. Ein Board kann als Nutzer mit einer Mailadresse auf dem Server eingetragen werden. Dessen Mailbox sollte für die restlichen Nutzer nur lesbar

sein. Hinzugefügt werden Nachrichten einfach durch das Senden an die Mailadresse.

Wie in IMAP2 verfügt der Client über einen lokalen Cache, den er über Datenanforderungen füllen kann. Für die Verwaltung ist er allein verantwortlich. Das schließt ein, daß er nach `EXPUNGE` den Cache selbst aktualisiert.

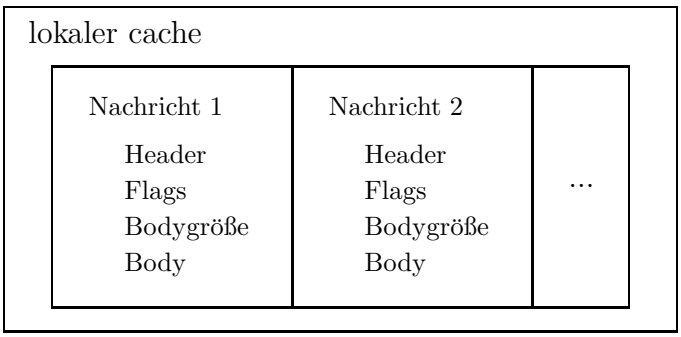


Abbildung 2.5: Der lokale Cache im Client

Eine Buchführung wie bei IMAP2 wird grundsätzlich ausgeschlossen. Das erzwingt etwas mehr Intelligenz im Client. Er sollte Daten nur anfordern, wenn er sie noch nicht besitzt.

Mit der Nutzung des Formates [822] besteht eine Nachricht aus zwei Teilen, dem

Header, der im Vergleich zum Body eher klein ist. Bei interaktiven UA werden dem Nutzer zumindest Teile angezeigt werden müssen (*subject, from*). Es erscheint deshalb sinnvoll zu verlangen, ihn bei Bedarf vollständig zu übertragen. Der andere Teil ist der

Body, der sehr groß werden kann und eventuell aus mehreren Teilen besteht. `FETCHBODY` ähnelt dem Systemruf *read*. Das Kommando erlaubt einen gut skalierbaren Zugriff. Intelligente Clients, die den internen Aufbau eines strukturierten Nachrichtenkörpers unterstützen, sollen damit in der Lage sein, nur die notwendigen Informationen zu beziehen. Diese Fähigkeit geht über die von IMAP2 und POP3 hinaus. Außerdem wird damit das Speichermanagement auf der Clientseite unterstützt, da bereits mit dem Ruf die maximale Anzahl der eingehenden Bytes bekannt ist.

Mit diesem Vorgehen soll ein Mittelweg zwischen den Forderungen Verringerung der Netzlast und Verringerung der Serverlast gegangen werden. Für den Header ist der Client allein zuständig, der Server wird entlastet. Nur der Body kann vor der Übertragung vom Server durchsucht werden.

Der Server ist zustandsbehaftet.
Ein übliches Szenario (ohne Superserver):

```
C: LOGIN thsc pass
S: ok
C: SELECT inbox
S: ok, "read-write access", 3 Nachrichten, 1 neu
```

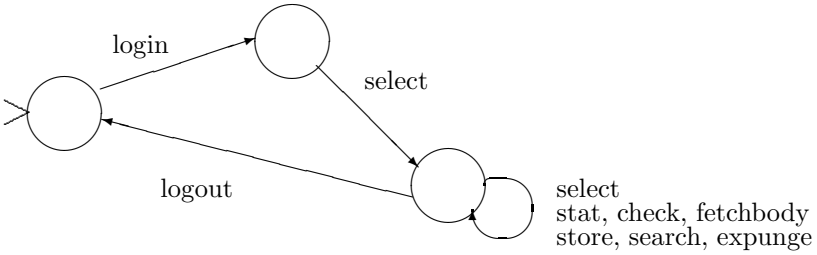


Abbildung 2.6: Protokollzustände

```
C: STAT 1 3
S: ok, status list fuer alle drei Nachrichten
C: SEARCH "ein Wort"
S: ok, 1-2
C: FETCHBODY 1 1 340 (Laenge aus der status list)
S: ok, ... Body ...
C: STORE 1 flags_plus SEEN DELETED
S: ok
C: EXPUNGE
S: ok 1
C: CHECK
S: ok, 2 Nachrichten, 1 neu
C: LOGOUT
S: ok, "save changes, bye"
```

Kapitel 3

DCE-RPC

Das **DCE (Distributed Computer Environment)** der OSF (**Open Software Foundation**) ist ein Betriebsystemaufsatz, der die Nutzung, Verwaltung und Entwicklung von verteilten Anwendungen unterstützt.

Die grundlegende Einheit ist die Zelle **(cell)**. In ihr werden mehrere Hosts vereint. Jede Zelle besitzt

- einen Verzeichnisdienst zur Verwaltung von Objekten dieser Zelle, den **CDS (Cell Directory Service)**. Im weiteren interessieren nur die Objekte, die Informationen eines Servers halten.
- einen **Security Service**, mit dem Principals, deren geheime Schlüssel und ihre Zugriffsrechte auf Objekte im DCE verwaltet werden.
- einen Zeitdienst, der eine einheitliche Zeit in der Zelle zur Verfügung stellt.
- optional einen GDA (**Global Directory Agent**), mit dem Server in anderen Zellen erreicht werden können. Diese Zellen müssen über einen anderen Verzeichnisdienst erreichbar sein (DCE unterstützt DNS (**Domain Name Service**) und GDS (**Global Directory Service**), das sich an den Standard X.500 hält.

Im weiteren sollen nur die Komponenten besprochen werden, die zur Erstellung des entfernten Postzugriffs notwendig waren. Die Abschnitte sollen jeweils mit einem kurzen Überblick über die vorhandenen Möglichkeiten beginnen und danach zeigen, was konkret benutzt wurde. Umfangreiche Hinweise finden sich in [Ken] [Shi] [DEC] [Sch].

3.1 Schnittstellenbeschreibung

Die Definition der Schnittstelle eines Servers erfolgt in zwei Files, einem **idl- (Interface Definition Language)**File und einem **acf- (Attribute Configuration)**File. Im ersten werden die (entfernten) Prozeduren und genutzten Datenstrukturen definiert, im zweiten können weiter Eigenschaften der RPC beschrieben werden, z.B. wird die Art des Bindens angegeben (Abschnitt 3.4) und Angaben wie (Netzwerk) Fehler zu behandeln sind (Abschnitt 3.7).

Das idl-File unterteilt sich in einen Header und einen Body. Im Header wird der Schnittstelle eine eindeutige Nummer zugewiesen, die sie von allen anderen Schnittstellen unterscheidet. Dazu wird eine **UUID (Universal Unique Identifier)** benutzt. Sie ist eine Kombination aus Netzadresse des Hosts und der aktuellen Zeit und ist deshalb im benutzten Netz einmalig. Ebenfalls im Header wird

eine Versionsnummer angegeben, beide Nummern zusammen bilden den **Interface Identifier** (im weiteren auch kürzer Interface ID). Im Prozeß des Bindens spielt dieser Identifier eine wesentliche Rolle.

Einige Anwendungen benutzen feste Endpunkte. Diese können bereits im idl-File vereinbart werden (Abschnitt Binden).

Im Body erfolgen die Definitionen der Prozeduren und der Parameter mit der **IDL**. Diese Sprache ist stark an C angelehnt. Ein wesentlicher Unterschied zu C sind die Attribute, mit denen sich die Datentypen näher spezifizieren lassen. Sie stehen in eckigen Klammern vor den Typen.

An dieser Stelle soll keine weitgehende Erläuterung der verfügbaren Sprachelemente erfolgen. Eine vollständige Beschreibung ist in [DEC] oder [Shi] zu finden. Das idl-File der Anwendung befindet sich im Anhang. Um das Lesen dieser Quellen zu erleichtern, sollen an dieser Stelle nur einige der benutzten Datentypen aufgezeigt werden und wie sie nach der Bearbeitung durch den idl-Compiler in C-Typen übersetzt wurden. Auf der linken Seite der Beispiele stehen die Deklarationen aus dem idl-File, auf der rechten die generierten C-Strukturen.

3.1.1 Konstante

Es können Konstanten eines Typs vereinbart werden. Als Beispiel sollen die Flags des Zugangsprotokolls dienen:

```
const flags F_SEEN     = 2;          #define F_SEEN (2)
const flags F_ANSWERED = 4;          #define F_ANSWERED (4)
const flags F_DELETED  = 8;          #define F_DELETED (8)
const flags F_FLAGGED  = 16;         #define F_FLAGGED (16)
const flags F_RECENT   = 32;         #define F_RECENT (32)
```

Nach der Übersetzung werden Anweisungen für den Präprozessor erzeugt, der Typ findet keine weitere Beachtung.

3.1.2 Pointer

Es werden zwei Arten von Pointern unterschieden. Pointer vom Typ *ref* zeigen auf einen bereits vorhandenen Speicherplatz, wobei zwei Pointer dieses Typs nicht auf den gleichen Datenbereich verweisen dürfen. Das ist die Standardeinstellung für Pointer[1]. Die Angabe kann aber jederzeit auch explizit erfolgen. Als Eingabeparameter werden in der Anwendung des öfteren Strings benötigt. Sie wurden wie folgt definiert:

```
typedef [string, ref] char* cstring_ref;    typedef idl_char *cstring_ref;
```

Diese Typdefinition weicht nur durch die beiden Attribute von der C-Notation ab. Das Attribut `string` kennzeichnet einen Pointer als Verweis auf einen String im C-Format[2].

Der andere Pointertyp ist der **Full Pointer**. Ein solcher Pointer kann außer einem Verweis auf einen Speicherbereich auch den Wert NULL enthalten. Zwei Pointer diesen Typs dürfen auf den gleichen Datenbereich weisen. Wird beispielsweise eine Prozedur mit zwei solchen (Full) Pointern aufgerufen, so wird der referenzierte Datenbereich nur einmal übertragen und die Pointer verweisen auch auf der Serverseite auf die gleichen Speicherstelle.

[1] Diese Standardeinstellung kann mit dem Schlüsselwort *pointer-default* im Header überschrieben werden.

[2] eine Reihe von Zeichen des Typs char, die mit einer NULL abgeschlossen wird

Die dazu notwendigen Tests erfolgen während der Laufzeit, Full Pointer sind wie Pointer in jedem anderen C-Programm benutzbar. Bei Referenzpointern entfallen diese Tests, der Overhead wird vermieden.

Benutzt werden Full Pointer meist bei den Ausgabestrings. Diese enthalten entweder Texte zur Begrüßung (bei `SELECT`) oder Fehlermeldungen, sie müssen aber (z.B. bei erfolgreicher Bearbeitung) nicht übertragen werden.

```
typedef [string, ptr] char* cstring;        typedef idl_char *cstring_ref;
```

3.1.3 Arrays

Es werden drei Arten von Arrays unterschieden:

- **Fixed Arrays** besitzen eine konstante Länge, sie werden vollständig übertragen.
- **Varying Arrays** haben ebenfalls eine konstante Länge, doch kann in der Laufzeit bestimmt werden, welche Teile des Arrays übertragen werden sollen. Dem dienen die Attribute **first_is, length_is** oder **last_is**.

  ```
  void proc(
      [in] long first,
      [in] long len,
      [in, first_is(first), lenght_is(len)] char a[100]
  );
  ```

  In der Laufzeit könnte ein Ruf erfolgen: `proc(10,20, feld);`, wobei aus `feld` nur die Elemente mit dem Index 10 bis 29 übertragen werden.

- Bei **Conformant Arrays** kann während der Laufzeit deren Länge festgelegt werden. Das erfolgt mit einem der Attribute **size_is, max_is**. Bei der Prozedur `fetchbody` wird ein solches Array benutzt.

  ```
  replay_status fetchbody(
      [in] msgnr               nr,
      [in] offset              off,               ...
      [in] offset              anz,
      [out, size_is(anz)] char buffer[],      idl_char buffer[]
      [out] cstring            *text              ...
  );
  ```

  Der `buffer` muß bereits vor dem RPC im Client angelegt sein. Mit `anz` wird damit nicht nur dem Server die Anzahl der gewünschten Zeichen mitgeteilt, die Laufzeitfunktionen erfahren so auch die Länge des Arrays.

3.1.4 Weitere Typen

Basistypen

IDL unterstützt eine Fülle von Basistypen, die sich an die C-Typen anlehnen. Es läßt sich aber genau festlegen, wieviele Bit ein solcher Typ hat. Es werden z.B. vier Integertypen angeboten:(small (8 Bit), short (16 Bit), long (32 Bit), hyper (64 Bit)). Siehe dazu auch [Shi].

Die Nummer einer Nachricht wurde definiert als:

```
typedef unsigned short msgnr;               typedef idl_ushort_int msgnr;
```

Strukturen

Eine Struktur wird wie folgt definiert:

```
typedef unsigned small flags;
typedef struct statentry {
        unsigned long size;
        flags         flags;
        cstring       header;
}
```

```
typedef idl_usmall_int flags;
typedef struct statentry {
        unsigned long   size;
        flags           flags;
        cstring         header;
} statentry;
```

Ein `statentry` ist ein Element der *status list*, die mit dem Kommando `STAT` angefordert wird.

Aufzählung

Eine Aufzählung erfolgt mit **enum**. Die drei möglichen Prozedurwerte werden z.B. wie folgt definiert:

```
typedef enum { ok, no, bad } reply_status;
```

```
typedef enum {ok, no, bad } reply_status;
```

3.1.5 Prozedurdeklaration

Die Prozeduren werden nahezu genauso wie in C deklariert. Es gibt keine Einschränkungen für die Anzahl der Parameter, die Prozedur kann einen beliebigen Wert zurückgeben.

Jedem Parameter ist ein Attribut (`in` oder `out`) vorangestellt, das die Richtung der Übertragung festlegt. So wird das bereits angesprochene Problem gelöst, daß man an der Deklaration einer Prozedur nicht erkennen kann, wie die referenzierten Daten benutzt werden. Eingabeparameter werden vom Client zum Server übertragen. Für die Ausgabeparameter werden der Prozedur die Adresse übergeben auf die diese Ausgabedaten geschrieben werden sollen.

```
typedef unsigned short msgnr;

reply_status select(
        [in]  cstring_ref  mailbox,
        [out] cstring      *text,
        [out] msgnr        *exists,
        [out] msgnr        *recent
);
```

```
typedef idl_ushort_int msgnr;
extern reply_status select(
        handle_t IDL_handle,
        cstring_ref mailbox,
        cstring *text,
        msgnr *exists,
        msgnr *recent,
        error_status_t *status
);
```

Die beiden zusätzlichen Parameter sind das Binding Handle und eine Variable für einen Fehlerkode, dazu später.

Mit `select` wird eine Mailbox ausgewählt, der Name muß zwingend vorhanden sein, deshalb wird der String im C-Format als Referenzpointer benutzt (`cstring_ref mailbox`). Der Ausgabetext enthält Informationen über die Mailbox. Der `text` wird als Doppelpointer deklariert, er verweist also auf eine Speicherstelle, in die der Verweis auf den Text eingetragen werden soll. Der Server hat die Möglichkeit, auf einen solchen Text zu verzichten, er kann NULL anstatt eines Textes übergeben. Wird ein Text gesandt, legt der Clientstub den benötigten Speicher an und übergibt die Adresse in die Variable `text`.

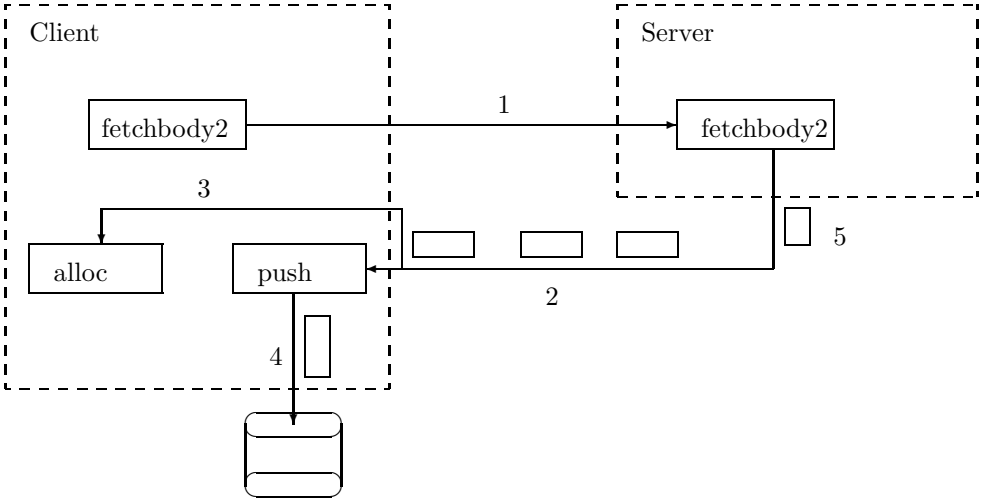
Die beiden letzten Werte sind die Adressen von Integerwerten, in die die Anzahl der existierenden und der neuen Nachrichten eingetragen werden.

3.1.6 Pipe

Der Datentyp `pipe` hat kein Gegenstück in C. Mit ihm ist es möglich, (unbekannt) große Datenmengen oder einen dauerhaften Strom von Daten zu übertragen. Werden die Daten vom Client zum Server transportiert, so wird von einer **Eingabepipe** gesprochen, da sie in der Prozedurdeklaration als Eingabeparameter auftritt, werden Daten vom Server zum Client übertragen, so spricht man von einer **Ausgabepipe**[Sch]. Als Beispiel für die Erläuterung soll die Ausgabepipe dienen.

Der Client muß zwei weitere Prozeduren zur Verfügung stellen. Zum einen eine Prozedur *alloc*, die Speicherplatz zur Verfügung stellt und eine Prozedur *push*, die die eintreffenden Daten bearbeitet.

Der Ablauf unterscheidet sich von anderen RPC. Der Client ruft eine Prozedur mit einem Ausgabeparameter vom Typ *pipe*. Damit tauschen Server und Client für die Dauer der Ausführung die Rollen. Der Server überträgt mit „rückwirkenden Aufrufen“ stückweise die Daten an die Prozedur *push* des Clients. Mit dem Eintreffen des ersten Datenpaketes ruft der Clientstub *alloc*, um Speicher für die Daten zur Verfügung zu stellen. Die Übertragung ist beendet, wenn der Server ein Paket mit keinen Nutzdaten sendet. Siehe Abbildung 3.1.



1. Der Client ruft eine Prozedur mit einer Ausgabepipe.
2. Der Server sendet Datenpakete an den Client.
3. Mit dem Eintreffen des ersten Datenpaketes wird Speicher angefordert.
4. Die Prozedur *push* bearbeitet die Daten, sie können z.B. auf einen Datenträger geschrieben werden.
5. Mit dem Absenden eines leeren Datenpaketes wird das Ende der Daten angezeigt.

Abbildung 3.1: Ausgabepipe

In der Prozedur `fetchbody` hätte eine Pipe zur Anwendung kommen können, hier wird mit großen Datenmengen gearbeitet. Wäre im Client nicht genügend Speicher für den Body vorhanden, könnte er wie in der Abbildung, stückweise übertragen und in ein File geschrieben werden.

In der Anwendung wird aber stattdessen ein Conformant Array benutzt, siehe Deklaration von `fetchbody` in Abschnitt 3.1.3. Soll der gesamte Body übertragen

werden, wird versucht für ihn Speicher anzulegen. Gelingt dies nicht, wird ein temporäres File angelegt, in das der Body geschrieben wird. Er wird nun in Teilen angefordert:

```
/* fetchbody */
if(!(body = malloc(size_of(body_len)))) {
    fp = open(TMPFILE, O_RDWR);
    body = malloc(BODY_PART_LEN);        /* entspricht alloc */
    anz = body_len;
    off = 0;
    while(anz) {
        clt_fetchbody(body,msgnumber,off,MIN(anz, BODY_PART_LEN))
        i = strlen(body);
        off += i;
        anz -= i;
        fwrite(tmp, 1, i, fp);      /* entspricht push */
    }
    ...
}
```

Diese Version benutzt ebenfalls ein *alloc* und ein *push*, nur steuert der Client die Übertragung. Im Gegensatz zur Pipe werden einige Aufrufe eingespart, das sind der Aufruf der Prozedur mit einer Ausgabepipe als Parameter und das letzte (leere) Datenpaket des Servers.

Messungen haben ergeben, daß die Übertragungsdauer einer Datenmenge mit einer Pipe länger dauert, als mit einer Lösung ohne [Bric]. Das erscheint, der zusätzlichen Aufrufe wegen nicht verwunderlich.

In der Anwendung wurde auf deren Einsatz deshalb verzichtet. Besonders sinnvoll scheint eine Pipe vor allem bei (auf dem Server) unregelmäßig eintreffenden Daten oder bei Daten zu sein, deren Erzeugung auch im Server längere Zeit in Anspruch nimmt. In diesem Fall sind die Datenpakete anfangs von unbekannter Größe, der Server kann Daten versenden, sobald er sie besitzt.

3.2 Werkzeug

Zur Übersetzung der so definerten Größen wird der Compiler *idl* benutzt. Er erzeugt die Stubs und ein Headerfile mit den erzeugten C-Strukturen, siehe Abbildung 3.2.

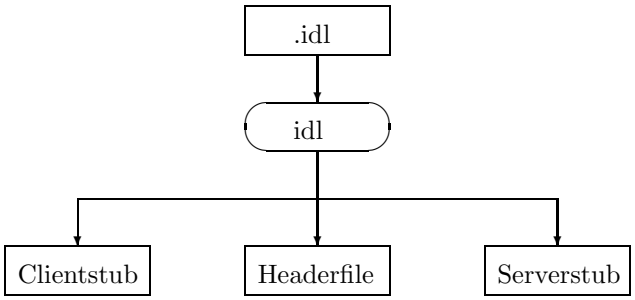


Abbildung 3.2: IDL-Compiler

3.3 Einordnung in das OSI-Referenzmodell

Das Protokoll des RPC regelt den Ablauf einer Sitzung, es ist demnach in die Schicht 5 einzuordnen. Die Datenkonvertierung — Aufgabe der Schicht 6 — erfolgt nicht über ein unabhängiges Format, der Absender übermittelt die Daten in seinem Format, der Empfänger wandelt sie bei Bedarf in das eigene um, siehe Abbildung 3.3

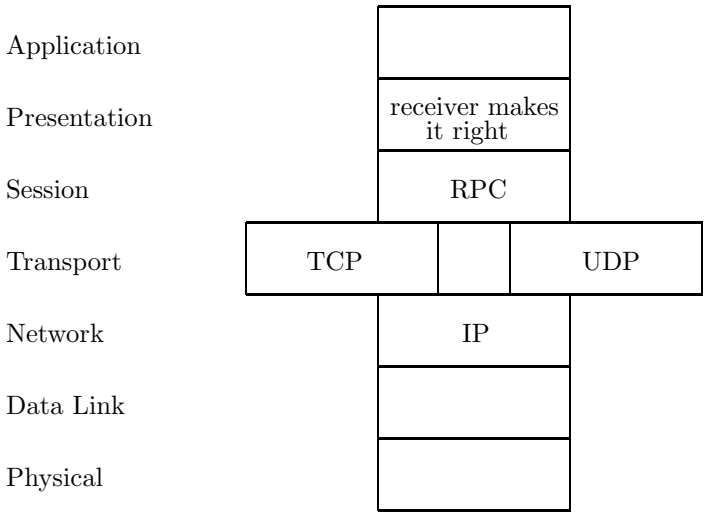


Abbildung 3.3: Einordnung in das OSI-Referenzmodell

3.4 Binding

Das CDS verwaltet ein hierarchisches Verzeichnis, vergleichbar dem UNIX-Filesystems. Im CDS werden Serverinformationen gehalten. Dazu wählt der Server eine Stelle im Verzeichnisbaum aus und legt dort einen Eintrag an. In der Anwendung trägt sich ein Superserver beispielsweise direkt unter der Root der Zelle ein:

```
/.:/rpcmail_alpha
```

Die Zellroot wird mit `/.:` gekennzeichnet, `rpcmail_alpha` ist der Servereintrag. Unter diesem Eintrag sind die Bindeinformationen des Servers enthalten, das sind die folgenden.

- Ein **Interface Identifier** definiert die angebotene Schnittstelle. Er wurde bereits in der Schnittstellenbeschreibung definiert und ist dem Client damit bekannt. Mit ihm wird im CDS nach einem passenden Server gesucht.
- Die **Adresse des Servers** besteht aus der Adresse des Serverhosts, dem benutzten Transportprotokoll und dem Endpunkt des Servers auf dem Host. Mit dem Internet Protokoll ist dieser Endpunkt als *port* bekannt.
- Mit einem **Objekt** kann der Server bekanntgeben, eine bestimmte Ressource zu unterstützen. Dieser Mechanismus ist sehr flexibel einsetzbar, da hier nur eine UUID eingetragen wird. Daran ist keine Semantik gebunden.

- Die **Transfersyntax** beschreibt das Format in dem die Daten über das Netz übertragen werden. Derzeit wird in DCE nur eine Syntax unterstützt. Die Datenkonvertierung bleibt dem Programmierer vollständig verborgen.

Man sagt, der Server **exportiert** die Bindeinformationen. Später kann der Client diese Informationen **importieren**. Das Verzeichnis ist verteilt, es werden an verschiedenen Stellen durch Caches die Leseoperationen beschleunigt. Um zu häufige Inkonsistenzen und zeitaufwendige Schreiboperationen zu vermeiden, sollten nur langlebige Informationen exportiert werden.

Üblicherweise wird einem Server beim Start ein Endpunkt vom Laufzeitsystem zugeordnet (**dynamic Endpoint**). Auf das Exportieren eines solchen Endpunktes sollte verzichtet werden, er ändert sich mit jedem Neustart des Servers, ist also nicht langlebig. Einige (meist zu DCE gehörende Server) benutzen stets den gleichen Endpunkt (**well-known Endpoint**). Dieser sollte in das CDS eingetragen werden.

Der Server exportiert zusätzlich seine *vollständigen* Bindeinformationen in die **Local Endpoint Map** seines Hosts. Diese Tabelle wird von einem Dämon (rpcd) verwaltet. Hier werden auch die dynamischen Endpunkte eingetragen. Siehe Abbildung 3.4.

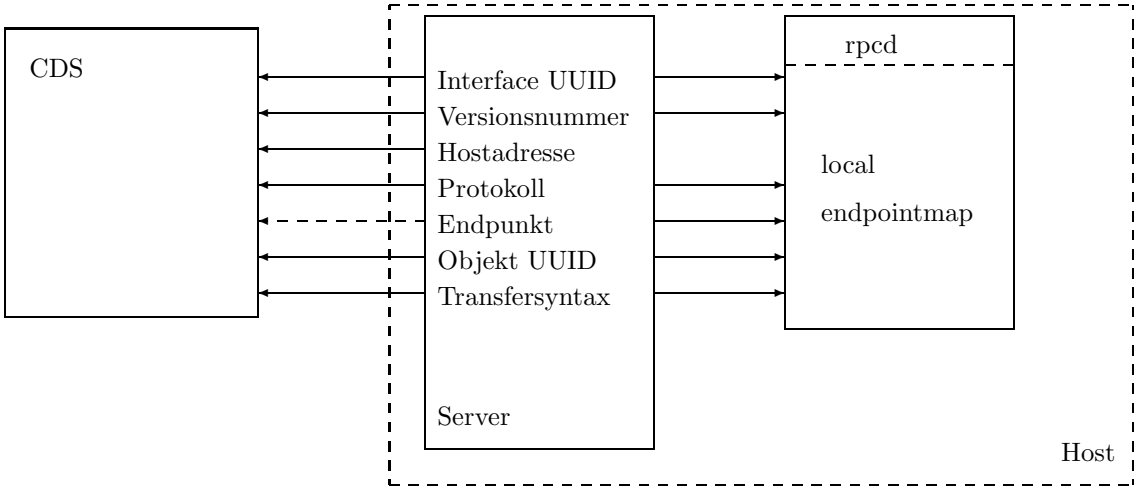


Abbildung 3.4: Server exportiert Bindeinformationen

Der Programmierer kommt mit diesen Bindeinformationen zunächst nicht direkt in Kontakt, sie werden in den Stubs gehalten und über ein **Binding Handle** benutzt. Ist der Endpunkt in den Bindeinformationen bereits enthalten, wird von einem **Fully Bound Binding Handle** gesprochen, ansonsten von einem **Partially Bound Binding Handle**.

Will ein Client einen Server benutzen, benötigt er ein Fully Bound Binding Handle. Die Interface ID besitzt er bereits durch die Schnittstellenbeschreibung.

Wie gelangt der Client an die restlichen Informationen ?

Der Client besitzt bereits die vollständigen Bindeinformationen.
Diese erhält er über Konfigurationsfiles oder sind anderweitig bekannt. Damit kann ein vollständiges Handle ohne weiteres gebildet werden (`rpc_string_binding_compose`). In der Anwendung wird so vorgegangen, wenn der Client die Adresse *seines* Servers ermittelt.

Der Client kennt alle Informationen außer den Endpunkt.
Der Client kann ebenfalls mit `rpc_string_binding_compose` ein Partially

Bound Binding Handle konstruieren. Der fehlende Endpunkt wird beim ersten Aufruf einer entfernten Prozedur automatisch vom Clientstub über den rpcd der Serverhosts angefordert. Das entspricht dem Vorgehen bei ONC - RPC. Die Aufgabe des Portmappers erfüllt hier der rpcd.

Der Client kennt nur den Namen des Servereintrags im CDS. Die RPC-API bietet Funktionen zum Importieren der Bindeinformationen an. Der Client benutzt die Interface ID und den Namen des Servereintrags, um die Suche zu beginnen.

`rpc_ns_import_begin`

Nacheinander kann der Client nun Binding Handle des Servers beziehen. Unterstützt der Server mehrere Transportprotokolle, erhält der Client für jedes ein Binding Handle[3].

`rpc_ns_import_next`

Der Client kann nun eines der Handle auswählen, z.B. anhand des Transportprotokolls.

Ist das Handle ein Fully Bound Binding Handle, kann er den Server direkt erreichen, sonst muß wieder der Weg über den rpcd gegangen werden.

Mit dem CDS benötigt der Client keine Kenntnis über die Adresse des Servers, damit können Server transparent für den Client den Host wechseln. Mit einem GDA können auch Server in einer anderen Zelle erreicht werden. Der GDA stellt dabei lediglich die Verbindung zu einem CDS-Server in der entfernte Zelle her, der über die notwendigen Informationen verfügt.

Das ist der übliche Weg und wird in der Anwendung vom Client bei der Suche nach dem Superserver benutzt.

Binding Handle im Client

Dem Clientprogrammierer stehen drei Möglichkeiten zur Verfügung, die binding Handle zu verwalten. Die Auswahl wird im acf-File getroffen, indem dem Interfacenamen ein Attribut vorangestellt wird.

```
/* TYP ist automatic, implicit oder explizit */
[TYP_handle] interface rpcmail
```

`[automatic_handle]` Dies ist die Standardeinstellung. Dem Programmierer bleibt sowohl das Binding Handle, als auch dessen Ermittlung verborgen. Mit dem ersten Aufruf einer entfernten Prozedur erstellt das Laufzeitsystem *automatisch* ein Handle. Dazu wird in der Umgebungsvariable `RPC_DEFAULT_ENTRY` der Name eines CDS-Eintrags erwartet. In diesem Eintrag wird nach einem Server gesucht, der die Interface ID anbietet, die der Client benötigt. Das erste Handle, das dieser Bedingung genügt, wird ausgewählt. Der Programmierer hat keinerlei Einfluß auf diese Auswahl. Alle weiteren RPC werden an diesen gefundenen Server gesandt.

Sollte die Verbindung unterbrochen werden, so wird erneut ein Server gesucht. Diese Methode kann nur benutzt werden, wenn es ohne Belang bleibt, an welchen Server die einzelnen Rufe gehen. Zustandsbehaftete Server scheiden damit aus.

[3] Ist der Eintrag kein Servereintrag, sondern ein Gruppeneintrag, so erhält der Client nacheinander alle Binding Handle der in der Gruppe enthaltenen Server.

`[implicit_handle]` Im acf-File wird der Name einer globalen Variablen vereinbart. Diese Variable wird im Client-Stub gehalten, sie muß vor dem ersten RPC mit einem gültigen Handle gefüllt werden. Das ist Aufgabe des Programmierers.

Das Handle kann aus dem CDS importiert werden oder wenn die notwendigen Informationen vorliegen (Host, Transportprotokoll und eventuell Endpunkt), kann ein Binding Handle konstruiert werden.

Das so bezogene Handle muß in die globale Variable eingetragen werden. Es wird bei den folgenden RPC (implizit) genutzt.

`[explicit_handle]` Jedem RPC wird das Handle (explizit) als erster Parameter übergeben. Es unterliegt damit vollständig der Verwaltung des Clients. Ein Handle wird wie im impliziten Fall erklärt ermittelt.

Der Client kann mit jedem Ruf entscheiden, an welchen Server dieser Ruf gehen soll, d.h. welches Handle er benutzen will.

3.4.1 In der Anwendung

In der Anwendung wird mit zwei Servern gearbeitet. Der Superserver soll von Clients zellweit erreichbar sein, der Server soll nur von *seinem* Client gefunden werden.

Wie empfohlen arbeitet der Superserver mit einem dynamischen Endpunkt, er unterstützt alle Protokolle, die Nutzung eines Objektes war nicht notwendig. Er exportiert die Hostadresse, die Protokolle und die Interface ID in das CDS. In der benutzten DCE - Implementierung werden zwei Protokolle unterstützt (UDP, TCP). Dem Client bietet das CDS also zwei Binding Handle an, eines für jedes Protokoll (die Hostadresse etc. ändert sich natürlich nicht). Er muß eines davon auswählen.

Das CDS unterstützt neben einfachen Einträgen für Server zwei weitere Arten: Gruppen und Profiles. Eine Gruppe enthält ein oder mehrere Einträge. Diese Einträge sind entweder Server- oder Gruppeneinträge. Gruppen können dazu genutzt werden, Server die das gleiche Interface bedienen unter einem Namen verfügbar zu machen. Konkret wurde so vorgegangen:

Der Superserver erzeugt einen CDS - Eintrag. Dieser hat das Format `rpcmail_`*hostname*. Er wird in die Gruppe `rpcmail` eingefügt. Damit ist es möglich, mehrere Superserver in der Zelle zur Verfügung zu stellen, die die Clients über einen Namen `rpcmail` erreichen können. Es ist Sache des Client zwischen den Superservern auszuwählen.

Danach trägt er sich zusätzlich in die lokale Endpunkttabelle ein.

Nach einem erfolgreichen (ersten) `LOGIN` wird der Server mit den Rechten des Nutzers instanziiert. Der Superserver stellt dazu fest, mit welchem Protokoll der Client mit ihm kommuniziert. Der Server wird im weiteren nur dieses Protokoll unterstützen. Mit nur einem Protokoll wird ihm auch nur ein Endpunkt dynamisch zugewiesen. Der Client entscheidet also allein, welches Transportprotokoll im weiteren zum Einsatz kommen soll.

Wie gelangt der Client an das Fully Bound Binding Handle des Servers ?

Der Client besitzt nach dem ersten `LOGIN` ein vollständiges Binding Handle des Superservers. Es ist bekannt, daß der Server auf dem gleichen Host läuft, er das gleiche Protokoll unterstützt und kein Objekt verwaltet. Es fehlt nur der Endpunkt.

Das erscheint Aufgabe des rcpd zu sein. Der Server könnte sich in der lokalen Endpunkttabelle anmelden, der Client könnte wie üblich eine Verbindung herstellen. Superserver und Server unterstützen aber ein und dasselbe Interface. Trägt sich der Server ohne weiteres ein, überschreibt er den Eintrag des Superservers.

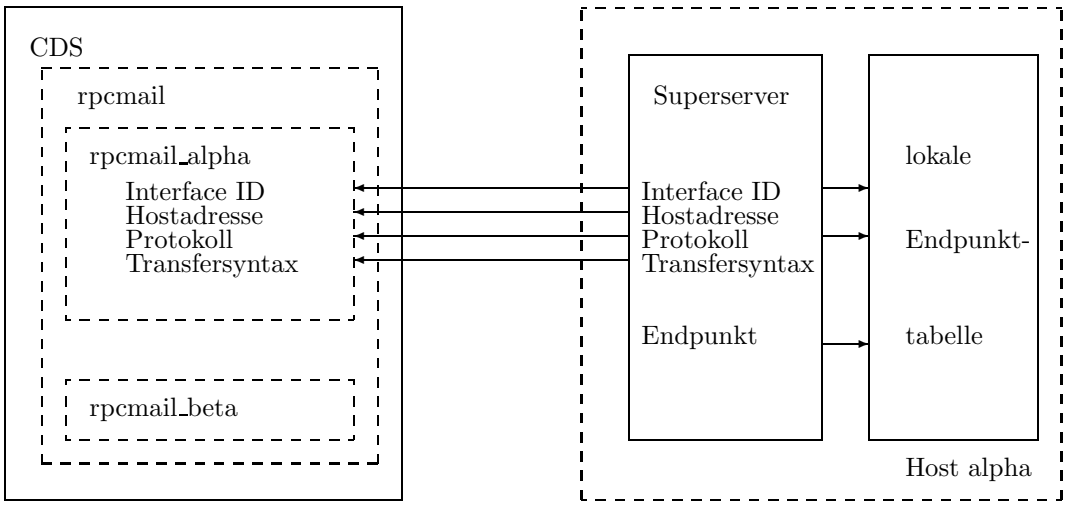


Abbildung 3.5: Superserver export Binding Informationen

In ONC wird mit variablen Programmnummern gearbeitet. Analog könnte mit einer variablen Interface ID gearbeitet werden. Dieses Vorgehen wird in DCE sehr erschwert, die Interface ID wird vor allem von Laufzeitfunktionen genutzt und bleibt dem Programmierer verborgen.

Eine weitere Möglichkeit die Server zu unterscheiden, ist ein Objekt. Der Superserver könnte ein Objekt unterstützen, dessen UUID bereits in der Schnittstellenbeschreibung definiert würde. Mit der Instanziierung generierte der Superserver eine weitere UUID. Diese könnte dem Server als Parameter übergeben werden. Der Server würde diese UUID als Objekt benutzen und meldete sich damit bei dem rpcd an. Die alten Einträge blieben unberührt. Der Client erhielte als Rückgabewert die UUID des Servers und könnte über den rpcd den fehlenden Endpunkt ermitteln.

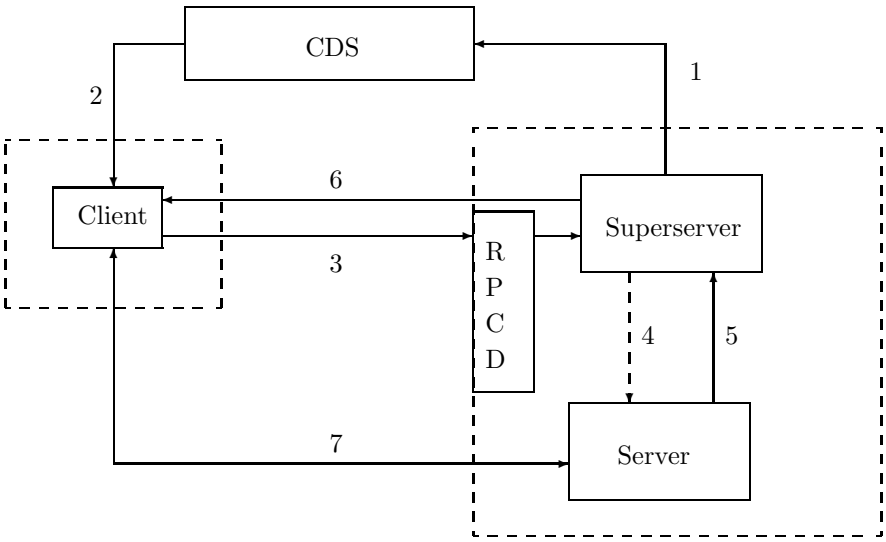
Um den Ablauf weiter zu vereinfachen wurde eine andere Möglichkeit gewählt. Der Server läßt sich nach der Instanziierung einen dynamischen Endpunkt zuweisen. Diesen übermittelt er (über eine UNIX-Pipe) dem Superserver, der den neuen Endpunkt dem Client sendet. Der Server trägt sich weder in das CDS noch in die Endpunkttabelle ein. Der Client besitzt nun den fehlenden Endpunkt und kann einen Fully Bound Binding Handle generieren, um den Server direkt zu erreichen. Siehe Abbildung 3.6

Binding Handle Verwaltung im Client

Das automatische Handle ist für die Anwendung ungeeignet. Der Superserver ließe sich zwar noch mit dieser Methode ermitteln, der Server trägt sich aber nicht in das CDS ein, kann also auch nicht automatisch gefunden werden.

Zur Auswahl stehen nur implizite und explizite Methode. Performanceuntersuchungen haben Unterschiede zwischen impliziten und expliziten Handle festgestellt, bei unterschiedlichen Parametern erschien aber einmal die explizite, einmal die implizite im Vorteil [Bric].

Für die Anwendung erscheint das implizite Handle als das geeignete. Es wird zu einer Zeit entweder der Superserver oder der Server angesprochen, niemals beide gleichzeitig. Der Quellcode wurde aber zu großen Teilen auch auf den anderen Plattformen benutzt. In ROSE und ONC-RPC muß aber mit einem, dem expliziten



1. Der Superserver exportiert Bindeinformationen.
2. Der Client erhält ein Partially Bound Binding Handle des Superservers.
3. Über rpcd findet der Client den Superserver und ruft `LOGIN`.
4. Der Server wird instanziiert. Über einen Parameter wird das zu benutzende Transportprotokoll festgelegt.
5. Nach erfolgreicher Initialisierung übergibt der Server seinen Endpunkt an den Superserver.
6. Der Client erhält den Endpunkt und generiert ein Full Bound Binding Handle für den Server.
7. Ohne weitere Zwischenschritte läuft das weitere Protokoll ab.

Abbildung 3.6: Verbindungsaufbau in der Beispielanwendung

Handle vergleichbaren Parameter gearbeitet werden, der mit jedem RPC übergeben wird. Da das explizite Handle keine Nachteile gegenüber dem impliziten zeigt, wurde es auch hier benutzt.

3.5 Authentication / Security

Der **Security Service** ist Bestandteil jeder DCE-Installation. Er unterteilt sich in drei Teile.

- Der **Registry Service** verwaltet **Principals**. Ein Principal ist ein Nutzer, ein Host oder ein Server. Der Service bietet an, Principals in Gruppen und in einer größeren Einheit, einer Organisation zu verwalten. Mit dem Programm *rgy_edit* können Einträge manipuliert werden. Die zugehörige API besteht aus den Prozeduren beginnend mit sec_rgy_ [DEC].
- Der **Authentication Service** ist eine Implementierung des Kerberos-Systems. Er verwaltet *secret keys* für die Principals. Ein direkter Zugriff auf die Prozeduren des Kerberos ist nicht möglich, sie sind undokumentiert und

werden indirekt bei Authenticated RPC benutzt. Mit drei Werkzeugen (kinit, klist, kdestroy) können Schlüssel manipuliert werden.

- Der **Privilege Service** verwaltet Zugriffsrechte auf Objekte in einer Zelle. Dazu werden **ACL (Access Control Lists)** benutzt. Der Service wird im wesentlichen dazu benutzt, DCE-eigene Objekte wie Teile des CDS zu verwalten. Ein interaktiver Zugriff ist über *acl_edit* möglich. Eine API existieren nicht. Der Service wird während eines DCE-login benutzt und im Authenticated RPC.

Es besteht für den Nutzer die Möglichkeit, Dienste des DCE zu nutzen ohne als Principal in der Zelle bekannt zu sein (**Unauthenticated User**).

Der Zelladministrator kann dieser Gruppe wie anderen Principals Rechte zuordnen. Im allgemeinen wird ihr das Recht zum Lesen im CDS gegeben. Damit können Bindeinformationen über Server ermittelt werden. Für die Anwendung bedeutet das, daß ein Nutzer des Mailservers in der Zelle nicht als Principal bekannt sein muß. Die im weiteren vorgestellten Möglichkeiten der Erhöhung der Sicherheit sind damit aber nicht anwendbar.

Der Zelladministrator kann einen Nutzer als Principal eintragen. Aus dem Passwort generiert der Security Service einen *secret key*. Mit jedem Einloggen in das DCE erhält der Nutzer dann ein *ticket*. Dieses *ticket* benutzen die Nutzerprogramme, um sich gegenüber anderen Diensten zu authentifizieren, z.B. um bestimmte Zugriffsrechte zu erlangen.

Während ein Client im allgemeinen von einem Nutzer gestartet wird, dessen ticket er benutzt, liegt wird bei Servern meist anders vorgegangen. Server sind nicht an bestimmte Nutzer gebunden, sie sind eigenständige Principals. Server besitzen aber kein Passwort aus dem ein *secret key* zu erstellen wäre. Der Authentication Service verwaltet deshalb eine weitere Tabelle mit *secret keys* für solche Principals (**Keytable**).

Nach der Anmeldung im Security Service können sich Client und Server wechselseitig authentifizieren. Wird diese Möglichkeit genutzt, so spricht man von **Authenticated RPC**, Standard ist der **Unauthenticated RPC**.

Der Programmierer hat sowohl im Client als auch im Server nur eine Prozedur zu benutzen, um ein ein Securitylevel zu setzen.

3.5.1 Server

Will der Server Authenticated RPC anbieten, muß er als Principal im Security Service eingetragen sein. In der Anwendung benutzen beide Server den Principalnamen `rpcmail`. Ohne den Eintrag scheitert der folgende Aufruf.

```
rpc_server_register_auth_info(principal, authn_svc,
get_key_fn,arg,status);
```

- Der Parameter `principal` ist der eigene Principalname.
- Mit dem zweiten Parameter kann ein Verschlüsselungsverfahren ausgewählt werden. Es gibt vier Möglichkeiten.
  - `rpc_c_auth_none`
    es wird kein key benutzt.
  - `rpc_c_auth_default`
    ist ein zellweiter Standard
  - `rpc_c_auth_dce_secret`
    Kerberos

  - `rpc_c_auth_dce_public`
    wurde bisher nicht implementiert

  Der Server kann mehrere dieser Varianten benutzen, indem er die Prozedur mehrfach benutzt. Da DCE momentan nur die Auswahl zwischen `rpc_c_auth_none` und `rpc_c_auth_dce_secret` anbietet, ist das nicht notwendig, es genügt ein einfacher Aufruf mit `rcp_c_auth_dce_secret`. Die Alternative `rpc_c_authn_none` wird als Standard immer angeboten.

- Die folgenden beiden Parameter lassen dem Programmierer einen Weg offen, einen eigenen Schlüssel zur Verfügung zu stellen. Diese Möglichkeit wurde nicht benutzt. Siehe dazu [DEC].

- Der letzte Parameter gibt Auskunft über den Erfolg der Ausführung.

Will der Server mit Kerberos arbeiten, muß er als Principal einen *secret key* in der Keytable besitzen. Die Einträge können mit rgy_edit erfolgen. Der Eintrag eines neuen Principals und die Generierung eines *secret keys* für einen Server erfolgt mit dem Editor `rgy_edit`. Die Keytable des Authentication Service ist eine sehr sensible Stelle des Systems. Standardmäßig hat nur *root* Schreibrechte auf dieses File. Bei einem Neueintrag muß die Benutzung von rgy_edit sowohl unter Rechten des Principals cell_admin, als auch unter den rechten von *root* ablaufen.

In der Anwendung werden die beiden verfügbaren Möglichkeiten angeboten. Dieses Vorgehen ist einfach, da keinerlei zusätzlicher Kode benötigt wird und es bleibt dem Client überlassen, eine geeignete Methode auszuwählen. Eine eigene Schlüsselverwaltung ist nicht notwendig.

Wie auf den anderen Plattformen muß auch hier der Superserver mit root- Rechten laufen, um dem Server mit einer beliebigen UID starten zu können.

3.5.2 Client

Der Client kann mit jedem Binding Handle Informationen über die zu nutzenden Sicherheitstufen halten, das heißt er kann mit jedem Server eine andere Strategie vereinbaren.

Zum Festlegen eines Securitylevel wird

```
rpc_binding_set_auth_info(principal, level, authn_svc,
identity, authz_svc, status);
```

gerufen.

- Als `principal` wird der Name des Servers (`rpcmail`) erwartet.

- Mit dem `level` kann der Client entscheiden, welche von sieben Sicherheitstufen er benötigt. In jeder nächst höheren Stufe werden die Methoden der unteren weiterhin benutzt. Das RPC wird somit schrittweise sicherer und langsamer.

  0. `rpc_c_protect_level_none`
     Unauthenticated RPC
  1. `rpc_c_protect_level_default`
     es wird ein zellweiter Defaultwert genutzt
  2. `rpc_c_protect_level_connect`
     Während der Verbindungsaufnahme authentifizieren sich Client und Server wechselseitig **(Mutual Authentication)**.

3. `rpc_c_protect_level_call`
   Eine Mutual Authentication findet mit jedem RPC statt. Die RPC werden in den unterliegenden Protokollen in (mehreren) Paketen übertragen. In dieser Stufe wird die Prüfsumme des ersten Paketes verschlüsselt.
4. `rpc_c_protect_level_pkt`
   Die Verschlüsselung findet in allen Paketen statt.
5. `rpc_c_protect_level_integrity`
   Neben der Prüfsumme werden auch die Nutzerdaten verschlüsselt.
6. `rpc_c_protect_level_privacy`
   Die gesamte Nachricht wird zusätzlich verschlüsselt.

Der Client kann unabhängig vom Server die gewünschte Sicherheitsstufe einstellen. In der Anwendung wird es dem Nutzer überlassen, eine geeignete Stufe zu finden. Standard ist der Unauthenticated RPC. Die Einstellung geschieht über das Konfigurationsfile mit dem Eintrag *securitylevel*. In dem Eintrag wird eine Nummer zwischen 0 und 6 erwartet, s.o. Wird ein Wert größer Null gewählt, wird als Verschlüsselungsmethode die des *secret key* verwendet.

- Generell erbt ein Prozeß die Identität des Nutzers, soll diese überschrieben werden kann `identity` benutzt werden. Dazu muß im Authentication Service ein Identity Handle erlangt werden [DEC]. In der Anwendung wird generell die Identität des Nutzers benutzt. Eine Änderung hätte auch keine weiteren Auswirkungen. Der Server arbeitet mit den (UNIX-) Rechten des Nutzers, unabhänig von dessen Identität in DCE.
- Mit `authz_svc` kann bestimmt werden, worüber der Server Informationen über Authorisierung erlangen soll, das wird nicht genutzt.
- Der letzte Parameter beschreibt den Erfolg der Abarbeitung.

3.6 Authorization

Der Security Service verwaltet Zugriffsrechte auf Objekte mit dem Privilege Service. Prinzipiell ist es auch jedem Server möglich, Zugriffsrechte auf seine Objekte darüber zu organisieren. Dazu muß ein eigener **ACL-Manager** geschrieben werden, zum anderen sollte noch eine Reihe von Prozeduren unterstützt werden, die die Administration dieser ACL-Manager erlauben. Dies erscheint als sehr umfangreiche Aufgabe, daß selbst in [DEC] abgeraten wird, eigene Manager zu konstruieren.

In der Anwendung werden Zugriffsrechte mit den Methoden des unterliegenden Betriebssystems verwaltet, nicht mit Werkzeuge des DCE.

3.7 Fehlerbehandlung

Die Behandlung von zwei Fehlerarten soll in diesem Abschnitt erläutert werden. Kommunikationsfehler sind Fehler, die das benutzte unterliegende Protokoll meldet, im Fall DCE-RPC ist das das Transportprotokoll. Die andere Fehlerquelle ist ein Fehler im Server. Gemeint ist ein Fehler der zu einer Ausnahmebehandlung (*exception*) führt. Daneben gibt es noch Fehler die die Anwendung generiert, z.B. wenn eine Nutzer/Passwort - Kombination falsch ist. Von diesen Fehlern soll nicht die Rede sein.

DCE unterscheidet beide Fehlerarten und bietet Möglichkeiten zu deren Behandlung. Mit einem Eintrag im acf-File kann ein zusätzlicher Ausgabeparameter vereinbart werden, der einen Fehlercode enthält. Dieser Parameter erscheint in jeder Prozedur als letzter Parameter und wird vom Laufzeitsystem mit Werten gefüllt. In der Anwendung wurden beide Fehlerarten angefordert.

```
select([comm_status, fault_status] status);
```

Daraus entsteht als Prozedurdeklaration (rpcmail.h):

```
select(
    /* [in] */     handle_t          IDL_handle; /* explizites Binding Handle */
    /* [in] */     cstring           mailbox,
    /* [out] */    cstring           *text,
    /* [out] */    msgnr             *exists,
    /* [out] */    msgnr             *recent,
    /* [out] */    error_status_t    *status    /* enthaelt Fehlercode */
);
```

Im Client muß nun mit jedem Aufruf ein Zeiger auf einen Speicherbereich vom Typ `error_status_t` übergeben werden, in diesen trägt der Clientstub einen Fehlercode ein oder die Konstante `rpc_s_ok`. Die RPC-API bietet Prozeduren, um einen lesbaren Fehlertext zu erzeugen (`dce_error_inq_text`).

Oben wurden beide Fehlerarten in einen Parameter gelenkt, es können auch zwei getrennte benutzt werden.

Wird eine Fehlerart nicht in dieser Art angefordert, so erzeugt der Clientstub stattdessen eine *exception*. In der vorliegenden Implementation wurde das Signal SIGABRT generiert.

3.8 Speichermanagement

An zwei Stellen kommt der Entwickler mit der RPC-eigenen Speicherverwaltung in Kontakt. Zum einen werden die Bindeinformationen in den Stubs gehalten. Das Binding Handle ist nur ein Verweis auf eine Struktur. Werden Informationen aus dem Handle benötigt, so wird vom Stub neuer Speicher angefordert und mit den interessierenden Werten gefüllt. Korrespondierend zu den anfordernden Prozeduren gibt es Prozeduren zur Freigabe des Speichers.

Beispiel:

```
rpc_server_inq_binding();
rpc_binding_vector_free();
```

Diese Prozeduren werden in Servercodes benutzt. Die erste fordert alle Binding Handle aus dem Stub an, die dieser unterstützt, ein **Binding Vector**. Die zweite gibt den Speicher für diesen Vektor wieder frei. Es gibt eine Reihe anderer Paare [DEC] [Shi].

Die Stubs stellen außerdem Speicher für die Rückgabewerte bereit.

3.8.1 Client

Die Eingabeparameter müssen wie in üblichen lokalen Prozeduren zu Verfügung gestellt werden. Die Rückgabeparameter sind prinzipiell Verweise auf bereits angelegten Speicherplatz, wie es auch in C üblich ist. Das Prozedurergebnis ist entweder ein einfacher Wert, dann wird er in die benutzte Variable eingetragen oder ein Verweis auf einen Speicher, dann wird der Speicher vom Clientstub angelegt.

Beispiel:

Deklaration im idl-File:

```
char* proc(
      [in]          char            *in_text,
      [out]         char            *out_text,
      [out]         char            **new_text
);
```

Deklaration im generierten Headerfile:

```
char* proc(         /* implizites Binding, kein zusätzlicher Fehlerstatus */
      /* [in] */    char            *in_text,
      /* [out] */   char            *out_text,
      /* [out] */   char            **new_text
);
```

Im Clientprogramm:

```
{
      ...
      char          *in_text = "Eingabetext";
      char          out_text[TEXTLEN], *new_text, *ret;
      ...
      ret = proc(in_text, out_text, &new_text);
      ...
      free(new_text);
      free(ret);
}
```

Der Eingabeparameter (`in_text`) ist ein üblicher Verweis auf einen String. Interessanter sind die Rückgabewerte. Für `ret` wird Speicher im Clientstub angelegt, der später freigegeben werden kann. Mit `out_text` wird vor dem RPC Speicher bereitsgestellt. Dem RPC wird nur der Verweis auf den Speicher übergeben. Für `new_text` legt der Clientstub ebenfalls Speicher an und legt den Verweis darauf in `new_text` ab.

3.8.2 Server

Die Eingabeparameter verwaltet der Serverstub. Wird ein Ruf empfangen, wird für sie Speicher zur Verfügung gestellt, dieser wird mit den Daten gefüllt. Kehrt die Prozedur zurück, wird der so bereitgestellte Speicher freigegeben.

Sind in einer Prozedur nur Eingabeparameter und werden für die Ausgabe nur Referenzpointer definiert, ist kein weiteres Speichermanagement notwendig.

Erwartet der Client aber neu bereitgestellten Speicher (`ret, new_text`), muß der Server diesen anlegen, bevor er ihn mit Werten füllt. Nach Beendigung der Arbeit wird diese Referenz auf neuen Speicher zurückgegeben. Bei einem lokalen Prozeduraufruf besteht darin kein Problem, die Adreßräume liegen zusammen. Der Server allokiert Speicher, der Client benutzt ihn und gibt ihn später wieder frei. Im RPC liegen die Adreßräume auseinander. Der Server hat nach dem `return` keine Möglichkeit mehr, diesen Speicher freizugeben.

```
      ...
      char          *ret;
      ...
      ret = (char*)malloc(LEN);
```

```
      return(ret);
      /* Speicher ist angelegt, Server hat Kontrolle abgegeben */
```

In ONC-RPC muß mit den vorhandenen Systemrufen dieses Problem bearbeitet werden. Im Kapitel ONC wird eine Lösung vorgestellt.

DCE-RPC stellt eine einfache Möglichkeit für solcherart angelegten Speicher zur Verfügung. Es stehen einige **Stub Support Routines** zur Verfügung, eine soll besprochen werden.

```
rpc_ss_allocate()
```

Diese Funktion fordert Speicher innerhalb einer Serverprozedur an. Der Speicher wird nach dem Absenden der Rückgabewerte wieder freigegeben. Die Prozedur `proc` von oben kann wie folgt bearbeitet werden:

```
/* Serverstub */
/* allokiere Speicher für in_text und fülle ihn mit “Eingabetext“ */
/* allokiere Speicher für out_text */
/* Ruf der Serverprozedur */
proc(in_text, out_text, new_text)
      char          *in_text, *out_text, **new_text;
{
      ...
      char          *ret = “Returntext“;
      char          *text;
      ...
      text = rpc_ss_allocate(TEXTLEN);
      strcpy(text, “neuer Text“);
      *new_text = text;
      ...
      strcpy(out_text, “Ausgabetext“);
      ...
      return(ret);
}
/* Von Serverprozedur zurückgekehrt */
/* Serverstub */
/* übertrage die Rückgabeparameter out_text, new_text, ret zum Client */
/* gebe über rpc_ss_allocate bezogenen Speicher frei (new_text) */
/* gebe out_text frei */
/* gebe in_text frei */
```

3.9 Aufrufsemantik

Im idl-File ist es möglich, jeder Prozedur über ein Attribute eine Aufrufsemantik zuzuordnen. Die benutzte Einteilung unterscheidet sich von der aus Abschnitt 1. Sie lassen sich aber teilweise zuordnen. Einfluß auf die Semantik haben nur Kommunikationsfehler, das schließt an dieser Stelle eine Unterbrechung der Verbindung und den Absturz des Servers mit ein. Unter Fehler wird in diesem Abschnitt nur diese Fehlerart verstanden. Vier Arten der Semantik werden unterschieden.

- Wird die Semantik einer Prozedur nicht weiter benannt, so wird versucht, sie *genau einmal* aufzurufen. Kehrt der RPC ohne Fehler zurück, so wurde diese Semantik erreicht. Ansonsten wurde sie *höchstens einmal* aufgerufen, d.h. es ist nicht entscheidbar, ob der Ruf oder die Quittung verlorenging.

- **idempotent**
  Eine Prozedur ist idempotent, wenn sie bei den gleichen Eingabeparametern stets die gleichen Ausgabeparameter liefert. Ist dieses Attribut gesetzt, so können die Laufzeitfunktionen im Fehlerfall mehrfach den Ruf absetzen.
  Liefert eine idempotente Prozedur einen Wert zurück, ist sichergestellt, daß die Prozedur *wenigstens einmal* ausgeführt wurde. Im Fehlerfall kann keine Aussage über die Häufigkeit der Ausführung gemacht werden.

- **broadcast**
  Der Ruf wird an alle Hosts im lokalen Netzwerk gesandt. Es muß ein broadcastfähiges Protokoll benutzt werden, in der vorliegenden Implementierung heißt das UDP (oder im DCE Format: ncadg_ip_udp[4]). Der Client erhält das erste eintreffende Ergebnis, die anderen werden verworfen. Einer Semantik aus Kapitel 1 ist broadcast nur schwer zuzuordnen. Erhält der Client Rückgabewerte wurde die Prozedur *genau einmal* auf einem unbekannten Host ausgeführt. Im Fehlerfall können entweder die Rufe oder die Quittungen verloren gegangen sein oder beides.
  Diese Art des Rufes entfernt sich auch stark von der Semantik eines *Prozedur*aufrufes. Es ist eher ein *Prozeduren*aufruf.

- **maybe**
  Die Prozedur liefert keinen Rückgabewert, es gibt keine Information über den Erfolg der Ausführung. Das ist eine eindeutige *höchstens einmal* Semantik. Wie bereits oben erwähnt, wird damit bewußt die Forderung nach Totality umgangen. An einigen Stellen mag diese Semantik Sinn haben. Denkbar ist eine Anwendung, die regelmäßig Werte verteilt. Nur wenn ein Empfänger bereits längere Zeit keine Werte mehr erhielt, könnte er sich an diesen Service wenden und eine (einmalige) sicherere Übertragung verlangen.

In der Anwendung sind alle Rufe idempotent, außer `LOGIN`, `LOGOUT`, `EXPUNGE` und `CHECK`.

`CHECK` könnte aber problemlos wie eine idempotente Prozedur behandelt werden. Sie liefert nicht immer die gleichen Werte, falls neue Nachrichten in der Mailbox eintreffen, ermittelt sie die aktuellen Zahlen. Diese aktuellen sind aber interessant. Gingen die alten dabei verloren, hätte das keine negativen Auswirkungen.

Bei `EXPUNGE` liegt der Fall genau andersherum, hier ist die Ausgabe des ersten Aufrufes interessant. Bei weiteren Rufen würden zwar keine weiteren Nachrichten gelöscht werden, doch die Ausgabe wäre unkorrekt.

Es wurde allerdings keine Prozedur als idempotent deklariert. Es ist unklar, wann ein wiederholtes Senden benutzt wird. Bei Nutzung von TCP sorgt das Transportprotokoll für einen sicheren Datenstrom, bei UDP garantiert das RPC die gleiche Semantik Wie konkret vorgegangen wurde, ist unbekannt.

Denkbar ist, daß die Idempotenz einer Prozedur zur Optimierung benutzt wird. So müßte im Fall UDP auf der Clientenseite nur ein Timer laufen. Bleibt eine gewisse Zeit die (vollständige) Rückantwort aus, könnte der Ruf wiederholt werden. Auf der Serverseite müßte bei den *nicht*idempotenten Prozeduren über die bearbeiteten Rufe Buch geführt werden, um die wiederholte Ausführung zu verhindern. Bei idempotenten ist das nicht nötig. Da die Interna des DCE-RPC nicht offengelegt sind, kann nur spekuliert werden.

[4] network computing architecture datagram protcol, internet protocol, udp

Es wird aber davon ausgegangen, daß in einem bestimmten Fehlerfall das wiederholte Rufen einer Prozedur stattfindet. Bei einer idempotenten Prozedur `FETCHBODY` hätte das den Effekt, daß dem offensichtlich gestörten oder überlasteten Netz weitere Daten übergeben werden. Wie lange ein wiederholtes Rufen erfolgt, ist ebenfalls unbekannt.

Ein anderes Problem entsteht bei `CHECK`, das die Mailbox auf dem Serverhost komplett aus dem zugehörigen Files neu einliest. Sollten oft die Quittungen verlorengehen und sollte damit die Prozedur wiederholt gerufen werden, so wird der Serverhost durch das wiederholte Einlesen zusätzlich belastet.

Ziel des Entwurfes war es, die Netz- und die Serverlast gering zu halten. Da unbekannt ist, wann und wie oft ein erneutes Rufen stattfindet, wurde konsequenterweise bei allen Prozeduren darauf verzichtet. So bleibt es in der Verantwortung des Clients mit Netzwerkfehlern umzugehen.

Kapitel 4

ONC-RPC

ONC (Open Network Computing) bezeichnet eine Reihe von Produkten der Firma SUN. Im weiteren werden davon die XDR und RPC mit dessen Compiler *rpcgen* besprochen.

Das Kapitel soll mit der Definition der im Zugriffsprotokoll notwendigen Größen beginnen.

4.1 Schnittstellenbeschreibung

Zur Beschreibung der Schnittstelle wird eine dem C ähnliche Sprache benutzt, die **RPCL (RPC Language)**. Sie ist eine Erweiterung der XDR - Beschreibungssprache und wird in [1057] vorgestellt. Eingehende Beschreibungen befinden sich auch in [Blo].

Die Beschreibung der Schnittstelle erfolgt standardmäßig in einem File mit der Extension *.x*. Das File rpcmail.x befindet sich im Anhang. In der Beschreibung werden Konstanten, Datenstrukturen und Prozeduren deklariert. Der angebotenen Schnittstelle wird eine eindeutige Nummer und eine Versionsnummer gegeben, um sie von anderen zu unterscheiden. Diese Nummern werden im Prozeß des Bindens benötigt, siehe Abschnitt 4.5.

Dieser Abschnitt soll nicht als Einführung in RPCL dienen, es soll vielmehr gezeigt werden, wie die für das Zugangsprotokoll notwendigen Größen deklariert wurden, und wie sie von dem Werkzeug *rpcgen*, siehe Abschnitt 4.1, in C-Strukturen umgesetzt werden. Auf der linken Seite der Quelltextbeispiele wird die Struktur aus rpcmail.x auf der rechten Seite aus rpcmail.h erscheinen.

4.1.1 Flags

Das Protokoll unterstützt Flags, die in jeder Nachricht gespeichert werden. Sie werden mit `STAT` vom Server bezogen und mit `STORE` manipuliert. Sie werden als Konstante vereinbart.

```
const F_SEEN    = 2;
const F_ANSWERED= 4;
const F_DELETED = 8;
const F_FLAGGED = 16;
const F_RECENT  = 16;
```

```
#define F_SEEN    = 2;
#define F_ANSWERED= 4;
#define F_DELETED = 8;
#define F_FLAGGED = 16;
#define F_RECENT  = 32;
```

Nach der Übersetzung erscheinen die Konstanten als Anweisungen für den C-Präprozessor.

```
typedef unsigned flags;
```

```
typedef u_int flags;
```

Durch ein bitweises ODER können die Flags zu einer Liste von Flags verknüpft werden. Diese kann eine Variable vom Datentyp `flags` aufnehmen.

4.1.2 Nachrichtennummern

Jeder Nachricht wird beim Öffnen der Mailbox eine eindeutige Nummer vom Server zugeordnet. Die Nummern beginnen mit 1.

```
typedef unsigned msgnr;
```

```
typedef u_int msgnr;
```

Der Client adressiert Nachrichten in einem Kommando als Sequence. Das ist eine Reihe von Nummernbereichen, von Intervallen.

```
struct interval {
    msgnr from;
    msgnr to;
};

typedef interval sequence<>;
```

```
struct interval {
    msgnr from;
    msgnr to;
};
typedef struct interval interval;

typedef struct {
    u_int    sequence_len;
    interval *sequence_val;
} sequence;
```

Mehrere Intervalle ergeben die Sequence. Sie wurde als variables Array vereinbart. Die Länge dieses Arrays wird erst zur Laufzeit angegeben. Dies geschieht einfach, indem neben einem Zeiger auf die Werte (`sequence_val`) eines Arrays auch dessen Länge in einer Struktur gehalten wird. Es werden in der Laufzeit jeweils nur `sequence_len` Werte aus dem Array übertragen.

Ähnlich wird mit der *status list* vorgegangen, die mit dem Kommando `STAT` vom Server bezogen werden kann. Diese Liste besteht aus einzelnen Einträgen.

```
struct statentry {
    unsigned size;
    flags   flags;
    string  header<>;
};
```

```
struct statentry {
    u_int   size;
    flags   flags;
    char    *header;
};
typedef struct statentry statentry;
```

Der Header einer Nachricht wird ebenfalls als variables Array aber vom Typ `string` deklariert. Damit wird es in der Laufzeit als ein Zeichenkette im C-Format interpretiert, die Längenangabe ist mit der abschließenden 0 implizit enthalten.

Die *status list* selbst ist wieder ein variables Array vom Typ `statentry`. Analog wird auch die Ausgabe des Kommandos `EXPUNGE` definiert. Es ist variables Array vom Typ `msgnr`, siehe Abschnitt B.

4.1.3 Indikatoren

Der Server übergibt mit jeder Antwort einen Fehlerindikator an den Client, der angibt, ob die Prozedur erfolgreich ausgeführt wurde (`ok`), ob das Protokoll nicht eingehalten wurde (`bad`) oder ob ein Fehler auftrat (`no`).

```
enum reply_status {
    ok, no, bad
};
```

```
enum reply_status {
    ok = 0,
    no = 1,
    bad = 2,
};
typedef enum reply_status reply_status;
```

4.1.4 Kommandos

Die Kommandos werden als Prozeduren vereinbart. In RPCL gilt die Einschränkung, daß eine Prozedur nur einen einzige Eingabeparameter besitzen darf und die Ausgabe nur über das Resultat erfolgt.

Mehrere Werte können mit Strukturen übergeben werden. Beispiel:

```
/* Request */
struct login_req {
    string          user<>;
    string          passwd<>;
};

/* Response */
union login_reply switch(reply_status status) {
    case ok :       long   prgnr;           /* neue Programmnummer */
    default :       string text<>;
};
```

`LOGIN` erwartet einen Nutzernamen und ein Passwort (`login_req`).

War der Prozeduraufruf erfolgreich, wird der Indikator `ok` und das Resultat übertragen, im Fehlerfall (`no, bad`) eine Fehlernachricht. Dieser Zusammenhang läßt sich mit der `union` gut ausdrücken. Sie ähnelt eher einem varianten Record aus PASCAL als einer Union aus C.

In Abhängigkeit von einer Variablen (`status`, der Indikator) enthält das Resultat eine der im case-Teil aufgeführten Strukturen.

Nach der Übersetzung entsteht:

```
struct login_reply {
    reply_status status;
    union {
        long prgnr;
        char *text;
    } login_reply_u;
};
typedef struct login_reply login_reply;
```

Die Prozedur selbst kann nun vereinbart werden:

```
login_reply LOGIN(login_req) = 1;
```

Per Konvention sind die Prozedurnamen mit großen Buchstaben anzugeben. In der Laufzeit werden die Prozeduren anhand der angegeben Nummern unterschieden.

Daraus entsteht:

```
#define LOGIN ((u_long)1)
extern login_reply *login_1();
```

Dem Prozedurnamen wird die Version hinzugefügt. Diese Prozedur kann auf bei Client als Stubprozedur benutzt und muß im Server zur Verfügung gestellt werden.

4.1.5 Programm

Sämtliche angebotenen Prozeduren werden in einer Struktur *program* zusammengefaßt. Der Schnittstelle wird eine achtstellige Hexadezimalzahl und eine Versionsnummer zugeordnet. Beide Nummern sollen auf einem Host einmalig sein. Um Überschneidungen zu vermeiden, wurde der Zahlenbereich aufgeteilt.

Bereich	Verwendung
0x00000000 - 0x1FFFFFFF	definiert von SUN
0x20000000 - 0x3FFFFFFF	nutzerdefiniert
0x40000000 - 0x5FFFFFFF	transient
0x60000000 - 0xFFFFFFFF	reserviert

```
program RPCMAIL_PROG {
    version RPCMAIL_VERSION {
        login_reply LOGIN(login_req) = 1;
            ...
        search_reply SEARCH(string)  = 9;
    } = 1;
} = 0x20000000;
```

Mit der Schnittstellenbeschreibung wurden die entfernten Prozeduren und ihre Datenstrukturen definiert. Es soll nun betrachtet werden, wie Daten in der Laufzeit in ein unabhängiges Format umgewandelt werden, danach wird das Werkzeug vorgestellt, mit dem aus diesen Definitionen die Stubprozeduren und die notwendigen C-Vereinbarungen generiert werden.

4.2 Datenkonvertierung mit XDR

In ONC-RPC werden die Daten aus der internen Darstellung in ein unabhängiges Datenformat umgewandelt, um dann übertragen zu werden. Das benutzte Format wird im **External Data Representation Standard (XDR)** beschrieben, [1014].

Für die Datenkonvertierung steht die XDR-Bibliothek zur Verfügung. In ihr werden Funktionen zur Konvertierung einiger Basistypen angeboten, siehe dazu auch [Blo]. Jede Funktion erwartet als ersten Parameter einen **XDR-Stream**. Mit dieser Struktur wird verborgen, woher bzw. wohin die Daten bei der Konvertierung geschrieben werden. Das ist entweder ein Speicherbereich oder ein Datenstrom. Zur Veranschaulichung soll ein Beispiel dienen:

```
#include <rpc/rpc.h>
#include <stdio.h>
void main() {
    FILE    *fp;
    XDR     xdr_str;
    char    *out_str = "Hello XDR";
    char    *in;

    fp = fopen("file", "w");
    /* XDR-Stream zur Ausgabe */
    xdrstdio_create(&xdr_str, fp, XDR_ENCODE);
    xdr_string(&xdr_str, &out_str, strlen(out_str));
    fclose(fp);

    fp = fopen("file", "r");
```

```
    /* XDR-Stream zur Eingabe */
    xdrstdio_create(&xdr_str, fp, XDR_DECODE);
    xdr_string(&xdr_str, &in, ~0);

    /* Ausgabe von "Hello XDR" */
    puts(in);
    fclose(fp);
}
```

Zur Ausgabe wird ein File geöffnet. Danach wird ein XDR-Stream angelegt, der dieses File benutzt. Der erste Parameter ist ein Verweis auf den neuen XDR-Stream, der zweite ein Verweis auf die Struktur `FILE (fp)`. Der letzte Parameter gibt die Richtung der Konvertierung an. Mit `XDR_ENCODE` wird ein XDR-Stream zur Ausgabe geöffnet, d.h. er konvertiert Werte von der internen Darstellung in das hier angegebene File. Eine Konvertierung findet mit `xdr_string` statt. Der angegebene String (`"Hello XDR"`) wird in das XDR-Format umgewandelt und über den XDR-Stream ausgegeben. Das File enthält nun den String im XDR-Format.

Wird ein XDR-Stream mit `XDR_DECODE` geöffnet, so erfolgt mit `xdr_string` die umgekehrte Konvertierung, es wird aus dem Stream ein String im XDR-Format gelesen und in die interne Darstellung umgewandelt.

4.3 Werkzeug rpcgen

Die Schnittstellenbeschreibung dient als Eingabe für den Compiler *rpcgen*. Die Ausgabe ist für C-Programme direkt nutzbar. Es werden vier Files ausgegeben, siehe Abbildung 4.1.

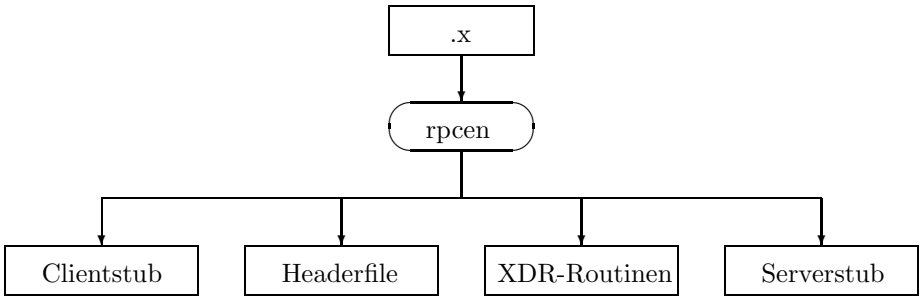


Abbildung 4.1: Arbeit des Generator rpcgen

Headerfile Die Datenstrukturen aus RPCL wurden in äquivalente C-Strukturen übersetzt. Einige Beispiele wurden bereits oben gezeigt.

XDR-Routinen Dieses File enthält für jede deklarierte Datenstruktur, eine eigene XDR-Routine, die diese Datenstruktur zwischen der internen Darstellung und XDR-Format konvertieren kann.

```
bool_t
xdr_login_req(xdrs, objp)
        XDR *xdrs;
        login_req *objp;
{
```

```
        if (!xdr_string(xdrs, &objp->user, ~0)) {
                return (FALSE);
        }
        if (!xdr_string(xdrs, &objp->passwd, ~0)) {
                return (FALSE);
        }
        return (TRUE);
}
```

Das Vorgehen ist einfach. Eine Struktur besteht aus Basistypen, für die es XDR-Routinen gibt. Mit der neuen Routine werden nacheinander die Elemente einer Struktur in den XDR-Stream geschrieben bzw. aus ihm gelesen. Da die XDR-Routinen in beide Richtungen arbeiten, benutzen sowohl Client als auch Server die gleichen.

Clientstub Er enthält für jede deklarierte Prozedur eine Stubroutine, die vom Clientprogramm gerufen wird. Diese Routinen benutzen die Routinen aus dem XDR-File, um die Eingabeparameter zu kodieren und die Rückgabewerte zu dekodieren.

Serverstub Der Serverstub stellt eine `main()`-Funktion zur Verfügung, die eingehende Rufe entgegennimmt. Sie dekodiert die Eingabeparameter, ruft die Serverroutine, kodiert die Rückgabewerte und sendet die Ergebnisse zurück. Dazu werden ebenfalls die XDR-Routinen benutzt.

4.4 Das RPC Protokoll

Das RPC-Protokoll baut auf dem Austausch von nur zwei Strukturen auf, einem Request (Abbildung 4.2) und einem Response (Abbildung 4.3)

XID	Transaction Identifier: eine ID für jeden Request
Typ	= 0 (Request)
Programmnummer	aus der Schnittstellenbeschreibung
Versionsnummer	aus der Schnittstellenbeschreibung
Prozedurnummer	aus der Schnittstellenbeschreibung
Credential	Identifiziert den Client
Verifier	verifiziert Identität des Clients
Argumente	Prozedurargumente, Struktur ist durch Prozedurnummer bekannt

Abbildung 4.2: Format eines RPC-Request

XID	Transaction Identifier des Requests
Typ	= 1 (Response)
Reply Status	Ruf wurde angenommen oder zurückgewiesen
Verifier	verifiziert Identität des Servers
Accept Status	Erfolg der Ausführung
Result	Ergebnisse des RPC

Abbildung 4.3: Format eines RPC-Response

Für beide Strukturen stehen XDR-Routinen zur Verfügung. Ruft der Client eine entfernte Prozedur auf, wird in eine Requeststruktur generiert und über einen XDR-Stream übertragen. Der Response wird über einen Eingabestream empfangen. Analog arbeitet der Server.

Das Protokoll stützt sich ausschließlich auf die benutzten Streams. Dadurch ist es einerseits sehr gut portierbar, andererseits reicht es die Einschränkungen des Transportprotokolls direkt an den Nutzer der RPC-API weiter. Wird beispielsweise UDP benutzt, kann eine Request oder Response maximal die Größe eines UDP-Paketes haben. Ebenso werden keine Vorkehrungen gegen den Verlust von Datenpaketen getroffen.

Für die Anwendung standen UDP und TCP als Transportprotokoll zur Auswahl. Es wird TCP benutzt. Zum einen sollte die Einschränkungen der Paketgröße umgangen werden, zum anderen wird die Programmierung durch das sicherere TCP vereinfacht, da keine Vorkehrungen gegen Datenverlust getroffen werden müssen.

Der Nachteil der Entscheidung ist, daß sich die Datenübertragung unter TCP gegenüber UDP verlangsamt.

Zur Einordnung des Protokolls in das OSI-Referenzmodell siehe Abbildung 4.4

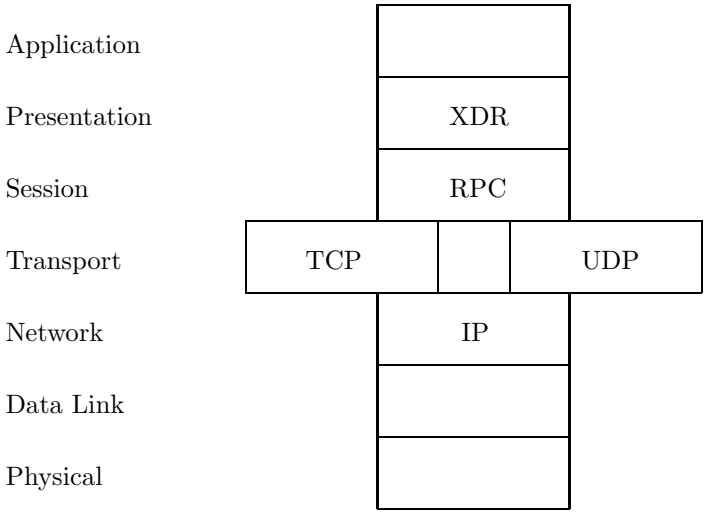


Abbildung 4.4: Einordnung des ONC-RPC in OSI-Schichtenmodell

4.5 Binden

Der Prozeß des Bindens ist nicht Bestandteil der RPC-Spezifikation. Will ein Client mit einem Server Kontakt aufnehmen, benötigt er dessen Adresse. Sie besteht aus drei Teilen.

- Die **Hostadresse** muß dem Client bekannt sein, RPC bietet keinen standardisierten Weg diese Information zu erlangen. In der Anwendung wird in einem Konfigurationsfile ein Host eingetragen oder wird während des Login erfragt.
- RPC unterstützt als **Transportprotokoll** TCP und UDP. Der Client muß sich vor der Verbindungsaufnahme für eines entscheiden. In der Anwendung wird nur mit TCP gearbeitet.
- Über einen **Port** ist der Server auf dem lokalen Host erreichbar. Ein Server kann stets den gleichen, einen **well-know Port** benutzen. Dieser Port sollte von einer zentralen Stelle zugewiesen werden, um Kollisionen zu vermeiden.

  Üblicherweise läßt sich ein Server bei jedem Start einen Port vom System zuweisen.

Nach jedem Start sollte sich ein Server in die **Portmap** seines Host eintragen, sie enthält drei Einträge.

- Programmnummer der Schnittstelle
- Versionsnummer der Schnittstelle
- Port des Servers

Diese Tabelle wird von einem Dämon, dem **Portmapper** verwaltet.

Will ein Client einen Server erreichen, muß er dessen Host kennen und ein Transportprotokoll ausgewählt haben. Arbeitet der Server mit einem well-known Port, kann er direkt die Verbindung herstellen.

Im anderen Fall, kann er dem Portmapper des Serverhost die gewünschte Programm- und Versionnummer übermitteln und erhält den aktuellen Port des Servers, wenn dieser sich angemeldet hat. Zu beachten ist, daß der Portmapper keinen Fehler meldet, wenn die Versionsnummer nicht übereinstimmt. Dieser Fehler wird erst mit dem Aufruf einer entfernten Prozedur bemerkt.

4.5.1 In der Anwendung

In der Anwendung soll der Superserver von allen Clients im Netz erreichbar sein, während der Server nur von *seinem* Client erreichbar sein muß. Wie kann das erreicht werden?

Der Superserver läßt sich beim Start wie üblich einen Port zuweisen und trägt sich in der Portmap ein. Mit dem Instanziieren eines Servers muß dessen Port auch für den Client verfügbar sein. Er sollte sich also ebenfalls in die Portmap eintragen. Die gleiche Programmnummer kann allerdings nicht benutzt werden, da sonst der Eintrag des Superservers überschrieben würde.

Um dies zu vermeiden, wird mit variablen Programmnummern gearbeitet [Hüb1]. Ruft ein Client `LOGIN` mit einer gültigen Name/Passwort - Kombination, so ermittelt der Superserver eine Programmnummer aus dem für transiente Programme vorgeschlagenen Bereich. Dazu versucht er mit `svc_register()` einen Programmnummer zu registrieren. Gelingt dies nicht, d.h. sind alle transienten Nummern vergeben, so scheitert das `LOGIN`. Bei 536870912 Möglichkeiten ist dies höchst unwahrscheinlich. Bei Erfolg wird ein Server instanziiert, diesem wird die neue Programmnummer übergeben.

Der Server gibt diesen Eintrag wieder frei, läßt sich einen Port zuweisen und trägt den Eintrag mit dem neuen Port erneut in die Portmap ein. Jetzt kann der Client den Port des Servers ermitteln. Siehe auch Abbildung 4.5

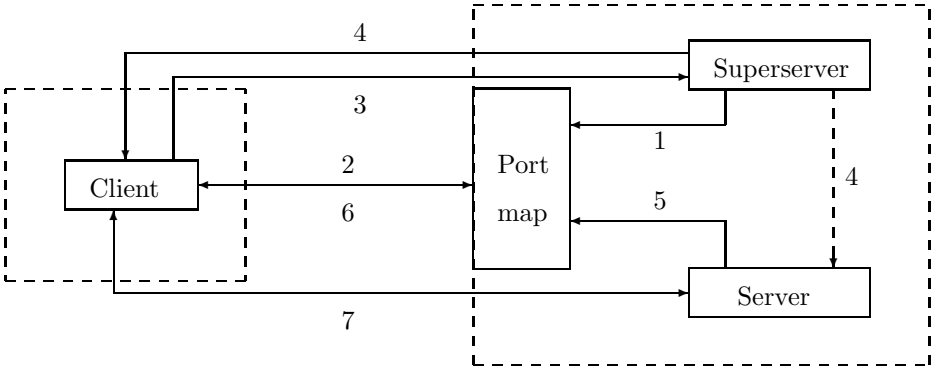
4.5.2 Der Internet Superserver

Ein Server auf RPC-Basis kann auch von dem Superserver *inetd* aufgerufen werden. Dazu ist zusätzlicher Kode notwendig. Dieser kann von einigen Versionen des Generators *rpcgen* durch die Angabe der Option -I erzeugt werden. Diese Möglichkeit wurde nicht in Anspruch genommen.

4.6 Authentication

Mit jedem RPC können Informationen über den rufenden Client übertragen werden. Dazu stehen die Felder *credential* und *verifier* bereit, siehe Abbildung 4.2.

Der Credential identifiziert den Client, mit dem Verifier sind diese Angaben überprüfbar. Umgekehrt kann sich der Server mit jeder Antwort authentifizieren, dazu füllt er den Verifier in der Antwort aus, siehe Abbildung 4.3.



1. Der Superserver trägt sich in die Portmap ein.
2. Der Client ermittelt die Portnummer des Superservers.
3. Der Client ruft LOGIN.
4. Der Superserver hat eine freie Programmnummer ermittelt, gibt sie dem Client zurück und instanziiert den Server.
5. Der Server trägt seinen Port mit der temporären Programmnummer ein.
6. Der Client ermittelt den Port des Servers.
7. Der Client stellt die Verbindung zum Server her und ruft zuerst LOGIN.

Abbildung 4.5: Verbindungsaufbau in der Beispielanwendung

ONC-RPC bietet drei Arten der Authentifizierung. Mit der ersten werden weder der Credential noch der Verifier benutzt. Es kann keinerlei Überprüfung der Identitäten stattfinden. Es werden nun die beiden anderen betrachtet.

4.6.1 UNIX - Authentication

Mit dieser Methode wird nur das Credential benutzt. In ihm werden weitere Informationen zum Client eingetragen. Das sind

- der Zeitpunkt an dem dieses Credential erzeugt wurde
- der Clienthostname
- die effektive UID des Clients auf dem Clienthost
- die aktuelle und alle anderen GID des Clients auf dem Clienthost

Der Verifier bleibt leer. Damit ist es nicht möglich, die Richtigkeit der Angaben zu prüfen, andersherum ist es problemlos möglich, die Angaben eines beliebigen Nutzers in das Credential einzutragen, wenn Leserechte für das Passwort- und Gruppenfile vorhanden sind.

Es werden die UID und GID des Clienthost genutzt. Sie müssen mit den ID im Serversystem nicht übereinstimmen. Denkbar ist die Nutzung zur Buchführung in einem Server, der von mehreren Clients benutzt wird. Überträgt der Client mit jedem Ruf diese Informationen, kann der Server die Häufigkeit der Zugriffe zählen (ohne allerdings sicher sein zu können, daß es sich tatsächlich um den richtigen Client handelt)

Diese Form der Authentifizierung ist von sehr eingeschränktem Nutzen.

4.6.2 DES - Authentication

Der Ablauf eines Protokolls mit DES (**Data Encryption Standard**) soll kurz beschrieben werden, siehe auch [Cor].

Zu Beginn des Datenaustausches erstellt der Client einen Schlüssel, der im folgenden nur ihm und dem Server bekannt sein soll (**Conversation Key**). Der einfachste Weg ist, eine Zufallszahl zu wählen. Das kann aber recht unsicher sein, da diese meist anhand der Systemzeit generiert werden und damit das Erraten erleichtert wird. Ein sicherer Weg besteht darin, den `keyserv`-Dämon einen Schlüssel generieren zu lassen, der auf beiden Hosts verfügbar sein muß, siehe unten. Dazu dient die Funktion `key_gendes`.

Dieser Schlüssel muß dem Server übergeben werden. Ihn im Klarformat zu übertragen verbietet sich natürlich. Client und Server können aber ohne Interaktion einen gemeinsamen Schlüssel ermitteln, den **Common Key**, dazu weiter unten. Der Conversation Key wird nun mit dem Common Key verschlüsselt.

Der Client legt eine Zeit fest, nach dem der Conversation Key ungültig wird (**Window**). Damit wird die Chance ihn zu brechen weiter verringert. Das Window wird mit dem Common Key verschlüsselt.

Mit diesen Werten wird das Credential des ersten Requests aufgebaut. Es hat drei Einträge.

- Netzname des Clients
- Conversation Key verschlüsselt mit dem Common Key
- Window verschlüsselt mit dem Conversation Key

Der Verifier besteht aus zwei Teilen.

- Zeitstempel verschlüsselt mit dem Conversation Key
- Window+1 verschlüsselt mit dem Conversation Key

Credential und Verifier werden mit dem ersten Request zum Server übertragen. Dieser kann mit dem ihm bekannten Common Key den Conversation Key ermitteln und damit das Window aus dem Credential entschlüsseln. Der Verifier dient zur Kontrolle der Angaben.

Der Server hält sich in einer Tabelle den Namen des Clients, den Conversation Key, das Window und den Zeitstempel aus dem Verifier.

Der Response enthält im Verifier eine ID, den *nickname*, der als Verweis auf einen Eintrag in der Servertabelle gesehen werden kann, und er enthält den vom Client übertragenen Zeitstempel verringert um eins und verschlüsselt mit dem Conversation Key. Der Client entschlüsselt den Zeitstempel und stellt damit die Authentität des Servers fest (oder nicht).

Im folgenden benutzt der Client nur die ID als Credential und verschlüsselt einen Zeitstempel. Der Server lehnt einen Aufruf ab, wenn der Zeitstempel älter ist, als der zuletzt eingegangene, womit verhindert wird, daß eine Nachricht aufgezeichnet und wiederholt gesendet die gleichen Aktionen anstößt, oder wenn die Lebensdauer des Credentials (Window) abgelaufen ist, siehe Abbildung 4.6.

In [1057] wird das Verfahren vorgestellt, mit dem Client und Server einen Common Key ermitteln. Dazu benötigen beide eine **Secret Key** und einen **Public Key**, letzterer ist beiden zugänglich. Mit dem Verfahren erhalten Client und Server den gleichen Schlüssel, wenn sie den eigenen Secret Key mit dem Public Key des Partners verknüpfen. Der Public Key wird zentral gehalten[1] und kann über den Dämon `keyserv` bezogen werden. Soll Secure RPC benutzt werden, muß dieser Dämon auf beiden Hosts laufen.

[1] z.B. in einer NIS-Table

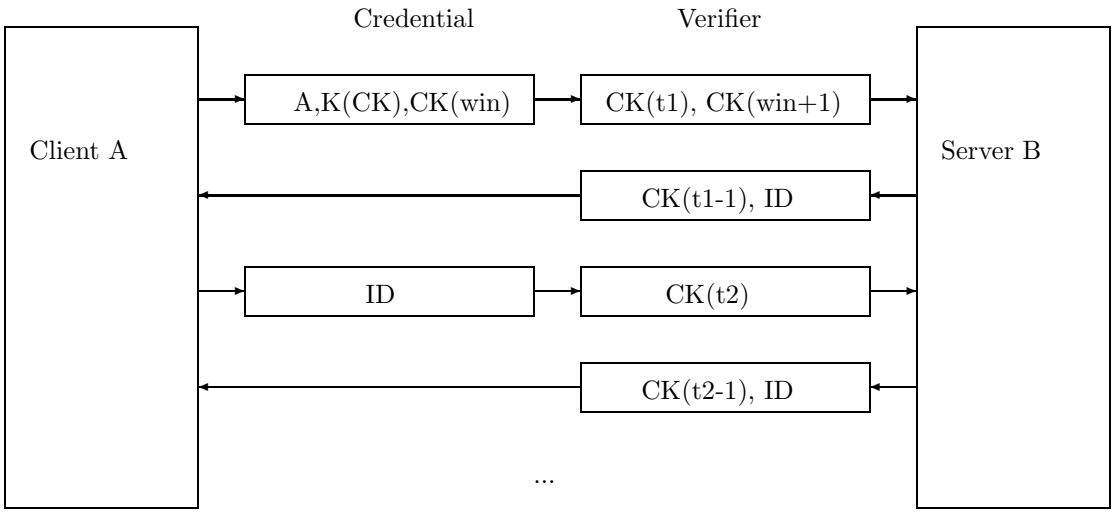


Abbildung 4.6: Secure RPC

Einen Eintrag dieser Schlüsseltabelle kann mit `chkey` vom Nutzer angelegt und geändert werden.

In der Anwendung

DES-Authentication wird in der Anwendung angeboten. Dazu muß im Konfigurationsfile die Wert von *securitylevel* auf die Lebensdauer des Credentials gesetzt werden. Mit dem Eintrag 0 wird kein DES benutzt.

Client Nach der Erzeugung des Clienthandles muß die Form der Authentifizierung festgelegt werden. Dazu wird der Name des Servers benötigt, um den Common Key zu erhalten. Es gibt zwei Möglichkeiten, den Netznamen eines Prozesses zu erlangen, `user2netname` und `host2netname`. Der Superserver muß mit Rechten von *root* arbeiten, dessen Name wird mit `host2netname` erlangt. Der Server arbeitet unter den Rechten des Nutzers auf dem Serverhost. Der Name wird mit `user2netname` erzeugt. Dazu wird allerdings die UID des Nutzers auf dem Serverhost benötigt, diese kann unterschiedlich zu der im Client sein. Dazu wird mit einem erfolgreichen `LOGIN`[2] anstelle einer neuen Programmnummer die UID zurückgegeben.

Server Die Authentifizierung wird dem Server von den Laufzeitroutinen abgenommen. Im Servercode kann in der Struktur `svc_req` ermittelt werden, welche Authentifizierung vom Client benutzt wird. Danach kann der Ruf abgelehnt werden, falls das Verfahren nicht ausreicht. Ein Server könnte somit vom Client erzwingen, DES zu benutzen. In der Anwendung findet keinerlei Test statt. Damit obliegt es vollständig dem Client eine Methode auszuwählen.

4.7 Security / Authorization

Die ONC-RPC bietet keine Möglichkeiten zur Datenverschlüsselung oder zur Verwaltung von Zugriffsrechten. Mit DES-Authentication werden nicht die Nutzerdaten

[2]Gemeint ist das zweite Login auf dem Server

verschlüsselt.

4.8 Speichermanagement

Wie bereits gesehen, erfolgt alle Ein- oder Ausgabe über die XDR-Routinen. Die Stubs haben nur die Aufgabe die die richtigen Routinen zu rufen. Die Behandlung des Speichers erfolgt dabei nach einem Schema: Für Ausgabewerte wird prinzipiell Speicher angelegt.

4.8.1 Clientstub

Für die meisten Fälle mag dieses Vorgehen sinnvoll sein, an einer Stelle sollte allerdings in bestehenden Speicher geschrieben werden.

Vor dem Ruf von `fetchbody` wird Speicher angelegt, allein um festzustellen, ob genügend Speicher verfügbar ist. In den so gewonnenen Speicher soll der (Teil des) Body geschrieben werden. Mit der Stubroutine ist dieses Vorgehen nicht möglich. Es wurde eine eigene geschrieben.

```
/* Clientstubroutine fetchbody */
fetchbody_reply *
fetchbody_1(argp, clnt)
        fetchbody_req *argp;
        CLIENT *clnt;
{
        static fetchbody_reply res;

/* Resultate vor dem Ruf auf 0 setzen, d.h. xdr_fetchbody_reply legt Speicher an */
        bzero((char *)&res, sizeof(res));
        if (clnt_call(clnt, FETCHBODY, xdr_fetchbody_req, argp, xdr_fetchbody_req, argp,
                      xdr_fetchbody_reply, &res, TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&res);
}

/* Alternative */
        ...
        static fetchbody_reply reply;
        char*    buffer;

        buffer = (char*)malloc(req->anz);
        /* Test, ob der Speicher angelegt werden konnte */
        ...
        /* er konnte */
        bzero((char*)&reply, sizeof(reply));

        /* Pointer auf Speicher uebergeben, dorthin erfolgt Ausgabe */
        reply.fetchbody_reply_u.body = buffer;
        if(clnt_call(pclt->clnt, FETCHBODY, xdr_fetchbody_req, req,
                     xdr_fetchbody_reply, &reply, timeout) != RPC_SUCCESS)
                return(NULL);
        ...
```

4.8.2 Server

Mit den XDR-Routinen wird der Speicher für die Eingabeparameter angelegt, dieser Speicher wird aber vom Serverstub selbst wieder freigegeben.

Ein Problem entsteht, wenn der Server Speicher neu anlegt, um dort Ausgabedaten einzutragen. Werden diese Ausgabedaten mit `return` zurückgegeben, so hat die Serverprozedur innerhalb dieses Aufrufes keine weitere Möglichkeit, den Speicher wieder freizugeben.

In der Anwendung wird der Verweis auf den angelegten Speicherplatz in einer globalen Variablen abgelegt. Diese wird bei jedem Prozeduraufruf getestet. Das garantiert, daß referenzierter Speicher mit dem nächsten RPC freigegeben wird.

```
...
/* global memory */
char*    memo = NULL;
#define  FREE_MEMO    {if(memo) free(memo); memo = NULL;}
...
reply_status everycall_1(req, svc_req)
    input_t         req;
    struct svc_req* svc_req;
{
    FREE_MEMO;
    ...
}

reply_status fetchbody_1(req, svc_req)
    fetchbody_req*      req;
    struct svc_req* svc_req;
{
    char    *tmp;

    FREE_MEMO;
    ...
    tmp = malloc(SIZE);
    /* fuelle Teil des Body in tmp */
    ...
    memo = tmp;
    ...
}
```

4.8.3 Pointer

In der Anwendung werden nur Pointer auf Strings benutzt. Die Definition `string mailbox<>` in der Schnittstellenbeschreibung wird übersetzt in `char* mailbox`. Die zugehörige XDR-Routine heißt `xdr_string`[3]. Im Gegensatz zu anderen Pointern, die mit `xdr_pointer` bearbeitet werden, unterstützt diese keine NULL-Pointer. Soll *kein* String übertragen werden, so kann nur ein String mit der Länge 0 benutzt werden.

4.9 Fehlerbehandlung

Der zusätzlich auftretende Fehler in RPC ist ein Fehler in der Kommunikation. Sie entstehen, wenn ein Netzwerkfehler auftritt oder ein beteiligter Prozeß ausgefallen

[3]Der Aufruf erfolgt indirekt über `xdr_wrapstring`.

ist. Beide Fehler bewirken einen Zusammenbruch der TCP-Verbindung.

4.9.1 Client

Ein RPC wird im Clientstub mit dem Ruf `clnt_call` abgesandt. Gleichzeitig wird eine Timer gestartet. Tritt ein Fehler auf oder ist der Timer abgelaufen, so gibt die Stubprozedur einen NULL-Pointer zurück. Die Ursache des Fehlers wird in der CLIENT-Struktur gehalten und kann z.B. mit `clnt_perror` zu einem lesbaren Text umgewandelt werden.

In der Anwendung folgen die Prozeduren einem Muster:

```
...
/* zusammenstellen der Uebergabeparameter */
...
reply = proc_1(parameter, client_handle);
if(!reply) {
    clnt_perror(client_handle, "proc");
    return(FEHLER_IN_PROC);
}
/* Auswertung der Rueckgabewerte */
...
```

In der Anwendung werden keine Methoden zum Wiederaufbau der Verbindung vorgesehen. Fehler werden dem Nutzer gemeldet, dieser kann selbst eine neuen Verbindungsaufbau initiieren.

4.9.2 Server

Der unreguläre Abbruch eines Clients kann der Server nicht ermitteln, da er über keine Möglichkeit des Rückrufes verfügt. Da für jeden Client aber ein spezieller Server instanziiert wurde, ist es wünschenswert daß diese bei fehlerhaften Clients nicht sehr lange weiterexistieren.

Dazu wird wie in den anderen Implementierungen im Server mit Start das Signal ALRM angefordert (Standard 20 Minuten). Sämtliche Signale werden auf eine Prozedur `shutdown_if` umgelenkt, in der der Server den Eintrag in der Portmap löscht und `exit` ausführt. Damit ist sichergestellt. daß die Server ihre Einträge in der Portmap bei alle abfangbaren Signalen wieder freigeben. Mit dem Signal SIGPIPE wird genauso verfahren.

4.10 Aufrufsemantik

Die Semantik wird direkt von dem unterliegenden Transportprotokoll bestimmt.

4.10.1 TCP

Treffen die Rückgabewerte ein, so ist sichergestellt, daß die Prozedur *genau einmal* ausgeführt wurde. Nach einem Timeout oder einem Zusammenbruch der TCP-Verbindung wurde die Prozedur *höchstens einmal* ausgeführt. In der Anwendung wird mit TCP gearbeitet.

4.10.2 UDP

Erhält ein RPC Rückgabewerte, so ist sichergestellt, daß die Prozedur *wenigstens einmal* ausgeführt wurde. Im Fehlerfall (nach einem Timeout) ist nicht entscheid-

bar, ob der Ruf oder die Antwort verlorenging. Die Prozedur kann gar nicht oder mehrfach ausgeführt worden sein[Cor].

4.10.3 Asynchroner RPC

Der Client kann die Dauer bestimmen, die er auf die Antwort des Servers warten will. Das geschieht entweder direkt im Ruf `clnt_call` im Clientstub oder außerhalb mit der Funktion `clnt_control`. Wird hier eine Dauer von 0 vereinbart, so kehrt `clnt_call` sofort nach dem Absetzen des Rufes mit einem Fehler (Timeout) zurück und wartet nicht auf die Antwort des Servers.

Dem Server ist es freigestellt, keine Antwort an den Client zurückzusenden. Gibt die entfernte Prozedur auf dem Server den Wert NULL als Resultat an den Serverstub zurück, so werden keine Werte an den Client übertragen. Sinnvoll kann dieses Vorgehen sein, wenn die Prozedur vom Typ *void* ist und der Client nicht auf die Beendigung der Prozedur wartet. Diese Arbeitsweise wird als **asynchroner RPC** bezeichnet.

Mit UDP kann keine Aussage über die Häufigkeit der Ausführung gemacht werden, mit TCP wurde die entfernte Prozedur *höchstens einmal* ausgeführt.

4.10.4 Batching

Batching ist ein Spezialfall des asynchronen RPC. Damit können mehrere RPC zuerst im Client gespeichert werden, um dann zusammen abgeschickt zu werden. Dazu muß die Prozedur `clnt_call` direkt manipuliert werden. Der fünfte Parameter ist ein Verweis auf die XDR-Routine, die zur Konvertierung der Rückgabewerte genutzt werden soll. Wird an dieser Stelle eine NULL vereinbart, so wird der Ruf nur gespeichert, nicht sofort abgesandt. Ein Absenden erfolgt erst, wenn ein RPC benutzt wird, der kein Batching unterstützt, in dem also eine XDR-Routine zur Umwandlung der Rückgabewerte angegeben wird (und sei es nur `xdr_void` als dummy-Routine).

Unterstützt wird diese Methode nur auf verbindungsorientierten Protokollen, wie TCP. Ein Performancegewinn kann eintreten, wenn mehrere kleine RPC-Messages zusammen übertragen werden. Es entfallen zum einen die zusätzlichen Header der unteren Protokolle, zum anderen wird auch nur einmal eine Übertragung angestoßen. TCP benutzt Puffer, um die zu versendenden Daten zu speichern. Laufen diese Puffer über, werden die gespeicherten RPC auch ohne expliziten Anstoß des Clients übertragen. Jeder einzelne Ruf wurde *höchstens einmal* ausgeführt.

Kapitel 5

ROSE

5.1 OSI

Mit **Open System Interconnection** bietet die ISO Standards für die Kommunikation in offenen Systeme. Die notwendigen Protokolle werden in einem Schichtenmodell eingeordnet.

Application	Management der Kommunikation zwischen Anwendungen
Presentation	strukturiert die zu übertragenden Daten
Session	Kontrollmechanismen für den Datenaustausch
Transport	Sichere Datenübertragung
Network	End-zu-End Übertragung von Daten über ein Netzwerk
Datalink	Übertragung von Daten über eine einzige Verbindung
Physical	Konvertierung von Bits in physikalische Größen des Mediums

Abbildung 5.1: Schichtenmodell nach OSI

Die Funktionalität einer Schicht wird von mehreren Elementen dieser Schicht zur Verfügung gestellt, zusammen bilden sie den **Provider**. Der Zugriff auf Dienste eines Providers erfolgt über einen **Service Access Point (SAP)**. Die Adresse eines SAP besteht aus zwei Teilen,

- einer/mehreren Adresse(n) der unterliegenden Schicht und
- einem **Selektor** der ein Element in dieser Schicht angibt.

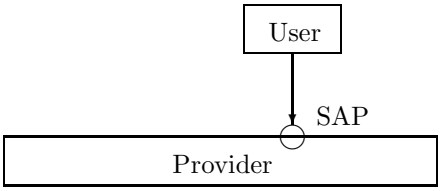


Abbildung 5.2: Dienstzugriff über SAP

5.1.1 Anwendungsschicht

In der Schicht 7 sind die Anwendungen enthalten. In der OSI-Terminologie wird von einem **Application Process (AP)** gesprochen. Um die Kommunikation zu

beschreiben, wird ein **Application Entity (AE)** eingeführt. Ein AP kann mehrere AE besitzen, die nach einem **Application Protocol** Daten austauschen. Zwei AE, die miteinander kommunizieren, müssen aus den gleichen ASE aufgebaut sein. Mit **Application Context Name** wird ein einzelnes Application Protocol bezeichnet. Ein AE wiederum enthält **Application Service Elements (ASE)**, siehe Abbildung 5.3.

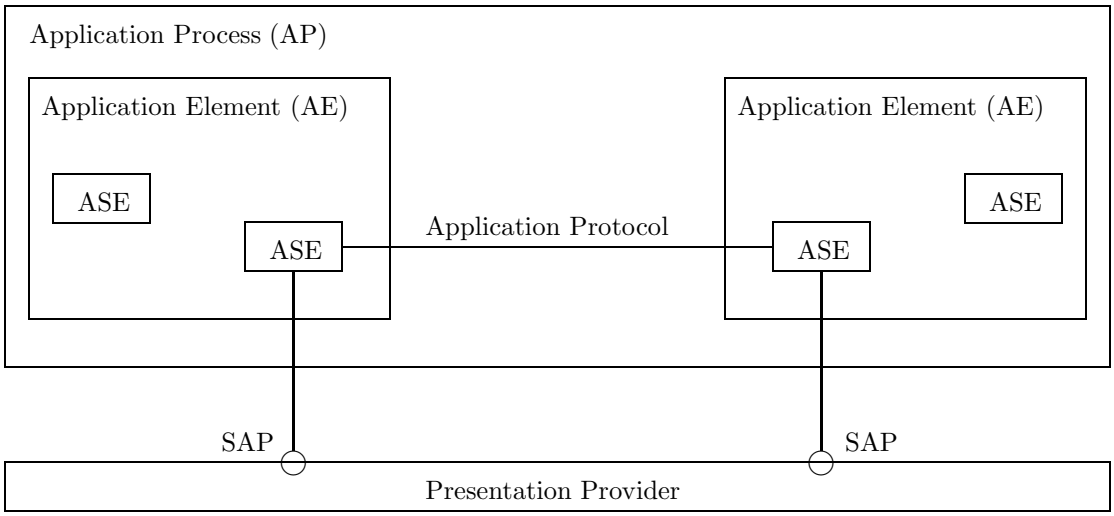


Abbildung 5.3: Elemente der Schicht 7

Die Verbindung zwischen zwei AE wird **Association** genannt. Für eine Association ist eine Verbindung auf der Schicht 6 notwendig, ebenso wie diese eine Verbindung auf der Schicht 5 zur Bedingung hat ...

Das AE, das den Verbindungsaufbau anfordert, ist der **Initiator**, das andere der **Responder**.

OSI dient der Beschreibung von verteilten Anwendungen, eine Association ist also ein elementarer Bestandteil jeder OSI-Anwendung. Aus diesem Grund wurde ein spezielles ASE standardisiert, das **Association Control Service Element (ACSE)**, das Associations eines AE verwaltet.

Ein weiteres standardisiertes ASE ist das **Remote Operation Service Element (ROSE)**. Über eine bestehende Association ist es mit ihm möglich, von einem AE aus, in einem anderen Operationen aufzurufen. Dieser Vorgang wird von einem User-Element gesteuert, siehe Abbildung 5.4.

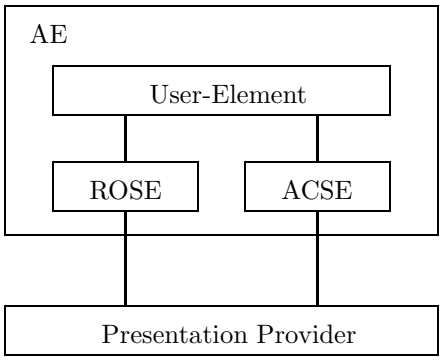


Abbildung 5.4: ACSE und ROSE

Im Zusammenhang mit ROSE wird von entfernten *Operationen* gesprochen, während bei RPC von entfernten *Prozeduren* die Rede ist. Beide Worte sind im folgenden als synonym anzusehen.

5.2 Das ROSE-Protokoll

Um eine entfernte Prozedur aufzurufen, wird eine **Invocation** an das entfernte AE gesandt, dieses antwortet entweder mit einem Resultat, einer Fehlermeldung oder einer Zurückweisung der Invocation.

Eine Invocation enthält

- die Nummer der zu rufenden Operation,
- das Argument,
- einen Identifikator (**Invocation ID**) und
- den Identifikator der anzeigt, ob diese Invocation zur Ausführung einer anderen Invocation genutzt wird (**linked Invocation ID**).

Das Resultat enthält

- die ID der Invocation auf die geantwortet wird und
- die Rückgabeparameter.

Eine Fehlermeldung enthält

- die ID der Invocation auf die geantwortet wird,
- einen Fehlercode und
- weiter Parameter.

Eine Zurückweisung (Rejection) enthält

- die ID der Invocation die zurückgewiesen wird und
- den Grund der Zurückweisung.

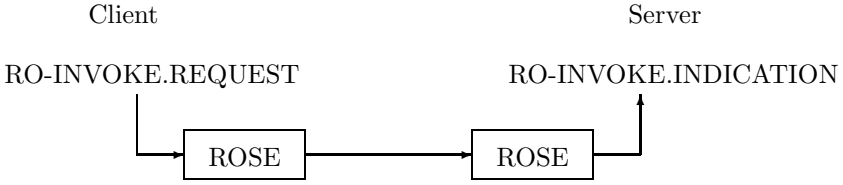
Die Übergabe dieser Werte erfolgt asynchron, es werden keine Quittungen ausgetauscht. Der Ablauf der Übertragung ist stets gleich. Der Invoker fordert ROSE auf, eine der oberen Meldungen zu übertragen und benutzt dazu einen REQUEST. Geht eine Meldung ein, generiert ROSE eine INDICATION. Dieser Mechanismus wird in beiden Richtungen benutzt.

OSI definiert einige Primitives, zum Versenden der Meldungen. Diese sollen im weiteren vorgestellt werden.

5.2.1 ROSE Primitives

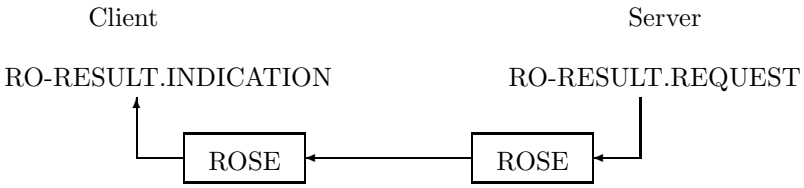
Das AE, das eine Invocation versendet, ist der **Invoker**. Das ausführende AE ist der **Performer**. Im Client-Server-Modell entspricht der Invoker dem Client und der Performer dem Server.

Invocation



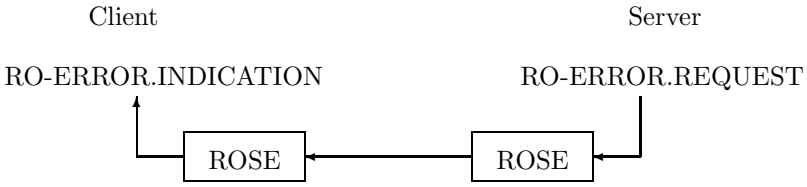
Der Client übergibt dem lokalen ROSE die Invocation. Sie werden zum entfernten ROSE übertragen, dieses erzeugt ein RO-INVOKE.INDICATION. Der Server kann die Eingabeparameter übernehmen und die gewünschte Prozedur ausführen. Bei Erfolg gibt er ein Resultat zurück.

Resultate



Dazu ruft der Server das lokale ROSE mit RO-RESULT.REQUEST. Auf der Clientseite erzeugt ROSE ein RO-RESULT.INDICATION, dem der Client die Rückgabewerte entnehmen kann.

Fehler



Schlägt die Operation auf der Serverseite fehl, wird eine Fehlermeldung übertragen.

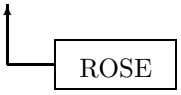
User-Rejection

Ein AE kann eine Invocation, ein Resultat oder eine Fehlermeldung zurückweisen. Die geschieht, wenn das AE die Meldung aus irgendeinem Grund nicht auswerten kann, z.B. wenn eine unbekannte Operation in einer Invocation verlangt wird, wenn die Resultate für den Client nicht deutbar sind oder wenn eine unbekannte Fehlermeldung eintraf.

Client und Server können eine Rejection generieren.

Provider-Rejection

RO-REJECT-P.INDICATION



ROSE generiert ein RO-PREJECT.INDICATION wenn ein Verstoß gegen das ROSE-Protokoll eintrat oder wenn in den unterliegenden Schichten ein Fehler auftrat.

5.2.2 Abbildung der Kommandos auf ROSE-Primitives

Mit diesem Primitives soll nun das Zugangsprotokoll aus Abschnitt 2 implementiert werden. Jedem Kommando wird einen entfernte Operation zugeordnet. Ein Kommandoaufruf entspricht dem Versenden einer Invocation.

ROSE arbeitet mit einem asynchronen Protokoll, das Zugangsprotokoll wurde synchron definiert. Das synchrone Protokoll läßt sich aber mit dem asynchronen nachstellen. Dazu wartet der Client nach dem Absenden einer Invocation auf eine Antwort. Diese Antwort kann das Resultat, eine Fehlermeldung oder eine Zurückweisung sein. Erst danach werden weiter Invocations versandt. Damit kann der Client die Antwort eindeutig der vorher abgesandten Invocation zuordnen, die Nutzung der Invocation ID wird unnötig.

Der Server wartet nach der Initialisierung auf das Eintreffen einer Invocation. Die zu rufende Operation ist in der eintreffenden Invocation verschlüsselt. Sollte eine unbekannte Operation gerufen werden, so muß der Ruf zurückgewiesen werden.

Das Zugangsprotokoll erzwingt eine Reihenfolge der Kommandos. Wird diese nicht eingehalten (wird z.B. ein STAT vor einem SELECT) gerufen, so wird ein Protokollfehler gemeldet (`bad`).

Die nun gerufene Operation kennt die Struktur der Eingabeparameter und kann diese lesen. Tritt hierbei ein Fehler auf, so sollte die Invocation zurückgewiesen werden. Danach kann die Operation ausgeführt werden, dabei kann es zu Fehlern kommen. Diese Fehler betreffen die Ausführung der Prozedur, werden also mit dem Indikator `no` bezeichnet.

Ohne auftretende Fehler kann das Ergebnis übertragen werden. Ein Ergebnis erhält der Client mit einem RO-RESULT.INDICATION. Siehe Abbildung 5.5.

5.3 Schnittstellenbeschreibung

Zur Beschreibung der Application Entity wird die **Abstract Syntax Notation No.1 (ASN.1)** benutzt. Eine Einführung dazu ist z.B. in [Ros1] [Hüb2] zu finden. Die folgende Darstellung basiert wesentlich auf [Ros1].

Für ROSE wurde die Sprache um einige *Macros* erweitert, der RO-Notation.

Mit der Beschreibung der AE sollen zwei Dinge erreicht werden,

- das AE soll im Prozeß des Bindens (Abschnitt 5.6) gefunden werden und
- die (entfernten) Operationen und ihre Datentypen sollen beschrieben werden.

Dazu dienen im wesentlichen zwei *Macros* (`APPLICATION-CONTEXT, OPERATION`). Die Deklarationen werden in einem ASN.1-Modul zusammengefaßt, das Modul der Anwendung befindet sich im Anhang, Abschnitt C.

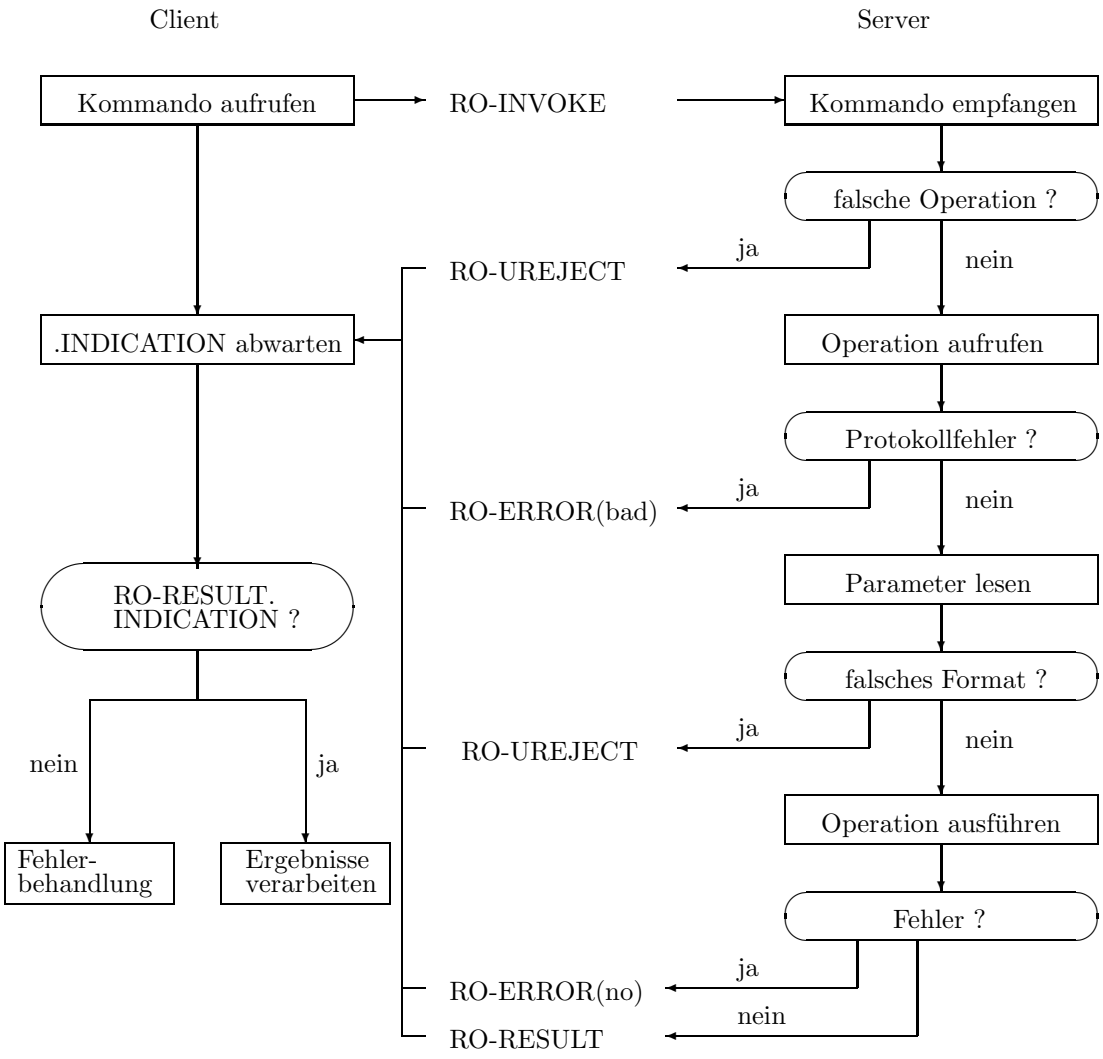


Abbildung 5.5: Kommandoaufruf mit ROSE

5.3.1 Application Context

Beispiel:

```
mailContext APPLICATION-CONTEXT
    APPLICATION SERVICE ELEMENT    {aCSE}
    BIND                           NULL
    UNBIND                         NULL
    REMOTE OPERATIONS              {rOSE}
    INITIATOR CONSUMER OF          {mailClient}
    ABSTRACT SYNTAXES              {aCSE-abstract-syntax, mail-abstract-syntax}
    ::= {1 17 2 1 2}
```

Zunächst wird für den Kontext der Applikation ein **Object Identifier** vereinbart (1 17 2 1 2). Ein Object Identifier verweist auf einen Eintrag in der MIB (Management Information Base), auf die nicht weiter eingegangen werden soll. Mit einem Object Identifier wird ein Element auch über das ASN.1-Modul hinaus eindeutig bezeichnet.

- Mit `APPLICATION SERVICE ELEMENT` werden weitere ASE des AE angegeben, die nicht für entfernte Operationen benutzt werden. ACSE ist Bestandteil jeder Anwendung.
- Mit `BIND` und `UNBIND` werden Datentypen deklariert, die beim Verbindungsaufbau bzw. -abbau übertragen werden. In der Anwendung werden keine benutzt.
- `REMOTE OPERATIONS` gibt das ASE an, das für den Aufruf der entfernten Operationen benutzt wird. Das ist an dieser Stelle natürlich ROSE. Durch dieses Konstrukt wird die Möglichkeit offen gehalten, ROSE durch andere ASE zu ersetzen.
- Die `ABSTRACT SYNTAXES` definiert die Syntax der übertragenen Daten, siehe auch Abschnitt 5.4. In diesem AE wird die abstrakte Syntax des ACSE benutzt und die Syntax dieser Anwendung. Sie wird mit einem Object Identifier bezeichnet.

  ```
  mail-abstract-syntax OBJECT IDENTIFIER ::=
      1 17 2 1 1
  ```

- Damit sind die Bestandteile des AE beschrieben, nun sollen die Verbindungen zwischen den ASE deklariert werden. Mit `INITIATOR CONSUMER OF` werden die ASE genannt, die ein Initiator benutzt.
  - ein analoges Konstrukt ist `RESPONDER CONSUMER OF`, es werden die ASE aufgezählt, die der Responder benutzt
  - mit `OPERATIONS OF` können für beide nutzbare ASE angegeben werden.

  In der Anwendung ist das Clientprogramm ein User-Element, also ein ROSE-basiertes ASE (`mailClient`). Nach dem entworfenen Zugangsprotokoll baut der Client die Verbindung auf (Initiator) und ruft entfernte Operationen auf dem Server. Der Server hat keine Möglichkeit, im Client Operationen aufzurufen. Damit ist nur der `INITIATOR` Nutzer eines (`CONSUMER OF`) ASE, das entfernte Operationen aufrufen kann. Die beiden anderen Konstrukte werden nicht benötigt.

Das User-Element wird als ASE deklariert.

```
mailClient APPLICATION-SERVICE-ELEMENT
    CONSUMER INVOKES {login logout select check expunge stat fetchbody store search}
    ::={1 17 4 1 1}
```

Dem ASE wird ein Object Identifier zugeordnet. Mit `CONSUMER INVOKES` werden die Operationen aufgezählt, die das ASE als entfernte Operationen benutzen kann. Das sind genau die Kommandos des Zugangsprotokolls.

Der Aufbau des AE ist nun beschrieben. Es enthält wie in Abbildung 5.4 drei ASE. Nun sind die Operationen zu deklarieren, die das User-Element nutzt.

5.3.2 Operationen

Zur Deklaration einer Operation wird das Macro `OPERATION` genutzt.

```
login OPERATION
    ARGUMENT    LoginReq
    RESULT      NULL
    ERRORS      { no, bad }
    ::=         0
```

Zunächst wird `login` ein Integerwert zugewiesen, der innerhalb des ASN.1-Moduls eindeutig ist. Mit `ARGUMENT` werden die Eingabeparameter der Operation definiert. Es gibt nur *einen* Parameter, sollen mehrere übergeben werden, so ist ein neuer Typ zu vereinbaren.

```
LoginReq ::=
    SEQUENCE {
        user      IA5STRING,
        passwd    IA5STRING
    }
```

Mit `login` wird der Nutzername und das Passwort als ein ASCII-String übertragen.

Mit `RESULT` wird der Typ des Rückgabeparameters definiert. Bei `login` wird keiner benötigt[1]. Zu beachten ist, daß die Operation bei Erfolg zwar keinen Parameter zurückgibt, aber allein durch die Rückgabe eines (leeren) Resultats den Erfolg der Operation angibt.

Mit `ERRORS` werden zwei Fehlertypen definiert (`no, bad`). Fehler sind vom Datentyp `ERROR`.

```
no      ERROR
    PARAMETER    IA5String
    ::=          0
```

Ein Fehler vom Typ `no` besitzt als Parameter einen String. In der Anwendung wird hier ein lesbarer Fehlertext eingetragen. Genauso ist `bad` definiert.

Durch die Trennung von Resultat und Fehlern muß der Indikator `ok` nicht explizit übertragen werden. Wird ein Resultat empfangen, so war der Ruf erfolgreich.

Ein letzter Parameter (der nicht benutzt wurde) ist `LINKED`.

```
operation_with_linked OPERATION
    ARGUMENT IA5STRING
    LINKED { read, write}
    ::= 42

read OPERATION
    RESULT IA5STRING
    ERROR  {no}
    ::= 43

write OPERATION
    ARGUMENT IA5STRING
    ::= 44
```

Werden während der Ausführung der ersten Operation noch weitere Werte benötigt, so wird vom Performer *read* gerufen. Der Invoker sendet einen String oder einen Fehler (`no`). Soll andererseits bereits während der Ausführung der Operation eine Ausgabe erfolgen, ruft der Performer `write` und überträgt einen String zum Invoker. Dieses Vorgehen erinnert an die rückwirkenden Aufrufe des Types Pipe in DCE.

Die anderen Operationen sind ähnlich deklariert. Im weiteren soll die Definition einiger Datentypen erläutert werden.

[1] Ein Superserver war mit dem benutzten ROSE nicht notwendig, deshalb müssen auch keine Adreßinformationen zurückgegeben werden, wie bei den anderen Versionen.

5.3.3 Flags

Das Zugangsprotokoll unterstützt Flags, die in jeder Nachricht gespeichert werden. Sie werden mit STAT vom Server bezogen und mit STORE manipuliert. Auch in den anderen Schnittstellenbeschreibungen belegte jedes Flag ein Bit in einem Integerwert. In ASN.1 kann dies direkt ausgedrückt werden.

```
Flags ::=
    BIT STRING ::= {
        seen (0), answered (1), deleted (2), flagged (3) recent (4)
```

Jedem Flag wird damit ein Bit zugeordnet.

5.3.4 Nachrichtennummern

Mit dem Öffnen der Mailbox weist der Server jeder Nachricht eine Nummer zu.

```
Msgnr ::= INTEGER(0<..MAX)
```

Während in Programmiersprachen sehr oft mehrere Integertypen benutzt werden, um die unterschiedliche Längen oder Interpretationen[2] zu definieren, gibt es in ASN.1 nur einen Typ INTEGER, der aber mit Untertypen in seinem Wertebereich eingeschränkt werden kann.

Für die Nachrichtennummern sind nur positive Zahlen zulässig.

In den Kommandos werden die Nachrichten mit einer `Sequence` adressiert, das ist eine Anzahl von Intervallen von Nachrichtennummern. Es ist zwischen den Schlüsselwörtern `SEQUENCE`, es deklariert eine Struktur, `SEQUENCE OF`, es deklariert ein Array und der `Sequence` aus der Anwendung zu unterschieden.

```
Interval ::=
    SEQUENCE {
        from Msgnr,
        to   Msgnr
    }

Sequence ::=
    SEQUENCE OF
        Interval
```

`Interval` ist eine Struktur und enthält zwei Elemente vom Typ `Msgnr`. Die `Sequence` ist ein Array von Intervallen.

5.3.5 Indikatoren

Da bereits bei der Definition einer Operation die Fehlertypen definiert wurden, müssen keine Indikatoren zusätzlich vereinbart werden.

5.4 Datenkonvertierung

Mit der Definition der Datenstrukturen im ASN.1-Modul entstand eine **Abstract Syntax**. In ihr werden keinerlei Aussagen über die interne Darstellung der Daten gemacht oder in welcher Darstellung die Daten über das Netzwerk (**Concrete Syntax**) übertragen werden.

[2] mit oder ohne Vorzeichen

In OSI wird der Weg über ein einheitliches Datenformat gegangen. Es werden in einer **Transport Syntax** Regeln definiert, wie die abstrakte Darstellung konkret in einen Bytestrom umgewandelt wird. Derzeit unterstützt OSI nur eine Transportsyntax, die **Basic Encoding Rules (BER)**.

Nähere Beschreibungen befinden sich in [Ros1] [Hüb2].

Mit BER wird jedes Datum mit einem *tag* eingeleitet, womit dem Empfänger dessen Struktur bekanntgegeben wird. Da Datentypen teilweise nur innerhalb eines ASN.1-Moduls bekannt sind, also innerhalb einer abstrakten Syntax, muß beiden Seiten diese Syntax bekannt sein. Aus diesem Grund wurde mit der Deklaration des AE die benutzte abstrakte Syntax angegeben.

Das Umwandeln der Daten ist Aufgabe der Schicht 6 im Referenzmodell. Ein Paar (abstrakte Syntax, Transportsyntax) wird hier als **Presentation Context** definiert, dieser wird durch einen **Presentation Context Identifier (PCI)**, einen Integerwert, repräsentiert.

5.5 Werkzeuge

Alle notwendigen Elemente sind definiert, nun sollten Werkzeuge vorgestellt werden, die die Beschreibungen für eine Programmiersprachen aufarbeiten. ROSE ist im Gegensatz zu den anderen RPC-Konzepten „nur“ ein Standard und keine Implementierung. Konkrete Aussagen über die Umsetzung werden in OSI natürlich offengelassen.

Als Implementierung einiger OSI-Konzepte stand **ISODE** zur Verfügung. Basierend auf UNIX und dem Transportprotokoll TCP wurden einige OSI-Standards implementiert. Auf dieser Plattform erfolgte auch die Referenzimplementierung.

5.5.1 ISODE

ISODE bietet eine API für ACSE und für ROSE, siehe [ISODE10]. Die Bibliotheken bieten C-Funktionen an, die Werkzeuge erzeugen C-Quellen.

Das ASN.1-Modul wird zunächst von **rosy (Remote Operation Stub Generator (yacc-based))** bearbeitet. Als Ergebnis entstehen vier Files. Eines enthält die Stubs, ein anderes die ASN.1-Deklarationen der im ursprünglichen Modul vereinbarten Datenstrukturen, alle RO-Notationen wurden entfernt. Die letzten beiden Files sind hier nicht von Interesse, siehe dazu [ISODE18].

Genau genommen werden zwei komplette Clientstubs angeboten. Der eine bietet einen asynchronen, der andere einen synchronen Zugang zu ROSE. In der Anwendung wurde der synchrone Zugang gewählt, da auch das Zugriffsprotokoll synchron definiert wurde. ROSE arbeitet auch dabei asynchron. Dem Programmierer wird lediglich das Warten auf die (asynchron) eintreffende Antwort verborgen.

Die Datenstrukturen werden **pepsy** übergeben. Als Ausgabe entstehen zwei Files. Eines enthält die C-Datentypen, die den ASN.1 - Deklarationen entsprechen, das andere dient der Konvertierung der Daten in die `Concrete Syntax`, siehe Abbildung 5.6.

5.6 Binding

Das ACSE wird zum Aufbau einer Association benutzt. Dazu werden mindestens zwei Parameter benötigt, vier weitere sind optional.

- Der **Application Context Name** benennt das zu nutzende Protokoll und damit implizit das ASE des zu rufenden AE.

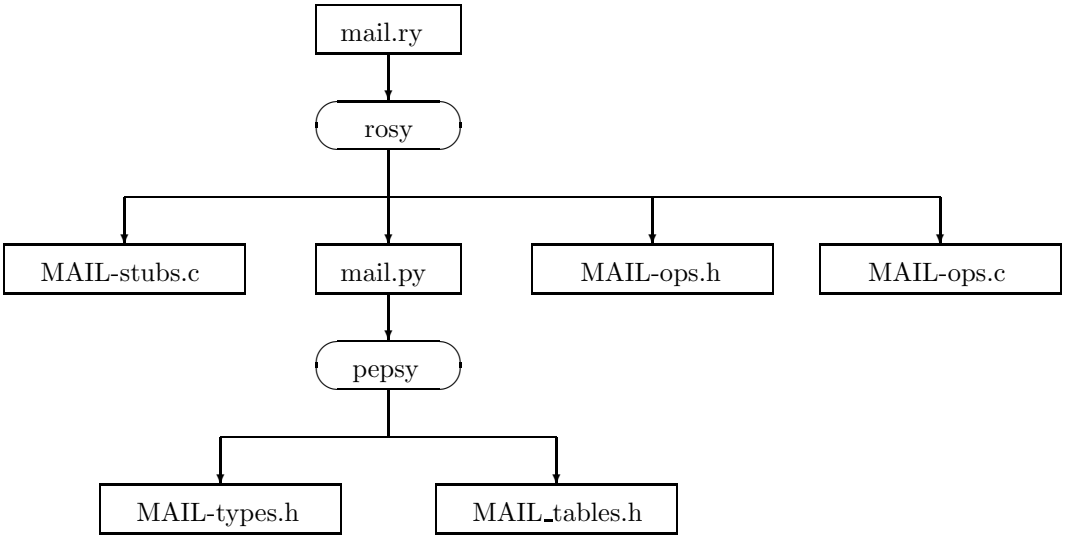


Abbildung 5.6: Die Generatoren rosy und pepsy

- Die **Presentation Context Definition List** enthält den PCI für jedes ASE im AE, d.h. für jedes ASE wird beschrieben, wie die abstrakte Syntax in die konkrete umgewandelt wird. Die folgenden Parameter sind optional.
- Die AE-Informationen des rufenden AE beschreiben den rufenden AP.
- Die AE-Informationen des gerufenden AE beschreiben den gerufenen AP.
- Es kann eine Auswahl stattfinden, welcher Standard aus OSI für ACSE benutzt wird.
- Daten können bereits mit dem Aufbau der Association übertragen werden.

Die AE-Informationen bestehen aus vier Einträgen.

- Mit dem **AP-Title** wird der AP benannt.
- Der **AE-Qualifier** benennt das AE innerhalb des AP.
- Der **AP-Invocation-Identifier** wird mit jedem Start des AP neu gesetzt.
- **AE-Invocation-Identifier** wird mit jedem Start des AE neu gesetzt.

AP-Title und AE-Qualifier bilden den **AE-Title**. Enthält der AP nur ein AE, so kann der AE-Qualifier entfallen.

Das rufende ACSE, der Initiator, kennt die Informationen auf der Anwendungsschicht (AE-Informationen, Application Context Name) aus dem ASN.1-Modul. Notwendig für die Presentation Context Definition List ist die Presentation Adress des Responders.

OSI bietet mit seinem Directory eine Möglichkeit diese zu erlangen. Bekannter ist das Directory unter dem fast gleichlautende Standard X.500 der CCITT.

Das Directory verwaltet Informationen über Objekte. Welche Informationen gehalten werden sollen, ist nicht definiert, dadurch wird das Konzept sehr flexibel. Derzeit dient das Directory im wesentliche zwei Diensten,

- der Verwaltung von E-Mailadressen über ein *white pages* Mechanismus und

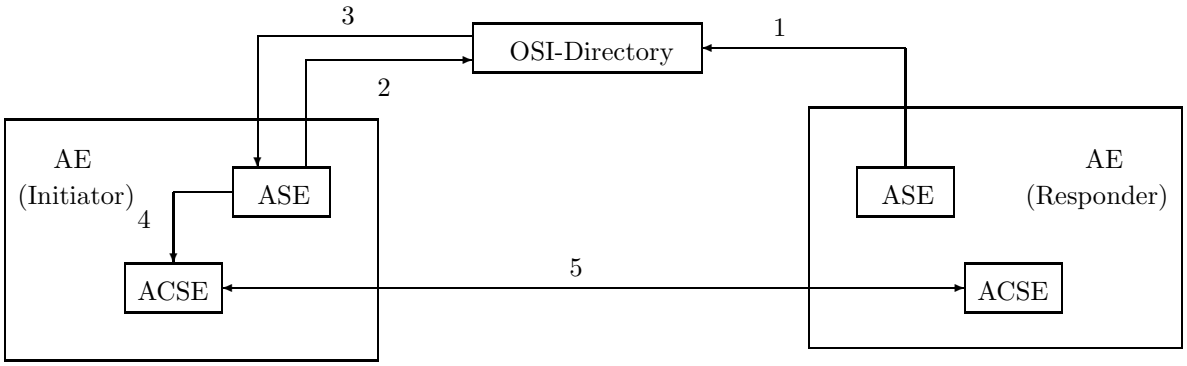
- der Verwaltung von Presentation Adresses von OSI-Anwendungen.

Das Directory besitzt im wesentlichen eine Baumstruktur[3], die Blätter enthalten die Objektinformationen. Ein solches Blatt kann über einen **Distinguished Name (DN)** adressiert werden. Dieses Vorgehen ähnelt dem Adressieren eines Files in einem hierarchischen Filesystem über den vollständigen Pfadnamen.

Die Idee besteht darin, den AE-Title als DN zu konstruieren und beim Start des AP die Presentation Adress unter diesem Namen einzutragen. Ein solcher Name könnte die Form haben:

```
countryName           =    "Germany"
organiationName       =    "TU-Chemnitz-Zwickau"
organizationUnitName  =    "Informatik"
commonName            =    "jupiter"
commonName            =    "mail"
```

Vor dem Aufbau der Association müßte ein ASE[4] die Presentation Adress über den DN ermitteln. Nun kann das ACSE eine Verbindung aufbauen, siehe Abbildung 5.7.



1. Der Responder trägt die Presentation Adress in das Directory ein.
2. Der Initiator übergibt den Distinguished Name des Responder und
3. erhält dessen Presentation Adress.
4. Das ACSE wird mit der Adresse des Responders aufgerufen und
5. stellt eine Association her.

Abbildung 5.7: Binden mit dem OSI-Directory

Das Directory bietet mit den DN die Möglichkeit, AE unter einem weltweit einheitlichen Namen zu erreichen.

5.6.1 ISODE

ISODE bietet einen andere Möglichkeit die notwendigen Informationen zugänglich zu machen.

Die abstrakte Syntax und der Application Context Name werden mit ihrem Object Identifier in das File `isoobjects` eingetragen, das sich im Verzeichnis `etc` unter der ISODE-Root befindet.

[3] mit *alias* Einträgen können Querverweise entstehen
[4] in [Ros1] wird ein *Directory Service Element* eingeführt.

```
"mail pci" 1.17.2.1.1
"mail"     1.17.2.1.2
```

Die Object Identifier wurden am Anfang des ASN.1 Moduls vereinbart.

Die Application Entity Informationen des zu rufenden Entity werden mit der zugehörigen Adresse in das File `isoentities` im gleichen Verzeichnis geführt.

```
mail    1.17.4.1.1    #1024
```

In `isoservices` werden die eigentliche Serverprogramme mit ihren Aufrufparametern eingetragen.

```
"tsap/mail" #1024 ros.maild -l /tmp/maillog
```

Server auf der Basis von ROSE werden per Konvention mit `ros.` eingeleitet. Wie zu sehen wird nur der Server bekanntgegeben, der Superserver kann in ISODE entfallen. Dessen Aufgabe übernimmt ein Dämon `tsapd`. Dessen Arbeitsweise ist dem Internetsuperserver `inetd` sehr ähnlich. Der Aufbau einer Association geschieht wie folgt.

Der Initiator kann über die Application Entity Informationen (`aei`) und dem zu nutzenden Application Context die zugehörige Presentation Adress (`pa`) beziehen.

```
aei = _str2aei (host, MAILSERVICE, MAILCONTEXT, NULL, NULLCP, NULLCP);
pa = aei2addr(aei);
```

Diese Informationen sind in `isoentities` des Serverhosts enthalten.

Die Object Identifier des Application Context (`ctx`) und des PCI (`pci`) erlangt er wie folgt:

```
ctx = ode2oid(MAILCONTEXT);
pci = ode2oid(MAILPCI);
```

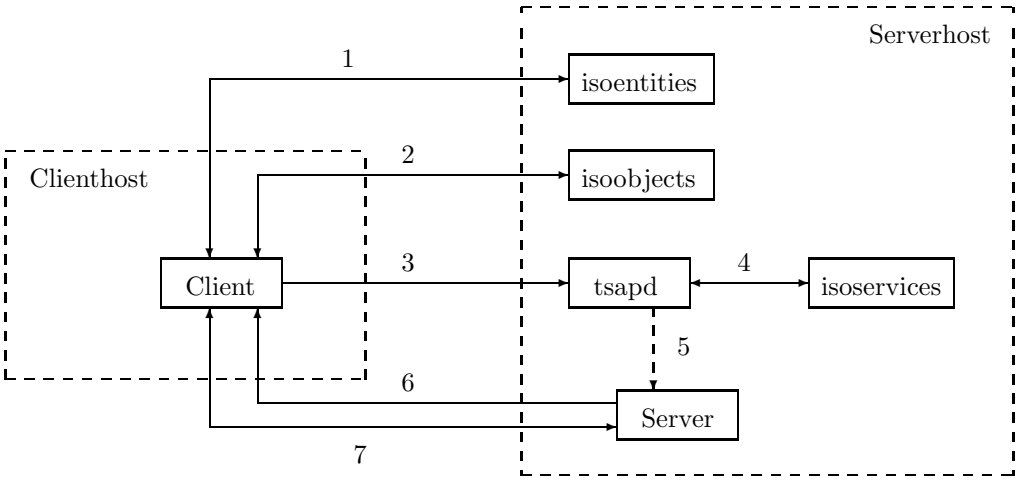
Diese Informationen sind in `isoobjects` auf dem Serverhost enthalten.

Nun kann eine Association mit `AcAssocRequest` angefordert werden. Der Request geht zunächst an `tsapd` auf dem Serverhost. Dieser ermittelt über `isoservices` das benötigte Serverprogramm und startet es mit den Rechten des Eigentümers des Files. Das geschieht mit jedem Ruf neu, so daß jeder Initiator einen eigenen Responder erhält. Der gestartete Responder „weiß“, daß er durch einen Association-Request gestartet wurde und akzeptiert entweder den Request oder weist ihn zurück[5] In der Anwendung wird eine Association generell akzeptiert (`AcAssocResponse`), die Authentifizierung des Nutzers findet über ROSE mit dem Login statt.

Damit ist eine Association aufgebaut, in der Anwendung fungiert der Initiator als Invoker und ruft im Responder Operationen auf, der damit als Performer arbeitet.

Die Rolle des Superserver übernimmt `tsapd`. Das Serverprogramm muß allerdings *root* gehören, um auch mit diesen Rechten gestartet zu werden. Nach einem erfolgreichen Login wechselt der Server sofort seine UID in die des Nutzers und arbeitet danach nur mit dessen Rechten weiter. Um aber die UID eines willkürlichen Nutzers annehmen zu können, muß der Prozeß am Anfang über die notwendigen Rechte verfügen (*root*).

[5]Mit dem Request übertragene Werte werden als zusätzliche Parameter dem Programm übergeben, auf sie muß der Entwickler aber nicht direkt zugreifen.



1. Der Client ermittelt die Presentation Adress über die Application Entity Information und den Application Context Name.
2. Der Client ermittelt den Object Identifier von PCI und Application Context.
3. Der Client sendet einen Request für eine neue Association an `tsapd`.
4. `tsapd` ermittelt das Serverprogramm
5. und ruft es mit den Parametern auf.
6. Der Server akzeptiert die Association.
7. Die weitere Kommunikation findet über diese Association statt.

Abbildung 5.8: Aufbau einer Association mit tsapd

5.7 Authentication / Security

Mit ROSE werden keine Möglichkeiten zur Datenverschlüsselung oder zur Authentifizierung von Nutzern oder Servern vorgeschlagen. Es ist der jeweiligen Implementierung freigestellt, entsprechende Dienste anzubieten. ISODE unterstützt keine solchen Dienste.

5.8 Speichermanagement in ISODE

5.8.1 Client

Der Clientstub legt generell für die Ausgabeparameter Speicher an, den der Client bei Bedarf freigeben kann. Es besteht keine Möglichkeit, Resultate in bereits angelegten Speicher eintragen zu lassen.

5.8.2 Server

Der Speicher der Eingabeparameter wird vom Serverstub angelegt und freigegeben. Speicher für Ausgabewerte muß mit den üblichen Systemrufen angefordert werden.

Die Verwaltung wird allerdings vereinfacht, da die Werte nicht mit `return` zurückgegeben werden.

```
/* Serverprozedur */
{
...
results = malloc(LEN);
/* trage Resultate ein */
...
/* RO-RESULT.REQUEST */
RyDsResult(assoc_id, invokation_id, results, ROS_NOPRIO, error_indication);
free(results);
...
return;
}
```

Die Serverprozedur verliert nach dem Absender der Resultate nicht die Steuerung. Das Problem, das damit bei ONC-RPC entstand, existiert hier nicht, siehe Abschnitt 4.8.2.

5.9 Fehlerbehandlung in ISODE

5.9.1 Server

Durch die Benutzung der Stubs muß die Behandlung der Fehler bei der eine Rejection gesendet werden soll (Abbildung 5.5) nicht vom Entwickler erfolgen. Eine falsche Operation wird aufgerufen, wenn der Client über ein falsches ASN.1-Modul verfügt. Dies wird bereits vom Serverstub bemerkt.

Eine Rejection sollte ebenfalls erfolgen, wenn die Parameter eine falsche Struktur besitzen. Das wird meist schon beim Compilieren bemerkt, spätestens aber durch einen Fehler bei der Datenkonvertierung vor oder nach der Übertragung. Wird auch hier kein Fehler entdeckt, sind die Daten syntaktisch korrekt, die Semantik ist falsch. Diese Fehler werden aber mit einer Fehlermeldung beantwortet, nicht mit einer Rejection.

Fehlermeldungen überträgt der Server wie ein Resultat, es wird die Funktion `RyDsError` benutzt. Bei dieser Funktion kann es wie beim Senden eines Resultates nur zu einer Rejection des Providers kommen, da der Client den Eingang der Daten nicht quittiert oder ablehnen, also auch keine Rejection erzeugen kann[6]. In der Anwendung beendet in einem solchen Fall der Server die Arbeit.

5.9.2 Client

Eine Stubroutine kehrt mit einem von drei möglichen Werten zurück.

NOTOK Die Invocation wurde zurückgewiesen (RO-REJECT-U.INDICATION oder RO-REJECT-P.INDICATION). Entweder wurde ein falsches ASN.1-Modul benutzt oder es kam zu einem Fehler in den unterliegenden Protokollen. Ein falsches Datenformat kann damit nicht entdeckt werden, entweder werden die Daten korrekt konvertiert oder es tritt dabei ein Fehler auf. Dieser führt aber zu einer Ausnahmebehandlung des Betriebsystems, nicht zu einer Rejection.

[6]Rufe in ROSE sind generell unbestätigt

OK Das Resultat oder eine Fehlermeldung wurde erfolgreich übertragen (RO-RESULT.INDICATION oder RO-ERROR.INDICATION). Die Fehler werden in der Anwendung immer mit einer lesbaren Fehlermeldung übermittelt, diese wird angezeigt.

DONE Ein anderer Fehler trat auf.

5.10 Aufrufsemantik

Die Semantik ist von der Zuverlässigkeit der unterliegenden Protokolle abhängig. ISODE arbeitet mit dem Transportprotokoll TCP. Damit ist sichergestellt, daß bei Erhalt einer Antwort die Operation *genau einmal*, ansonsten *höchstens einmal* ausgeführt wurde.

Kapitel 6

Zusammenfassung

Ziel der Arbeit war es, anhand eines entfernten Mailzugriffs, die Möglichkeiten von drei RPC-Implementierungen zu untersuchen. Dazu wurde zunächst aus zwei existierenden Zugangsprotokollen ein weiteres, für die Implementierung mit RPC geeignetes, abgeleitet. Danach wurden die benötigten Komponenten der Systeme vorgestellt und es wurde gezeigt wie sie für die Implementation genutzt wurden. Das sollte dazu dienen, die auftretenden Probleme aufzuzeigen und Lösungen vorzustellen. Gleichzeitig sollten damit aber auch die Quellen der Anwendung verständlich werden. Die entstandene Implementierung kann als Beispiel für andere Arbeiten dienen.

Nimmt man RPC wörtlich, *entfernter Prozeduraufruf*, so umfaßt dies lediglich das Marshalling und Unmarshalling von Übergabeparametern und das Übertragen dieser Daten. Aus dieser Sicht ist ein Vergleich der Performance sinnvoll. Für ONC-RPC und DCE-RPC bietet sich dies an, ROSE ist allerdings ein Standard und bezeichnet keine Implementierung. Zeitgleich mit dieser Arbeit wurden für die ersten beiden RPC solche Untersuchungen mit eigens dazu entworfenen Programmen durchgeführt ([Bric]) und deshalb nicht wiederholt.

Es war aber auch zu sehen, daß RPC nicht losgelöst von seiner Umgebung gesehen werden kann. ROSE ist Bestandteil von OSI und kann davon nicht getrennt werden. Die RPC-API in DCE bietet einen Zugang zu den anderen Diensten (CDS oder GDA) oder nutzt sie nahezu verborgen vor dem Programmierer (Security Service). Obwohl mit ONC-RPC zunächst nur der Austausch von zwei Strukturen definiert wurde (Request und Response), muß doch der Portmapper und der Keyserver, ohne den Secure RPC nicht möglich wäre, zu dem Konzept dazugezählt werden. Diese Komponenten wurden vorgestellt und wie sie zur Erstellung der Beispielanwendung eingesetzt wurden.

Das RPC in DCE bietet von den betrachteten die besten Möglichkeiten, eine verteilte Anwendung zu erstellen. Die Beschreibungssprache IDL und der zugehörige Compiler stellen sehr praktische Mittel zur Verfügung, mit den benötigten Datenstrukturen umzugehen und lassen einen entfernten Prozeduraufruf zu großen Teilen wie ein lokalen Aufruf erscheinen. Durch den einfache Zugang zu den Komponenten des DCE lassen sich die anderen notwendigen Dienste (Security, Authentication, Verbindungsaufbau) einfach in eine Anwendung integrieren. Mit der Bereitstellung einer sicheren Datenübertragung auch auf UDP und der Angabe einer Prozedursemantik wird versucht, auftretende Netzwerkfehler soweit möglich bereits mit den Laufzeitroutinen zu bearbeiten. Nachteilig ist, daß die Art und Weise der Implementierung nicht veröffentlicht ist, die aber an einigen Stellen von Interesse ist, siehe Abschnitt 3.9.

Der große Vorteil an ONC-RPC ist, daß jederzeit nachvollzogen werden kann, wie die Rufe umgesetzt werden. Die RPC-API bietet etwas vereinfacht gesagt, einen

bequemeren Zugang zur Socketschnittstelle. Einerseits wird damit der zusätzliche Overhead minimiert, andererseits wird der Programmierer direkt mit den Fehlern und Beschränkungen des Transportprotokolls konfrontiert. Die Einfachheit und die offengelegte Spezifikation haben ONC-RPC zu einem Quasi-Standard werden lassen.

Mit OSI bestehen sehr flexible Möglichkeiten, verteilte Anwendungen zu beschreiben. Diese Flexibilität wird mit einer ebenso großen Komplexität erkauft, was bisher eine weite Verbreitung und Akzeptanz verhindert hat.

Die Beschreibung von entfernten Prozeduren mit ASN.1 nimmt keinen Bezug auf eine mögliche Implementierungssprache, wozu diese Sprache auch nicht entwickelt wurde.

Das führt im Endeffekt leider dazu, eine sehr flexible Beschreibung des Problems zu besitzen, aber durch einen Compiler unhandliche Datenstrukturen zu erhalten. Außerdem ist auch mit einer guten Dokumentation, wie sie z.B. mit ISODE ausgeliefert wird, einiger Einarbeitungsaufwand notwendig, um die Konzepte zu verstehen.

6.1 Ausblick

Mit der zunehmenden Vernetzung wächst die Möglichkeit, durch die Verteilung von Anwendungen, Informationen und Dienste weiter als in einem einzelnen Host oder einem LAN zu anzubieten. RPC kann das Konzept sein, daß es auch dem „netzwerkunerfahrenen“ Programmierer erlaubt, verteilte Anwendungen zu erstellen. ONC-RPC wird dabei sicherlich eine wichtige Rolle spielen. Daß aber ein Bedarf an leistungsfähigeren Compilern besteht, zeigt in jüngster Zeit die Vorstellung eines Compilers, der zwar Code für ONC-RPC erzeugt, aber eine, der IDL ähnliche Sprache benutzt[1].

An Bedeutung wird in Zukunft sicherlich auch die Frage gewinnen, wie Server ihre Dienste im Netzwerk und darüber hinaus anbieten können. Einen Ansatz gibt das DCE mit dem Zelldirectory. Über ein globales Directory können auch entfernte Server erreicht werden. Es bleibt abzuwarten, ob sich DCE durchsetzen wird, aber dieses Konzept könnte zur Verbreitung beitragen.

Wenn Server ihre Dienste nicht nur anbieten, sondern wenn auch die Möglichkeit besteht, einige Parameter auszuhandeln, wird in die Richtung Trading gegangen. Wieweit diese Konzepte allerdings Eingang in DCE finden können, ist noch unklar.

[1] IX-Mai-Ausgabe 1994

Anhang A

Schnittstellenbeschreibung DCE

```
[
uuid(1e1fdca6-3b2b-11cd-8d14-08002b3bdc7a),
version(1.0),
pointer_default(ptr)
]
interface rpcmail
{

typedef unsigned small flags;

/* unterstuetzte flags */
const flags F_NOFLAG              = 1;
const flags F_SEEN                = 2;
const flags F_ANSWERED            = 4;
const flags F_DELETED             = 8;
const flags F_FLAGGED             = 16;
const flags F_RECENT              = 32;

typedef    [string, ptr] char*    cstring;
typedef    [string, ref] char*    cstring_ref;

typedef struct statentry {
    unsigned long    size;
    flags            flags;
    cstring          header;
} statentry;

/* moegliche Daten fuer store */
typedef enum {
    FLAGS_SET, FLAGS_PLUS, FLAGS_MINUS
} store_data;

/*
 * die moeglichen Arten eines reply
 * ok .. erfolgreiche Ausfuehrung
 * no .. Fehler bei der Ausfuehrung
```

```
 * bad.. Protokollfehler
 */
typedef enum {
    ok, no, bad
} reply_status;

typedef unsigned short msgnr;

typedef struct interval {
    msgnr from;
    msgnr to;
} interval;

typedef struct sequence {
    unsigned    short      sequence_len;
    [size_is(sequence_len)] interval *sequence_val;
} sequence;

typedef    unsigned long endpt;
reply_status login_if(
    [in]    cstring_ref    user,
    [in]    cstring_ref    passwd,
    [out]   cstring        *endp,   /* der Endpoint des neuen Servers */
    [out]   cstring        *text    /* Begruessung */
);

reply_status logout_if(
    [out]    cstring* text
);

reply_status select_if(
    [in]    cstring_ref    mailbox,
    [out]    cstring       *text,    /* Text zur Mailbox */
    [out]    msgnr         *exists,
    [out]    msgnr         *recent
);

reply_status check_if(
    [out]    msgnr         *exists,
    [out]    msgnr         *recent,
    [out]    cstring       *text      /* Fehlertext */
);

typedef struct msgnrvector {
    short                  len;
    [size_is(len)]msgnr    *pmsgnr;
} msgnrvector;

reply_status expunge_if(
    [out]    msgnrvector   **expunged,
    [out]    cstring       *text      /* Fehlertext */
);

typedef struct statlist {
```

```
    short                   len;
    [size_is(len)]          statentry *pentry;
} statlist;

reply_status stat_if(
    [in]    msgnr           from,
    [in]    msgnr           to,
    [out]    statlist       **slist,
    [out]    cstring        *text       /* Fehlertext */
);

typedef    long    offset;
reply_status fetchbody_if(
    [in]    msgnr           nr,
    [in]    offset          off,
    [in]    offset          anz,
    [out,size_is(anz)] char buffer[],
    [out]    cstring        *text       /* Fehlertext */
);

reply_status store_if(
    [in]    sequence        *seq,
    [in]    store_data      data,
    [in]    flags           flag,
    [out]   cstring         *text        /* Fehlertext */
);

reply_status search_if(
    [in]    cstring_ref     sstring,
    [out]   sequence        **match,
    [out]   cstring         *text        /* Fehlertext */
);
}
```

Anhang B

Schnittstellenbeschreibung ONC

```
typedef unsigned flags;

const F_NOFLAG         = 1;
const F_SEEN           = 2;        /* S */
const F_ANSWERED       = 4;        /* A */
const F_DELETED        = 8;        /* D */
const F_FLAGGED        = 16;       /* F */
const F_RECENT         = 32;       /* R */

struct statentry {
        unsigned     size;
        flags        flags;
        string       header<>;
};

/* aktion fuer STORE */
enum store_data {
        FLAGS_SET, FLAGS_PLUS, FLAGS_MINUS
};

/*
 * die moeglichen Arten eines reply
 * ok .. Erfolg
 * no .. Fehler in der Abarbeitung
 * bad.. Protokollfehler
 */
enum reply_status {
        ok, no, bad
};

/* Nummer einer Nachricht im Server */
typedef unsigned msgnr;

struct interval {
        msgnr from;
        msgnr to;
```

```
};
typedef interval sequence<>;

const NOSTRING = "";

/********************** REQUEST / REPLY  -  DATENTYPEN ********************/

struct login_req {
        string          user<>;
        string          passwd<>;
};

union login_reply switch(reply_status status) {
        case ok :       long   prgnr;               /* neue Programmnummer */
        default :       string text<>;
};

union logout_reply switch(reply_status status) {
        case ok :       string bye<>;
        default :       string text<>;
};

/* Rueckgabe fuer SELECT */
struct text_exists_recent {
        string          text<>;
        msgnr           exists;
        msgnr           recent;
};

union select_reply switch(reply_status status) {
        case ok :       text_exists_recent reply;
        default :       string text<>;
};

/* Rueckgabe fuer CHECK */
struct exists_recent {
        msgnr           exists;
        msgnr           recent;
};

union check_reply switch(reply_status status) {
        case ok :       exists_recent reply;
        default :       string text<>;
};

/* Erfolg: Liste Nummern der geloeschten */
union expunge_reply switch(reply_status status) {
        case ok :       msgnr expunged<>;
        default :       string text<>;
};

union stat_reply switch(reply_status status) {
         case ok :      statentry msgs<>;
         default :      string text<>;
```

```
};

typedef long offset;

struct fetchbody_req {
        msgnr             nr;
        offset            off;
        offset            anz;
};

union fetchbody_reply switch(reply_status status) {
        case ok :         string body<>;
        default:          string text<>;
};

struct stat_req {
        msgnr             von;
        msgnr             bis;
};

struct store_req {
        sequence          sequence;
        store_data        data;
        flags             flags;
};

union store_reply switch(reply_status status) {
        case ok :         void;
        default:          string text<>;
};

union search_reply switch(reply_status status) {
        case ok :         sequence *msgs;
        default :         string text<>;
};

const RPCMAIL_PROTOKOLL = "tcp";

/************** DAS PROGRAMM INTERFACE *************/

program        RPCMAIL_PROG {
        version RPCMAIL_VERSION {

        login_reply LOGIN(login_req)             = 1;

        logout_reply LOGOUT(void)                = 2;

        select_reply SELECT(string)              = 3;

        check_reply CHECK(void)                  = 4;

        expunge_reply EXPUNGE(void)              = 5;

        stat_reply STAT(stat_req)                = 6;
```

```
        fetchbody_reply FETCHBODY(fetchbody_req)  = 7;

        store_reply STORE(store_req)              = 8;

        search_reply SEARCH(string)               = 9;
        } = 1;
} = 0x20000000;
```

Anhang C

Schnittstellenbeschreibung ROSE

```
MAIL DEFINITIONS ::=

BEGIN

-- wird in mit der benutzten Mothode nicht benoetigt
--
-- mailContext APPLICATION-CONTEXT
--     APPLICATION SERVICE ELEMENT    {aCSE}
--     BIND                           NULL
--     UNBIND                         NULL
--     REMOTE OPERATIONS              {rOSE}
--     INITIATOR CONSUMER OF          {mailClient}
--     ABSTRACT SYNTAXES              {aCSE-abstract-syntax, mail-abstract-syntax}
--     ::= {1 17 2 1 2}
--
-- mailClient APPLICATION-SERVICE-ELEMENT
--     CONSUMER INVOKES {login logout select check expunge stat fetchbody store search}
--     ::= {1 17 4 1 1}
--
-- mail-abstract-syntax OBJECT IDENTIFIER
--     ::= 1 17 2 1 1

-- operations

login OPERATION
    ARGUMENT    LoginReq
    RESULT      NULL
    ERRORS      { no, bad }
    ::=        0

logout OPERATION
    RESULT      IA5String
    ERRORS      { no, bad }
    ::=        1

select OPERATION
```

```
    ARGUMENT    IA5String
    RESULT      SelectResp
    ERRORS      { no, bad }
    ::=         2

check OPERATION
    RESULT      CheckResp
    ERRORS      { no, bad }
    ::=         3

expunge    OPERATION
    RESULT      MsgnrVector
    ERRORS      { no, bad }
    ::=         4

stat OPERATION
    ARGUMENT    StatReq
    RESULT      StatList
    ERRORS      { no, bad }
    ::=         5

fetchbody OPERATION
    ARGUMENT    FetchbodyReq
    RESULT      IA5String
    ERRORS      { no, bad }
    ::=         6

store    OPERATION
    ARGUMENT    StoreReq
    RESULT      NULL
    ERRORS      { no, bad }
    ::=         7

search    OPERATION
    ARGUMENT    IA5String
    RESULT      Sequence
    ERRORS      { no, bad }
    ::=         8

-- errors

no    ERROR
    PARAMETER   IA5String
    ::=         0

bad    ERROR
    PARAMETER   IA5String
    ::=         1

-- types

LoginReq ::=
    SEQUENCE {
        user          IA5String,
```

```
        passwd          IA5String
    }

Msgnr ::= INTEGER(0<..MAX)

SelectResp ::=
    SEQUENCE {
        greeting        IA5String,
        exists          INTEGER(0<..MAX),
        recent          INTEGER(0<..MAX)
    }

CheckResp ::=
    SEQUENCE {
        exists          INTEGER(0<..MAX),
        recent          INTEGER(0<..MAX)
    }

Flags ::=
    BIT STRING {
        seen (0), answered (1), deleted (2), flagged (3)
    }

Offset ::= INTEGER(0<..MAX)

MsgnrVector ::=
    SEQUENCE OF INTEGER(0<..MAX)

StatEntry ::=
    SEQUENCE {
        size            INTEGER(0<..MAX),
        flags           INTEGER(0..32),
        header          IA5String
    }

StatList ::=
    SEQUENCE OF
        StatEntry

StatReq ::=
    SEQUENCE {
        from            INTEGER(0<..MAX),
        to              INTEGER(0<..MAX)
    }

FetchbodyReq ::=
    SEQUENCE {
        nr              INTEGER(0<..MAX),
        len             INTEGER(0<..MAX),
        off             INTEGER(0<..MAX)
    }

Interval ::=
    SEQUENCE {
```

```
        from          INTEGER(0<..MAX),
        to            INTEGER(0<..MAX)
    }

Sequence ::=
    SEQUENCE OF
        Interval

Storedata ::=
    ENUMERATED {
        flagsset(0),
        flagsplus(1),
        flagsminus(2)
    }

StoreReq ::=
    SEQUENCE {
        sequence      Sequence,
        storedata     Storedata,
        flags         INTEGER(0..32)
    }

END
```

Bemerkung

Die von pepsy erzeugten Strukturen sind nicht immer günstig. Beispiel:

```
Msgnr ::= INTEGER(0<..MAX)

Interval ::=
    SEQUENCE {
        from    Msgnr,
        to      Msgnr
    }
```

ergibt in MAIL-types.h:

```
struct type_MAIL_Msgnr {
    int parm;
}

struct type_MAIL_Interval {
    type_MAIL_Msgnr    from;
    type_MAIL_Msgnr    to;
}
```

Obwohl `from` nur ein Integerwert ist muß auf den Inhalt über eine weitere Struktur zugegriffen werden. Der Quellkode wird dadurch unnötig komplizierter.

Deshalb wurde Interval ohne `Msgnr` vereinbart:

```
Interval ::=
    SEQUENCE {
        from    INTEGER(0<..MAX),
        to      INTEGER(0<..MAX)
    }
```

Es entsteht

```
struct type_MAIL_Interval {
    int    from;
    int    to;
}
```

An den Datentypen wird nichts geändert, der entstehende Kode wird aber übersichtlicher.

Literaturverzeichnis

[Blo] Bloomer, John; Power Programming with RPC, O'Reilly & Associates,Inc 1992

[Bric] Richter Birk; Performance Evaluation on OSF-DCE (Diplomarbeit TU Chemnitz-Zwickau 1994, Betreuer Prof. Kalfa)

[Com] Comer, Douglas E. / Stevens, David L. , Internetworking with TCP/IP Volume 3 , Prentice-Hall 1993

[Cor] Corbin, John R.; The Art of Distributed Applications, Springer Verlag 1991

[DEC] dxbook, ein Manual zu DEC DCE

[Hüb1] Hübner, Uwe; Verteiltes E-Mail-Zugangssystem auf RPC-Basis

[Hüb2] Hübner, Uwe; Planung und Management von Rechnernetzen (Vorlesungsscripte 1992)

[ISODE10] ISODE Consortium; Manual IC R1 von ISODE Version 8.0 Teil 10

[ISODE18] ISODE Consortium; Manual IC R1 von ISODE Version 8.0 Teil 18

[Ken] Rosenberry, Kenney, Fisher; Understanding DCE , O'Reilly & Associates,Inc 1992

[Mey] Meyer, Bertrand; Objektorientierte Softwareentwicklung, Hanser Verlag 1990

[Ros1] Rose, Marshall T.; The Open Book, Prentice Hall, Inc. 1990

[Ros2] Rose, Marshall T.; The Internet Message, Prentice Hall, Inc. 1993

[Sch] Schill, Alexander; DCE Einführung und Grundlagen, Springer Verlag 1993

[Shi] Shirley, John; Guide to Writing DCE Applications, O'Reilly & Associates,Inc 1992

[Tan2] Tannenbaum A.S.; Moderne Betriebssysteme, Hanser Verlag 1994

[X.400] Plattner B., ..; X.400 elektronische Post und Datenkommunikation, Addison-Wesley 1993 (3.Auflage)

[822] Crocker, David h.; RFC822: Standard for the Format of ARPA Internet Text Message 1982

[1014] SUN Microsystems; RFC1014 External Data Representation Syntax 1987

[1057] SUN Microsystems; RFC1057 RPC-Protocol Specification Version 2 1988

[1081] Rose, Marshall T.; RFC1081: Post Office Protocol - Version 3 1988

[1176] Crispin, M.; RFC1176: Interactive Mail Access Protocol - Version 2 1990

[1460] Rose, Marshall T.; RFC1460: Post Office Protocol - Version 3 1993