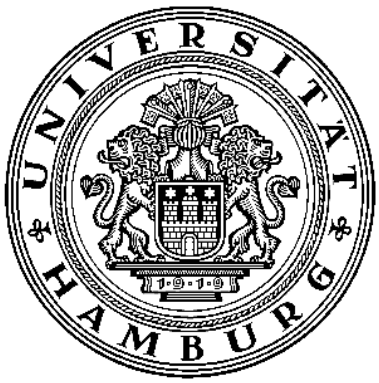

Universität Hamburg
FB Informatik
Datenbanken und Informationssysteme

STUDIENARBEIT

Entfernter Datenzugriff auf persistente Objektspeicher unter Benutzung standardisierter Protokolle (TSP/NFS)

November 1995

Lukas Gorzynski
Kroonhorst 126
22549 Hamburg
Tel.: +49 -40 -8316679

Betreuer:
Prof. Dr. Joachim W. Schmidt

Inhaltsverzeichnis

1. Einleitung **1**

1.1 Flache Datei-Strukturen und strukturierte Objektspeichersysteme 2

1.2 Gliederung der Arbeit 2

2. NFS - Ein hierarchisches Dateisystem **3**

2.1 Struktur und Arbeitsweise von NFS 3

2.2 Das NFS-Protokoll 5

3. TSP - Ein untypisierter Objektspeicher **7**

3.1 Die TSP-Schnittstelle 7

3.2 Datenstrukturen des TSP 8

3.2.1 Elementare Datentypen 8

3.2.2 Die allgemeine Objektstruktur 8

3.3 Operationen des TSP 9

3.3.1 Operationen auf Objektspeichern 10

3.3.2 Operationen auf Objektspeicherobjekten 10

4. Implementierung des Tycoon File System **12**

4.1 Das Datenmodell von TFS 12

4.1.1 Persistente Datenstrukturen 12

4.1.2 Dateideskriptor 13

4.2 Das TFS-Gateway 14

4.2.1 Die Serverkonfiguration 14

4.2.2 Prozeduren des NFS- und Mount-Protokolls 15

4.3 Erweiterte Funktionalität 17

4.4 Die NFS-Tycoon-Bibliothek 17

5. Grundlagen des entfernten Prozeduraufrufs **19**

5.1 Konzepte der Klient/Server-Kommunikation 19

5.2 Kommunikationsphasen 21

6. Administration von TFS **23**

6.1 Einrichten eines Tycoon-Dateisystems 23

6.2 Exportieren und Importieren von Dateisystemen 23

7. Zusammenfassung und Ausblick **27**

A. Schnittstellen des NFS- und Mount-Protokolls **29**

A.1 Schnittstellenbeschreibung des NFS-Protokolls 29

A.2 Schnittstellenbeschreibung des Mount-Protokolls 30

B. NFS- und Mount-Datenstrukturen **31**

C. Schnittstelle der Tycoon-Bibliothek *tfsenv* **39**

Literaturverzeichnis **41**

1. Einleitung

Das am Arbeitsbereich Datenbanken und Informationssysteme der Universität Hamburg entwickelte und implementierte *Tycoon*[1]-System ist eine moderne, integrierte Programmierumgebung [MMM93]. Das System bietet einen sprachlichen und architektonischen Rahmen für die Entwicklung persistenter Objektsysteme, die einen effizienten, flexiblen und sicheren Umgang mit großen Mengen langlebiger Objekte unterschiedlichster Art ermöglichen. Die Verwaltung dieser komplex strukturierten Datenobjekte erfolgt durch Objektspeichersysteme [MMS92].

Das im Rahmen dieser Arbeit entwickelte *Tycoon File System* (TFS) bietet die Möglichkeit, auf die Inhalte der Tycoon-Objektspeicher unter Verwendung von Dateisystemoperationen zuzugreifen. Mit den Operationen des *Tycoon Store Protocols* (TSP) bildet das TFS die Objektspeicherinhalte auf Dateisystemstrukturen ab. Auf der Basis des *Network File Systems* (NFS) implementiert das TFS einen Netzwerkdateiserver (*TFS-Gateway*) mit dem dann diese Dateisystemstrukturen in das Netzwerk exportiert werden (Abbildung 1.1). Somit stehen die Tycoon-Datenobjekte den Standard-Anwendungen als Dateisystemobjekte zur Verfügung. Es ist aber genauso möglich, daß Dateisystemobjekte von UNIX-Anwendungen in Tycoon-Programmen verwendet werden.

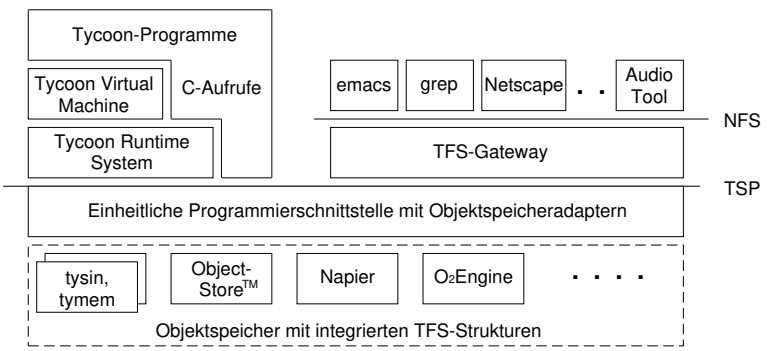

Abbildung 1.1: Integration von NFS in die persistente Umgebung von Tycoon

[1] Tycoon: Typed communicating objects in open environments

1.1 Flache Datei-Strukturen und strukturierte Objektspeichersysteme

Für den Benutzer erscheint das TFS als ein NFS-Dateisystem. Eine charakteristische Eigenschaft eines Dateisystems [Ster93] ist die Möglichkeit, flache, benannte Bytesequenzen (Dateien) in hierarchische Strukturen einzubetten. Das TFS bildet Dateisystemobjekte auf Objekte der Objektspeicher ab. Mit anonymen Objektspeicherobjekten werden Strukturen aufgebaut, die ausreichend Informationen enthalten, um die Abstraktion eines NFS-Dateisystems zu realisieren. Mit Hilfe des *TFS-Gateway* werden die NFS-Operationen auf Dateien und Verzeichnissen in Operationen auf den entsprechenden *TFS-Strukturen* in Objektspeichern überführt. Somit erscheinen Objektspeicherobjekte in einem importierten Dateisystem als benannte Dateien. Die als Verzeichnisse fungierenden strukturierten Objektspeicherobjekte können wie reguläre Verzeichnisse verwaltet werden. Mit Betriebssystemoperationen werden diese Objekte erzeugt, bearbeitet, umbenannt, verschoben oder gelöscht. Befehle wie *rm*, *cp*, *mv*, *vi* und *grep* können die Objektspeicherobjekte manipulieren, indem sie auf die Dateisystemobjekte (Dateien und Verzeichnissen) von TFS angewendet werden.

1.2 Gliederung der Arbeit

Das Kernstück von TFS ist das TFS-Gateway, ein Netzwerkdateiserver. Das dem TFS-Gateway zugrundeliegende NFS-Konzept und die wichtigsten Dateisystembegriffe werden in Kapitel 2 erörtert. In Kapitel 3 erfolgt eine kurze Darstellung der TSP-Schnittstelle mit den dazugehörigen Funktionen. Dabei wird nur auf die für die Implementation des Servers und die von ihm für den Aufbau und die Manipulation der TFS-Strukturen benötigten Funktionalitäten der Schnittstelle eingegangen. In Kapitel 4 werden die technischen Details der Implementation der TFS-Komponenten näher betrachtet. Das Kapitel stellt das Datenmodell von TFS vor und beschreibt die NFS-Operationen, die die Funktionalitäten des TFS-Servers realisieren. Es werden außerdem die Funktionen erläutert, mit denen aus Tycoon-Programmen heraus die TFS-Strukturen manipuliert werden können. Kapitel 5 gibt einen Überblick über die Programmierung von verteilten Anwendungen mit Hilfe des *entfernten Prozeduraufrufs*. Kapitel 6 zeigt wie das TFS eingerichtet und genutzt werden kann. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf mögliche Erweiterungen von TFS.

2. NFS - Ein hierarchisches Dateisystem

Das von *Sun Microsystems* entwickelte *Network File System*-Protokoll (NFS) stellt Operationen zur Verfügung, die einen transparenten Zugriff auf Dateien in einem Netzwerk ermöglichen [StAh87]. Mit NFS erscheinen Dateien eines entfernten Dateisystems wie Dateien der lokalen Dateisysteme. NFS bietet Transparenz in Bezug auf Netz und Dateisystem, da es die tatsächliche physikalische Position und die Struktur des Dateisystems versteckt. Die Benutzung der Ressourcen auf allen Maschinen im Netzwerk wird damit vereinheitlicht und stimmt mit den Abläufen in einer nicht vernetzten Umgebung überein.

Das NFS-Protokoll ist für verschiedene Rechnerarchitekturen, Betriebssysteme, Netzwerkarchitekturen und Transportprotokolle verfügbar. Diese Eigenschaft wird durch die Nutzung eines standardisierten Protokolls für entfernte Prozeduraufrufe (*Remote Procedure Call*, *RPC*), das eine standardisierte, portable Datenrepräsentation (*External Data Representation*, *XDR*) verwendet, erreicht. Mit dem RPC-Protokoll sowie der XDR-Datenrepräsentation beschäftigt sich Kapitel 5 dieser Arbeit.

2.1 Struktur und Arbeitsweise von NFS

NFS ist ein verteiltes Dateisystem [Sand85]. Ein *NFS-Server* stellt ein oder mehrere Dateisysteme zur Verfügung, die von *NFS-Klienten* in den sogenannten *Mount-Punkten* importiert werden. Für die NFS-Klienten erscheinen die entfernten Dateien wie lokale Dateien. Alle Programme können mit diesen NFS-Dateien wie mit Dateien auf lokalen Festplatten arbeiten.

Seine Transparenz erreicht NFS durch die Definition einer Grundmenge von Dateisystemoperationen, die auf einem *virtuellen Dateisystem* (*Virtual File System*, *VFS*) durchgeführt werden. Über VFS kann ein Klient auf viele verschiedene Typen von verteilten Dateisystemen zugreifen. VFS verbirgt die unterschiedlichen Implementierungen der Dateisysteme unter einer einheitlichen Schnittstelle, die auf einem Klienten unter UNIX dafür sorgt, daß alle NFS-Dateisysteme wie UNIX-Dateisysteme aussehen [Tan92].

Die Arbeitsweise des NFS stellt die Abbildung 2.1 dar. Ein Dateisystemaufruf wird vom VFS auf den lokalen Dateisystemen ausgeführt oder in einen RPC-Aufruf einer NFS-Operation

auf dem Dateiserver umgewandelt. Auf dem Server besteht die Implementierung von VFS darin, diese RPC-Operation entgegenzunehmen und sie in die jeweilige Aktionen auf dem tatsächlich zugrundeliegenden Dateisystem umzuwandeln.

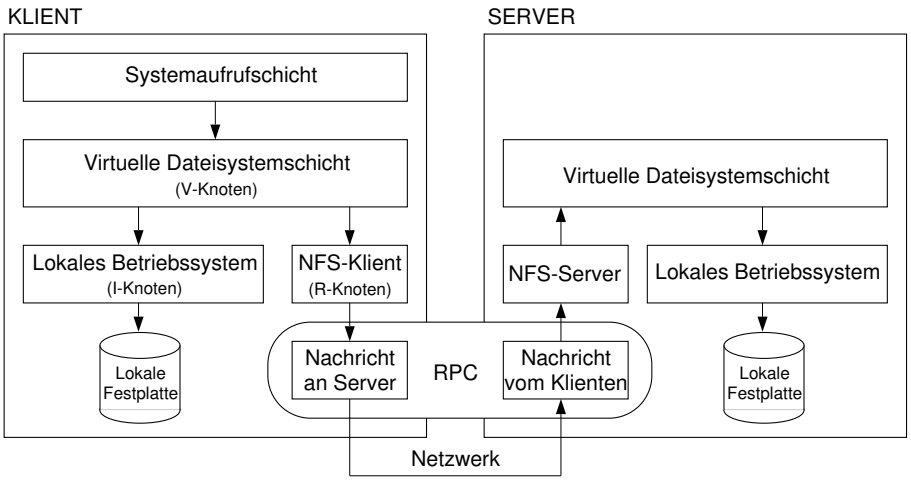

Abbildung 2.1: Architektur von NFS

Ein NFS-Dateisystem besteht, genau wie ein UNIX-Dateisystem, aus zwei Arten von benannten Objekten: *Dateien* und *Verzeichnissen*. Dateien dienen der transparenten Speicherung von Informationen auf einem Speichermedium. Eine Datei ist eine unstrukturierte Sequenz von Bytes. Die Dateien werden in Verzeichnissen verwaltet, die ihrerseits rekursiv geschachtelt werden können. Ein Dateisystem besitzt somit eine *hierarchische Struktur*.

Um Dateien zu verwalten, bietet das NFS-Dateisystem Verzeichnisse an. Ein Verzeichnis enthält eine Anzahl von Einträgen. Ein Eintrag wiederum enthält entweder den Namen einer Datei oder den eines Unterverzeichnisses sowie einen Zeiger auf eine weitere Datenstruktur, den *virtuellen Knoten* (V-Knoten). Die virtuellen Knoten besitzen eine ähnliche Struktur wie die *Index-Knoten* in UNIX-Dateisystemen (I-Knoten, [PeSi88; Bach86]). Sie enthalten Informationen bezüglich der Lokalität der Dateien. Sie legen fest, ob Dateien lokal vorhanden oder auf einem entfernten Rechner gespeichert sind. Virtuelle Knoten verweisen entweder auf die *Index-Knoten* im lokalen Betriebssystem oder auf die Datenstrukturen im Kode des NFS-Klienten (R-Knoten, *Knoten der entfernten Dateien*), die ausreichend Informationen enthalten, um Dateien der importierten Dateisysteme eindeutig zu referenzieren.

Dateien und Verzeichnisse werden über einen abstrakten Dateiidentifikator referenziert. Um auf einem der Objekte des NFS-Dateisystems Operationen ausführen zu können, muß sein Identifikator bekannt sein. Bei Operationen auf lokalen Dateisystemen werden die Identifikatoren von dem lokalen Betriebssystem vergeben. Im Falle einer NFS-Anfrage übernimmt

diese Aufgabe, je nach Dateisystemimplementation, das Betriebssystem des entfernten Rechners oder das VFS (bzw. der NFS-Server). Die Identifikatoren von entfernten Dateien werden in den Knoten der entfernten Dateien zusammen mit allen anderen Informationen, die zur eindeutigen Identifikation der Dateien benötigt werden, gespeichert.

Mit jedem Dateisystemobjekt sind *Attribute* assoziiert. In der Tabelle 2.1 sind alle in NFS-Dateisystemen benutzten Attribute beschrieben.

Attribut	Beschreibung
type	Typ der Datei
mode	Typ und Zugriffsrechte der Datei (legen zusammen mit uid und gid fest, wer in welchem Umfang auf die Datei zugreifen kann)
nlink	Anzahl der harten Verweise auf die Datei
uid	Identifikator des Inhabers der Datei
gid	Identifikator der Gruppe, zu der die Datei gehört
size	Größe der Datei in Bytes
blocksize	Größe eines Dateisystemblocks in Bytes
rdev	Laufwerksbezeichnung (nur bei speziellen Zeichen- und Block-Dateien)
blocks	Größe der Datei in Blöcken
fsid	Identifikator des Dateisystems
fileid	Eindeutiger Identifikator der Datei innerhalb des Dateisystems
atime	Datum und Uhrzeit der Erzeugung der Datei
mtime	Datum und Uhrzeit des letzten Zugriffs auf die Datei
ctime	Datum und Uhrzeit der zuletzt durchgeführten Modifikation

Tabelle 2.1: NFS-Dateiattribute

2.2 Das NFS-Protokoll

Das RPC-Protokoll von NFS definiert Prozeduren, die auf den Objekten der Typen Datei, Verzeichnis, Verweis oder Dateisystem ausgeführt werden können. Auf Dateien operieren Prozeduren, die ihrer Erzeugung oder Entfernung und dem Lesen oder Schreiben ihrer Inhalte dienen. Die Verzeichnisoperationen realisieren UNIX-Systemaufrufe, die Verzeichnisse anlegen, lesen oder löschen. Für die Verwaltung der Verweise sind Verweisoperationen verantwortlich. Einige NFS-Prozeduren beziehen sich sowohl auf Dateien als auch auf Verzeichnisse und Verweise. Mit diesen Prozeduren können die Eigenschaften der Dateisystemobjekte abgefragt oder verändert werden.

Darüber hinaus existiert eine Gruppe von Prozeduren, deren Aufgabe es ist, die Benutzung entfernter Dateisysteme zu organisieren. Diese Prozeduren gehören zum Mount-Protokoll

und sind von NFS getrennt. Beide Protokolle nutzen aber den gleichen Identifikationsmechanismus. Die Operationen zum Exportieren und Importieren von NFS-Dateisystemen sind die wichtigsten Operationen des Mount-Protokolls.

Das NFS-Protokoll ist *zustandslos*. Auf dem Server werden keine Informationen über frühere NFS-Anfragen oder über den Zusammenhang verschiedener NFS-Anfragen [Ster93] gehalten. Der Klient verwaltet alle Informationen, die für Anfragen an den Server nötig sind.

Die Wahl eines zustandslosen Protokolls hat mehrere Auswirkungen auf Struktur und Implementierung von NFS:

- Jede NFS-Anfrage enthält genügend Informationen, um ohne einen Verweis auf den Zustand des Servers oder des Klienten abgeschlossen zu werden.
- Da der NFS-Server keinen Zustand speichert, muß bei einem Ausfall des Servers der Klient alle nicht bestätigten Anfragen wiederholen.
- Da nicht alle NFS-Operationen idempotent sind, können mehrfach ausgeführte Anfragen zu unerwünschten Ergebnissen führen. Zum Beispiel kann ein wiederholter Versuch, eine Datei zu erzeugen oder zu löschen, zu Problemen auf der Klienten-Seite führen.

Die Wahl eines zustandslosen Protokolls minimiert den Aufwand beim Wiederaufsetzen nach einem Systemfehler. Da der Server zustandsunabhängig ist, kann er zu beliebigen Zeitpunkten neu gestartet werden.

Im Gegensatz zum NFS-Server verwaltet der Mount-Server eine Liste mit Daten seiner Klienten. Diese Liste hat nur Informationscharakter und ist deswegen für die eigentliche Funktionsfähigkeit des Mount-Servers und seiner Klienten nicht relevant.

Zur Übertragung von RPC-Anfragen wird das verbindungslose UDP-Protokoll [Post80] verwendet. UDP ist zustandsfrei und garantiert weder die Zustellung der Pakete noch ihre zeitliche Reihenfolge.

3. TSP - Ein untypisierter Objektspeicher

Gegenstand dieses Kapitel ist das *Tycoon Store Protocol* (TSP). TSP ist eine datenmodellunabhängige Objektspeicherschnittstelle, die am Arbeitsbereich DBIS der Universität Hamburg entwickelt wurde.

Das in dieser Arbeit realisierte TFS nutzt die TSP-Funktionalität, um die NFS-Dateisystemstrukturen auf Objektspeicherstrukturen abzubilden. Darüber hinaus macht das TFS von den transaktionalen Fehlererholungsmechanismen der TSP-Schnittstelle Gebrauch (siehe Abschnitt 4.3).

3.1 Die TSP-Schnittstelle

Die TSP-Schnittstelle gestattet eine vollständige Abstraktion von den operationalen Eigenschaften der Objektspeicher und bietet ihren Benutzern somit eine einheitliche Sicht auf untypisierte Objektspeicher:

Persistenz: Das Konzept der *Persistenz* läßt zu, daß jedes Objekt potentiell länger leben kann als der es erzeugende Betriebssystemprozeß. Ein Objekt wird persistent, wenn es über ein ausgezeichnetes Wurzelobjekt transitiv erreichbar ist.

Atomarität: TSP erlaubt das Setzen von *Sicherungspunkten*. Durch das Setzen eines Sicherungspunktes werden die seit dem letzten Sicherungspunkt durchgeführten Modifikationen im Objektspeicher persistent gespeichert.

Fehlererholung: Die seit dem letzten Sicherungspunkt an persistenten Objekten durchgeführten Änderungen können durch eine explizite *Rücksetzoperation* (*Rollback*) rückgängig gemacht werden.

Freispeicherverwaltung: Nicht mehr erreichbare Objekte werden durch einen *Speicherrückgewinnungsmechanismus* (*Garbage Collection*) automatisch aus dem Objektspeicher gelöscht.

3.2 Datenstrukturen des TSP

TSP unterstützt zwei Arten von persistenten Datenstrukturen: *Feldobjekte* und *Bytefeldobjekte*. Sie können Werte der Klientensysteme aufnehmen oder zur Speicherung von Identifikatoren anderer Objekte benutzt werden.

3.2.1 Elementare Datentypen

Das TSP ist eine parametrisierte Schnittstelle. Die elementaren Typen werden unabhängig von der operationalen Schnittstelle definiert. Die Beziehungen zwischen den einzelnen TSP-Typen stellt die Abbildung 3.1 dar.

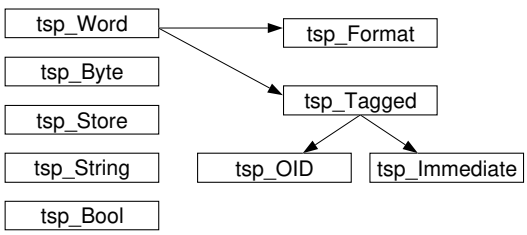

Abbildung 3.1: Hierarchie der TSP-Typen

Von den definierten TSP-Typen besitzt der Typ *tsp_Word* besondere Beduetung. Der typischerweise 16, 32 oder 64 Bit lange Typ wird als Supertyp der ganzen Typ-Hierarchie genutzt. Werte vom Typ *tsp_Format* dienen der Beschreibung von Objektstrukturen. Der Typ *tsp_Tagged* vereinigt Werte vom Typ *tsp_OID* und *tsp_Immediate*. Werte vom Typ *tsp_OID* sind Identifikatoren für Objekte, Werte vom Typ *tsp_Immediate* repräsentieren atomare Werte des Klientensystems. Der Typ *tsp_Tagged* wird in jedem Kontext benutzt, in dem *tsp_OID* sowie *tsp_Immediate* gültig sind.

3.2.2 Die allgemeine Objektstruktur

Beide unterstützten Objektarten, *Feldobjekte* und *Bytefeldobjekte*, besitzen die in der Abbildung 3.2 dargestellte Struktur.

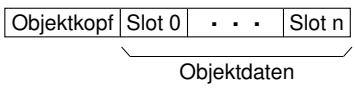

Abbildung 3.2: Struktur der TSP-Objekte

Ein Objekt besteht aus einem *Objektkopf* und einer benutzerdefinierbaren Anzahl von Feldern (im Folgenden als *Slots* bezeichnet):

Objektkopf: Im Objektkopf werden Informationen über das Format des Objektes, die Anzahl seiner Slots und weitere Verwaltungsdaten gespeichert.

Objektdaten: Die Objektgröße ist konzeptuell nicht beschränkt. In Abhängigkeit vom Objektformat können in den Slots Werte vom Typ *tsp_Word*, *tsp_Tagged* oder *tsp_Byte* gespeichert werden. In die Slots von Feldobjekten können sowohl Werte vom Typ *tsp_Word* als auch vom Typ *tsp_OID* und *tsp_Immediate* aufgenommen werden. Bytefeldobjekte nehmen nur Werte vom Typ *tsp_Byte* auf.

Mit jedem Objekt ist ein eindeutiger *Objektidentifikator* assoziiert. Die Objektidentifikatoren können in Feldobjekten gespeichert werden, um Beziehungen zwischen Objekten zu modellieren. Objektidentifikatoren sind veränderlich. Sie können durch bestimmte Systemoperationen geändert werden. TSP garantiert jedoch die Erhaltung der Topologie des Objektgraphen. TSP übernimmt die automatische Aktualisierung von Objektidentifikatoren, die in Datenstrukturen eines Objektspeichers gespeichert sind und unterstützt die Aktualisierung von Objektidentifikatoren, welche außerhalb des Objektspeichers gehalten werden.

3.3 Operationen des TSP

Abbildung 3.3 zeigt die möglichen Zustände eines Objektspeichers und die durch die Ausführung der TSP-Prozeduren ausgelösten Zustandsübergänge.

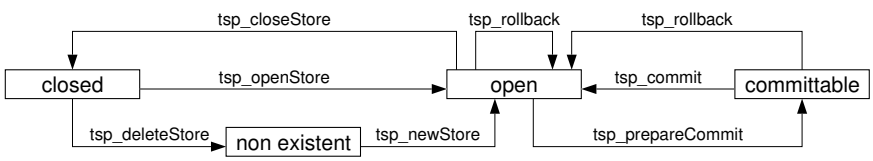

Abbildung 3.3: Zustände eines Objektspeichers

3.3.1 Operationen auf Objektspeichern

Bevor TSP-Operationen benutzt werden können, müssen einige Sitzungsstrukturen, die von einigen Objektspeicherimplementationen benötigt werden, erzeugt und initialisiert werden. Dies geschieht durch Aufruf der Funktion *tsp_init()*.

Nach der Initialisierung einer Protokollinstanz kann ein Objektspeicher mit *tsp_newStore()* neu erzeugt oder mit *tsp_openStore()* geöffnet werden. Wenn es gelingt, den Objektspeicher zu öffnen, liefern beide Funktionen einen abstrakten Objektspeicheridentifikator (vom Typ *tsp_Store*) zurück. Die nachfolgenden TSP-Operationen benötigen diesen Identifikator, um den Objektspeicher, in dem die Operationen ausgeführt werden sollen, zu identifizieren.

Zusätzlich bietet das TSP Funktionen zur Stabilisierung der Zustände eines Objektspeichers. Die *tsp_commit()*-Operation speichert alle neu erzeugten und modifizierten Objekte seit der letzten Stabilisierung. Bei der Nutzung des 2-Phasen-Commit-Protokolls muß vor dem Aufruf von *tsp_commit()* die Operation *tsp_prepareCommit()* aufgerufen werden. Ein erfolgreicher Abschluß dieser Prozedur garantiert eine fehlerlose Speicherung der Daten auf einem persistenten Speichermedium während der Stabilisierung. Mit der Funktion *tsp_rollback()* können die seit dem letzten *tsp_commit()* vorgenommenen Änderungen verworfen werden. Es wird bei dem zuletzt stabilisierten Objektspeicherzustand aufgesetzt.

Die Operation *tsp_closeStore()* beendet die Benutzung eines Objektspeichers. Sie invalidiert den Objektspeicheridentifikator, so daß keine weiteren Objektoperationen auf dem Objektspeicher ausgeführt werden können. Der Objektspeicher befindet sich dann in dem Zustand *closed*. Nur in diesem Zustand kann ein Objektspeicher mit der Operation *tsp_deleteStore* gelöscht werden.

3.3.2 Operationen auf Objektspeicherobjekten

TSP stellt eine Menge von Operationen zur Manipulation von Objekten zur Verfügung. Dabei unterscheidet man zwischen Operationen zur Erzeugung von Objekten, Operationen, mit denen die Eigenschaften der Objekte verändert oder ermittelt werden können, und Operationen, die den schreibenden oder lesenden Zugriff auf der Daten der Objekte realisieren.

Der Einstieg in den Objektgraphen eines Objektspeicher wird durch ein Wurzelobjekt ermöglicht. Durch die Operation *tsp_setRoot()* kann ein Wurzelobjekt in einem Objektspeicher definiert werden. Der Objektidentifikator eines Wurzelobjektes kann mit *tsp_getRoot()* ermittelt werden.

TSP stellt Operationen zum Erzeugen von initialisierten oder nicht initialisierten Objekten zur Verfügung. Als Parameter muß diesen Funktionen die Größe des Objektes, sein Format und gegebenenfalls der Initialisierungswert übergeben werden. Die Größe eines Objekts kann mit der Funktion *tsp_nSlots()* ermittelt und mit der Operation *tsp_resize()* nachträglich geändert werden. Das Format der Objektdaten kann durch *tsp_setFormat()* und *tsp_getFormat()* modifiziert und gelesen werden.

TSP bietet außerdem eine Sammlung von Operationen zum Ermitteln und Modifizieren von einzelnen oder mehreren Datenfeldern der Objekte. Die meisten Operationen erlauben das selektive Lesen bzw. Schreiben der Objektdaten. Mittels der Operationen *tsp_moveWords()* und *tsp_moveBytes()* können Daten zwischen Objekten oder innerhalb eines Objekts kopiert werden.

4. Implementierung des Tycoon File System

Mittels TFS sollen Objektspeicherobjekte auf die gleiche Weise wie Dateien in einem NFS-Dateisystem durch Standard-UNIX-Anwendungen manipuliert werden können.

Die Realisierung von TFS umfaßt:

- die Implementation von Objektspeicherstrukturen, mit denen NFS-Dateisysteme in Objektspeichern realisiert werden,
- die Implementation eines Gateways, das die Umsetzung der NFS-Anfragen in TSP-Operationen eines Objektspeichers realisiert,
- die Implementation von Tycoon-Funktionen, die Tycoon-Datenobjekte auf Dateisystemobjekte des Objektspeichers abbilden.

4.1 Das Datenmodell von TFS

Die Integration von NFS und TSP verlangt geeignete *persistente Strukturen*, die die hierarchische Architektur der Dateisysteme in Objektspeichern abbilden und dadurch die Abstraktion eines NFS-Dateisystems in ihnen realisieren. Außerdem wird ein abstrakter *Dateideskriptor* für die eindeutige Identifikation der NFS-Dateien bzw. der sie repräsentierenden Objektspeicherobjekte benötigt.

4.1.1 Persistente Datenstrukturen

Zur Abbildung der baumartigen Dateisystemstruktur im Objektspeicher werden in TFS dynamisch Datenstrukturen (*TFS-Objekte*) erzeugt, in denen alle für ein Dateisystem relevanten Informationen verwaltet werden. Sie enthalten die für den Aufbau von Dateisystemhierarchien nötigten Verweise auf andere TFS-Objekte sowie zusätzliche Informationen über die abgebildeten Dateien und Verzeichnisse.

Alle TFS-Strukturen werden auf Objektspeicherobjekte abgebildet. Aufgrund des Persistenzkonzeptes von TSP sind alle TFS-Objekte über ein ausgezeichnetes Wurzelobjekt eines Objektspeicher erreichbar.

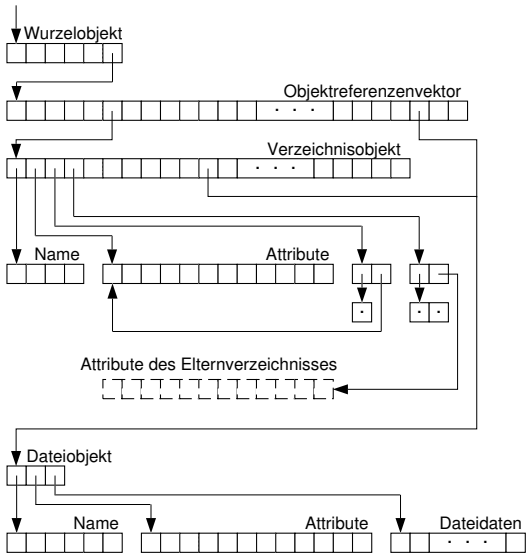

Abbildung 4.1: TFS-Strukturen in einem Objektspeicher

TFS-Objekte, die Verzeichnisse repräsentieren (*Verzeichnisobjekte*), werden durch ein Tupel realisiert, das einen Namen, eine Liste von Attributen (siehe Abschnitt 2.1) und eine Anzahl von Verweisen auf andere TFS-Objekte enthält. In jedem Verzeichnis existieren Verweise auf das Elternverzeichnis und das Verzeichnis selbst. Hierfür werden unter UNIX die Symbole ".." und "." verwendet. TFS-Objekte dieser Verzeichniseinträge bestehen nur aus einem Namen und einer Liste von Attributen.

Dateien und symbolische Verweise werden durch *Dateiobjekte* repräsentiert. Die Dateiobjekte besitzen, außer dem Namen und der Attributliste, einen Verweis auf *Dateidaten*. In der vorliegenden TFS-Implementation werden die Dateidaten durch Bytefelder realisiert.

Darüber hinaus existiert in TFS ein Objektreferenzenvektor, durch den der Zugriff auf TFS-Objekte optimiert wird.

4.1.2 Dateideskriptor

Alle NFS-Operationen benutzen *Dateideskriptoren*, um die Dateien oder Verzeichnisse zu bestimmen, auf den sie ausgeführt werden sollen. Die Deskriptoren werden auf dem Server erzeugt und enthalten Informationen, die Dateien oder Verzeichnisse eindeutig identifizieren.

Für den Dateideskriptor sieht NFS 32 Bytes vor. TFS nutzt die ersten 4 Bytes für einen Objektspeicherindex. Der Index bestimmt den Objektspeicher, in dem sich das durch den Deskriptor referenzierte Objekt befindet. In den folgenden 4 Bytes wird die Position des Verweises auf das referenzierte Datei- oder Verzeichnisobjekt im Objektreferenzenvektor gespeichert. Diese beiden Angaben reichen aus, um die eindeutige Identifikation der Objekte zu gewährleisten. Die restlichen 24 Bytes bleiben unbenutzt.

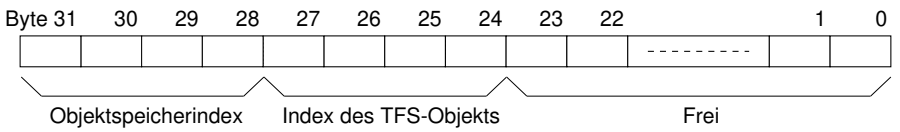

Abbildung 4.2: Realisierung des Dateideskriptors

Der Dateideskriptor enthält für den Klienten keine nutzbaren Informationen. Da seine Struktur von der Implementation des Dateisystems abhängt, kann nur der Dateiserver den Dateideskriptor interpretieren.

4.2 Das TFS-Gateway

Der Austausch von Daten zwischen einem Objektspeicher und einer Anwendung verläuft grundsätzlich über ein Gateway, das für die Umsetzung der NFS-Anfragen auf Funktionen der Objektspeicherschnittstelle sorgt. Die Aufgaben des Gateways bestehen darin, den NFS-Klienten die Nutzung der TFS-Dateisysteme über die NFS-Schnittstelle zu ermöglichen und NFS-Anfragen des Klienten in die jeweiligen Aktionen auf dem TFS-Dateisystem umzuwandeln.

Das Gateway zum TFS-Dateisystem wurde als zweistufiger NFS-Server realisiert. Im Folgenden werden die Beziehungen zwischen den einzelnen Komponenten des NFS-Gateways und die Gründe für den zweistufigen Lösungsansatz erörtert.

4.2.1 Die Serverkonfiguration

Das Gateway besteht aus dem *NFS-Dämon* (*nfsd*), dem *Mount-Dämon* (*mountd*) und dem *TFS-Dämon* (*tfsd*). Die eigentliche Funktionalität des Gateways (NFS- und Mount-Prozeduren) ist im TFS-Dämon implementiert. Der NFS- und der Mount-Dämon dienen ausschließlich der Weitergabe von NFS-Anfragen an den TFS-Dämon. Beide laufen auf einem als NFS-Server dedizierten Rechner und sind auf die gleiche Art und Weise wie die gleichnamigen UNIX-Dämonen mit Hilfe von RPC ansprechbar. Den Weg einer NFS-Anfrage zeigt die Abbildung 4.3.

Der TFS-Server, d.h. der Rechner, auf dem der TFS-Dämon gestartet wird, muß unter bestimmten Umständen vom NFS-Server verschieden sein. Da das TFS-Gateway nicht mit der Absicht entwickelt worden ist, die Funktionalitäten der Standard-NFS-Dämonen zu implementieren, dürfen der NFS-Dämon und Mount-Dämon nicht auf dem gleichen Rechner gestartet werden, auf dem UNIX-Dateisysteme exportiert werden. Soll der TFS-Dämon auf einem solchen Rechner laufen, ist es notwendig, für den TFS-Server und den NFS-Server verschiedene Rechner einzusetzen.

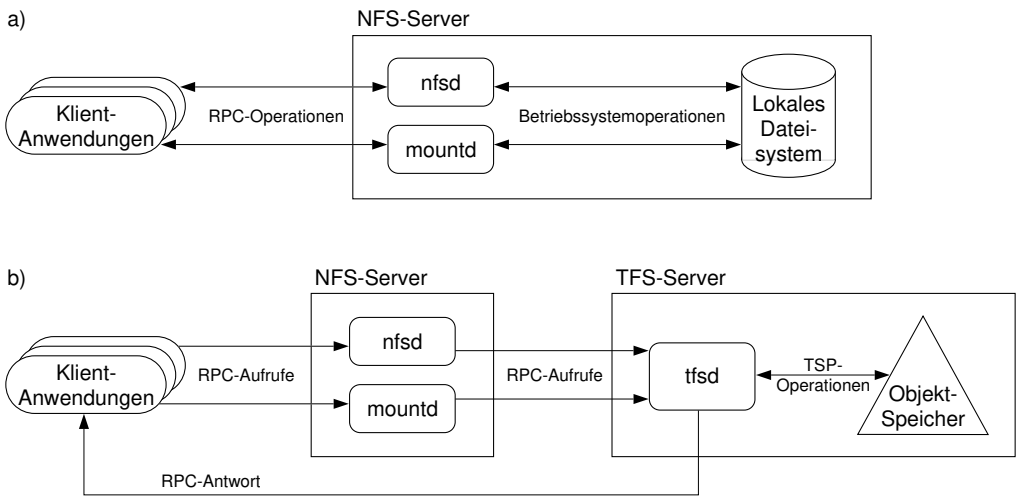

Abbildung 4.3: NFS-Anfragen im Standard-Fall (a) und beim TFS-Gateway (b)

4.2.2 Prozeduren des NFS- und Mount-Protokolls

NFS-Protokoll und Mount-Protokoll sind durch RPC-Prozeduren implementiert, die das XDR-Protokoll benutzen, um Argumente zwischen Klient und Server zu übergeben. Bei der Implementierung des TFS-Gateways wurde die Version 2 des NFS-Protokolls mit der standardisierten NFS-Programmnummer 100003 sowie die Version 1 des Mount-Protokolls verwendet. Die vollständige Schnittstellenbeschreibung beider Protokolle ist in Anhang A zu finden. Im Folgenden wird auf die Funktion der einzelnen Prozeduren näher eingegangen.

Mount-Operationen Bevor auf ein TFS-Dateisystem zugegriffen werden kann, muß der Dateideskriptor seines Stammverzeichnisses bekannt sein. Die Prozedur *MOUNT_PROC_MNT* sorgt dafür, daß dem Klienten der Deskriptor des von ihm importierten Dateisystems bekannt gegeben wird. Zusätzlich trägt sie den Klienten in eine interne Liste ein, in der alle die TFS-Dateisysteme importierenden Rechner verwaltet werden. Als Parameter benötigt diese Prozedur den vollen Pfad auf die Objektspeicherdatei, die das TFS-Dateisystem enthält. Die Operation *MOUNT_PROC_DUMP*

liefert eine Liste der bereits importierten TFS-Dateisysteme. Die Liste enthält jeweils einen Eintrag für jedes Klient-Dateisystem-Paar. Beim Entfernen der importierten TFS-Dateisysteme aus der Dateibaumstruktur ruft der TFS-Klient entweder die *MOUNT_PROC_UMNT*- oder die *MOUNT_PROC_UMNTALL*-Prozedur auf. Beide Prozeduren entfernen die entsprechenden Einträge aus der Liste der importierten TFS-Dateisysteme. Die Prozedur *MOUNT_PROC_UMNT* löscht den Eintrag für das vom Klienten angegebene TFS-Dateisystem. Mit *MOUNT_PROC_UMNTALL* werden alle Einträge des Klienten gelöscht. Die Operationen *MOUNT_PROC_EXPORT* und *MOUNT_PROC_EXPORTALL* listen Dateisysteme auf, die von dem TFS-Server exportiert werden. Außerdem liefern sie Informationen darüber, welche Dateisysteme von welchen Rechnern benutzt werden dürfen.

Operationen auf den Dateisystemobjekten Mit jedem Dateisystemobjekt sind Attribute assoziert, die mit der Operation *NFS_PROC_GETATTR* abgefragt und mit *NFS_PROC_SETATTR* modifiziert werden können. Um auf eine Datei oder ein Verzeichnis zugreifen zu können, muß der entsprechende Dateideskriptor bekannt sein. Ein Klient erhält Dateideskriptoren für existierende Dateien mit der Anfrage *NFS_PROC_LOOKUP*. Die *NFS_PROC_LOOKUP*-Prozedur sucht einen Verzeichniseintrag über den Namen und liefert seinen Dateideskriptor zurück. Mit *NFS_PROC_RENAME* können Dateien, Verzeichnisse und Verweise in einem Verzeichnis umbenannt oder in ein anderes Verzeichnis verschoben werden. Diese Prozedur wird exzessiv bei UNIX-Befehlen zum Kopieren und Verschieben von Dateien benutzt.

Dateioperationen Die Dateioperationen *NFS_PROC_READ* und *NFS_PROC_WRITE* übertragen Daten vom und zum TFS-Klienten. Mittels der Lese- und Schreiboperationen können Dateiabschnitte gelesen oder geschrieben werden. Die Prozedur *NFS_PROC_CREATE* legt einen neuen Verzeichniseintrag für eine Datei an. Der Name und die Initialisierungsattribute für die neue Datei werden der Prozedur als Parameter übergeben. Die Dateien können mit Hilfe der *NFS_PROC_REMOVE*-Prozedur gelöscht werden.

Verzeichnisoperationen Die Verzeichnisoperationen enthalten *NFS_PROC_MKDIR* und *NFS_PROC_RMDIR*, die wie die entsprechenden UNIX-Systemaufrufe Verzeichnisse anlegen und löschen. Die Operation *NFS_PROC_RMDIR* kann nur dann ihre Aufgabe ausführen, wenn das zu löschende Verzeichniss leer ist. Die Operation *NFS_PROC_READDIR* liest ein Verzeichnis, wobei ein Verzeichniszeiger für fortlaufende Lesevorgänge im gleichen Verzeichnis benutzt wird.

Verweisoperationen Bei den Verweisoperationen gibt es die Prozeduren *NFS_PROC_LINK*, *NFS_PROC_SYMLINK* und *NFS_PROC_READLINK*. Die *NFS_PROC_LINK*-Prozedur richtet einen neuen Verzeichniseintrag auf eine existierende Datei ein. Die so erzeugten Einträge (*hard links*) unterscheiden sich nicht von den regulären Verzeichniseinträgen. Die Prozeduren *NFS_PROC_SYMLINK* und *NFS_PROC_READLINK* erzeugen einen symbolischen Verweis (*symbolic link*) beziehungsweise lesen dessen Wert.

Die symbolischen Verweise sind Verzeichniseinträge eines speziellen Typs. Sie enthalten die Pfade der Dateisystemobjekte, auf die sie verweisen.

Dateisystemoperationen Die Prozedur *NFS_PROC_STATFS* ist eine Dateisystemoperation, die Informationen über das importierte Dateisystem liefert. Durch Aufruf dieser Prozedur hat der Klient die Möglichkeit sich über den Zustand des entfernten TFS-Dateisystems zu informieren.

4.3 Erweiterte Funktionalität

Über die Funktionalität eines NFS-Server hinaus unterstützt das TFS die Benutzung des Fehlererholungsmechanismus des TSP-Protokolls. Mittels TSP-Prozeduren können Inhalte eines TFS-Dateisystems persistent gespeichert oder die seit der letzten Speicherung vorgenommenen Änderungen an Dateien und Verzeichnissen rückgängig gemacht werden.

Die Implementation des TFS-Gateways unterstützt die Benutzung der Funktionen des Fehlererholungsmechanismus durch NFS-Anfragen, denen vordefinierte Parameter übergeben werden. Die entsprechend konstruierten NFS-Anfragen werden bei dem Aufruf des Programms zur Erzeugung neuer Verzeichnisse (*mkdir*) mit Argumenten *@save* oder *@restore* generiert:

- Der Aufruf mkdir @save speichert das TFS-Dateisystem, in dem versucht wird, das "Verzeichnis" *@save* zu erzeugen.
- Mit dem Befehl mkdir @restore wird bei dem zuletzt gespeicherten Zustand des TFS-Dateisystems, auf das sich dieser Befehl bezieht, aufgesetzt.

4.4 Die NFS-Tycoon-Bibliothek

Die Implementierung des Tycoon-Systems ist mit einer Sammlung von Funktionen ausgestattet, die den Austausch von Datenobjekten zwischen Tycoon und TFS-Dateisystem innerhalb eines Objektspeichers ermöglichen. Der Anhang C beinhaltet die Beschreibung der Schnittstelle der Tycoon-Bibliothek, in der die folgenden Funktionen implementiert sind:

- Zum Aufbau von TFS-Dateisystemen in Objektspeichern dient die Funktion *createFileSystem()*. Sie erzeugt und initialisiert die für den Aufbau nötigen TFS-Strukturen.
- Mit der Funktion *removeFileSystem()* kann ein vorhandenes, leeres TFS-Dateisystem aus dem Objektspeicher entfernt werden.
- Die Funktion *newFile()* dient der Integration der Tycoon-Datenobjekte in TFS-Strukturen. Mit ihr werden Datenobjekte des Tycoon-Systems als benannte Dateien in die Hierarchie des TFS-Dateisystems aufgenommen, so daß sie unter Verwendung der NFS-Prozeduren manipuliert werden können.

- Die *getFile()*-Funktion stellt Daten einer TFS-Datei im Tycoon-System zur Verfügung. Die Dateidaten können somit als Tycoon-Objektdaten behandelt werden.

Im Folgenden wird ein Beispiel für die Benutzung der oben aufgeführten Funktionen gegeben:

```
import tfs;
import image;

tfs.createFileSystem();
tfs.newFile(image.snapshot() "/picture" 0666);
image.display(image.new(tfs.getFile("/picture")) true);
```

Die Operation *createFileSystem()* erzeugt ein TFS-Dateisystem. Ein mit Hilfe von *snapshot()* erfaßtes Bild kann dann mit der Funktion *tfs.newFile()* in diesem Dateisystem abgelegt werden (im Verzeichnis "/" unter dem Namen "picture"). Ab diesem Zeitpunkt stehen die Bilddaten auch den UNIX-Anwendungen zur Verfügung. Die Operation *getFile()* ermöglicht die Anzeige des in der Datei "/picture" gespeicherten Bildes.

5. Grundlagen des entfernten Prozeduraufrufs

Der *entfernte Prozeduraufruf* (eng. *Remote Procedure Call*, kurz *RPC* [Corb91]) stellt die eigentliche Kommunikationskomponente zwischen dem NFS-Klientenprozeß und dem TFS-Serverprozeß dar. Seine Aufgabe ist es, die notwendige Kommunikation vor dem Anwendungsprogramm zu verbergen, so daß dieses sich idealerweise nicht von einer lokalen Variante unterscheidet. Dieses Ziel kann in der Regel nur teilweise erreicht werden. Der bei Sun verwendete Mechanismus des RPC ist zwar so gestaltet, daß das Anwendungsprogramm diesen transparent ausführt (d.h. im Programmcode ist er von einem lokalen Prozeduraufruf nicht zu unterscheiden), allerdings müssen vorher die Schnittstellen zu den Prozeduren, welche entfernt ausgeführt werden sollen, explizit spezifiziert werden. Dazu wird an die Programmiersprache "C" angelehnte Schnittstellenbeschreibungssprache "RPC Language" verwendet. In der Regel ist auch zusätzlicher Programmkode für den für die Kommunikation notwendigen Verbindungsaufbau und -abbau notwendig.

5.1 Konzepte der Klient/Server-Kommunikation

Den schematischen Ablauf eines entfernten Prozeduraufrufs stellt die Abbildung 5.1 dar. Der Klient ruft einen von externen Generatoren erzeugten *Klient-Stub* auf. Dieser übernimmt das Verpacken (*Marshalling*) der Argumente und ruft Funktionen aus der jeweiligen RPC-Laufzeitbibliothek auf, die dann die eigentliche Datenübertragung mittels des TCP/IP-Protokolls [Sun1] vornehmen. Auf der Seite des Server geschieht das gleiche in umgekehrter Reihenfolge. Die Funktionen der RPC-Laufzeitbibliothek rufen den zum aktuellen entfernten Aufruf gehörigen *Server-Stub* auf. Dieser ist ebenfalls von externen Generatoren aus einer Schnittstellenbeschreibung erzeugt worden und hat die Aufgabe, die ankommenden Daten in Argumente für die gewünschte Prozedur zu transformieren. Danach wird durch den Stub die Prozedur der Server-Applikation aufgerufen, die der Klient angefordert hat. Das Ergebnis der Prozedur wird vom Server-Stub wieder verpackt und über die RPC-Laufzeitbibliothek an den Klienten zurückgesendet. Dessen Stub entpackt die empfangenen Daten und liefert das so gewonnene Ergebnis als Ergebnis einer lokalen Prozedur zurück. Da alles unterhalb

der Prozedur-Schnittstelle geschieht, ist der entfernte Aufruf für die Klienten- und Server-Applikation transparent.

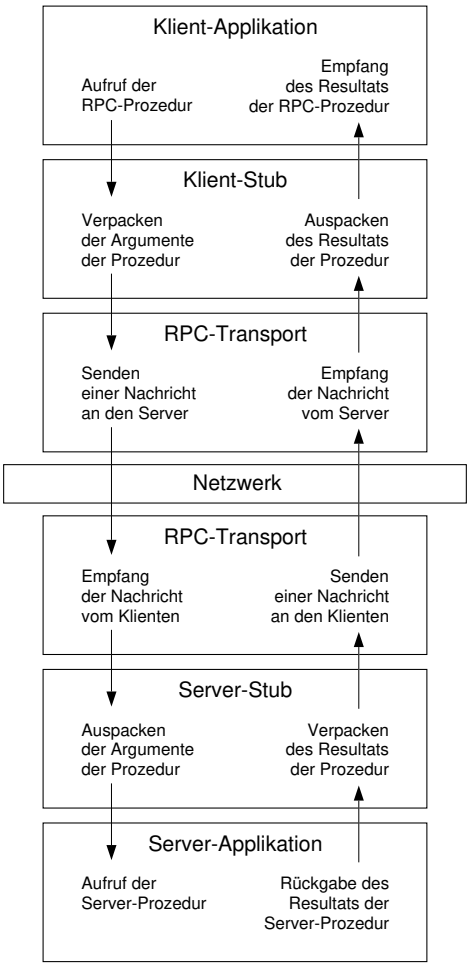

Abbildung 5.1: RPC-Architektur

Klient- und Server-Stub müssen notwendigerweise aus einer identischen *Schnittstellenbeschreibung* gewonnen werden. Der Sun-RPC bietet dafür eine gesonderte *Beschreibungssprache* an. Aus den mit dieser "C"-ähnlichen Sprache spezifizierten Beschreibungen werden mit Hilfe eines Generators (*rpcgen*) die Stubs erzeugt.

RPC-Server können nicht mehrere Anfragen gleichzeitig bearbeiten. Zur Leistungssteigerung können zwar mehrere Kopien des gleichen RPC-Servers gestartet werden, jede Instanz des Servers behandelt aber nach wie vor nur jeweils eine Anfrage. Ein RPC-Server kann viele Klienten haben. Ihre Anfragen werden jedoch in einer Warteschlange eingereiht und in der

Reihenfolge des Eintreffens bearbeitet.

RPC-Dienste werden über eine *Dienstnummer* (*Programmnummer*) angesprochen. Die Liste der Server und ihre Dienstnummern werden in einer speziellen Betriebssystemdatei verwaltet. Jedes Programm kann mehrere Prozeduren enthalten (18 im Fall von NFS). Um die zugrunde liegenden Protokolle zu erfüllen, müssen RPC-Dienste TCP/UDP-Portnummern benutzen. Die Abbildung von RPC-Programmnummern auf Portnummern wird von einem speziellen Dämon durchgeführt (*Portmapper*, siehe Abschnitt 5.2).

5.2 Kommunikationsphasen

Die Kommunikation zwischen zwei Partnern im Klient/Server-Modell basiert generell auf einer Reihe gesonderter Phasen oder Aufgaben.

Server-Identifikation

Eine Applikation, die Dienste einer anderen Applikation nutzen will, muß im Netz mögliche Diensterbringer lokalisieren können und feststellen, ob die benötigte Funktionalität überhaupt bereitgestellt wird. Server-Komponenten müssen folglich eindeutig benannt werden können. Dies geschieht in der Regel über einen eindeutigen Namen oder über Nummern. Für die netzweite Verwaltung solcher Identifikatoren beim Sun-RPC sorgt der *Portmapper*. Der Portmapper [Sun5] ist ein Prozeß, der auf jedem Netzknoten gestartet werden muß und der die Verbindung zwischen der Applikation und dem zugrundeliegenden Netz herstellt. Jeder Dienst muß eine ihn eindeutig bezeichnende Nummer verwenden. Diese Nummer sollte von einer übergeordneten Instanz vergeben werden, da ansonsten Konflikte auftreten können. Für Applikationen, die weltweit in den unterschiedlichsten Netzen verwendet werden sollen, übernimmt Sun Microsystems selbst die Vergabe von eindeutigen Identifikatoren. Beispielsweise besitzt der NFS-Dienst die weltweit eindeutige Nummer 100003 [Sun2].

Wenn sich ein RPC-Server initialisiert, registriert er seinen Dienst beim Portmapper. Der RPC-Server teilt dem Portmapper mit, welchen Port er auf eingehende Anfragen hin beobachtet und welches Netzprotokoll (TCP oder UDP) er für die Übertragung der Daten verwendet. Ein RPC-Klient nimmt direkt Verbindung mit dem RPC-Server auf, wenn ihm seine Portnummer bekannt ist. Er kann aber auch den Portmapper-Dämon benutzen, um die Portnummer des Server zu ermitteln.

Dienstspezifikationen werden nicht vom Portmapper verwaltet. Jeder Klient muß die Dienst-Schnittstelle seiner Server kennen und einhalten.

Binden

Bevor ein Klient einen Dienst nutzen kann, muß er sich an einen Server binden. Durch diesen Vorgang wird eine (virtuelle) Verbindung zum gewählten Diensterbringer aufgebaut und der Klient erhält nach erfolgreicher Überprüfung einen *Verbindungsidentifikator*, der für die weitere Kommunikation verwendet werden muß.

Marshalling

In heterogenen Netzen müssen die zu übertragenden Daten kodiert versandt werden, so daß ihre Bedeutung unabhängig von der Architektur der beteiligten Knotenrechner erhalten bleibt. Dies ist notwendig, da auf unterschiedlichen Plattformen unterschiedliche Datenrepräsentationen verwendet werden. Um trotzdem einen korrekten Datenaustausch gewährleisten zu können, werden die eigentlichen Daten vor ihrem Versand verpackt (*Marshalling*). Beim Sun-RPC geschieht das nach dem XDR-Standard (*External Data Representation* [Sun6]). Dabei werden die Daten in ein kanonisches Format umgewandelt. Die kanonische Datenrepräsentation muß nach dem Empfang in das Format des jeweiligen Rechners transformiert werden.

Datentransfer

Das Kernstück jeder Kommunikation ist der Transfer der Daten von einem Sender zu einem oder mehreren Empfängern. Hierbei muß die Synchronisation der beteiligten Komponenten und die Einhaltung der vom verwendeten Protokoll zugesicherten Eigenschaften der Übertragung gewährleistet werden. Diese Zusicherungen werden von den zugrundeliegenden Netzen und ihren Protokollkomponenten, beispielsweise TCP- und UDP-Protokolle, erbracht.

Dispatching

Damit ein Server die ankommenden Anfragen zu einem Dienst bearbeiten kann, muß er feststellen, ob und welche Dienstfunktionen ausgeführt werden sollen. Um dies zu erreichen, kann der Server aktiv in einer Schleife nachfragen, ob ein Auftrag für ihn anliegt. Dieses Verfahren (*Polling*) führt aber zu einer unnötig hohen Systembelastung, da der Serverprozeß, auch ohne eingehende Aufträge zu bearbeiten, immer aktiv sein muß. Eine Möglichkeit, dies zu vermeiden, ist das passive Warten auf Systemunterbrechungen (*Interrupts*). Hier kann ein Serverprozeß direkt vom Betriebssystem Signale anfordern, auf deren Eintreten der Prozeß dann passiv wartet. In der Wartezeit verbraucht der Serverprozeß keine Prozessorzeit, da er vom Betriebssystem bis zum Eintreten des Signals eingefroren wird.

6. Administration von TFS

Das Einrichten von TFS-Klienten und -Servern erfordet das Starten von Prozessen, die das RPC-Protokoll für TFS behandeln und die TFS-Funktionalitäten realisieren. Darüber hinaus müssen die von TFS-Servern exportierten Dateisysteme von den TFS-Klienten importiert werden.

6.1 Einrichten eines Tycoon-Dateisystems

Das Erzeugen von TFS-Dateisystemen erfolgt unter Verwendung des Tycoon-Systems. Die nötigen TFS-Strukturen werden zur Laufzeit des Systems im Objektspeicher der aktuellen Tycoon-Sitzung aufgebaut.

Um die so erzeugten TFS-Dateisysteme mit Standard-Anwendungen nutzen zu können, muß zunächst der *TFS-Dämon* gestartet werden. Auf einem NFS-Server werden danach die NFS- und Mount-Dienste bereitgestellt. Der *NFS-Dämon* behandelt die von Klienten kommenden NFS-Anfragen. Der *Mount-Dämon* behandelt Anfragen von Klienten zum Importieren von Dateisystemen. Beide leiten die Anfragen an den TFS-Dämon weiter. Der TFS-Dämon nimmt RPC-Anfragen für NFS entgegen und führt sie aus. Anders als bei einem UNIX-NFS-Server wird bei TFS jeweils nur eine Instanz der Dämonen gestartet, da einige dem TFS zugrundeliegenden Objektspeichersysteme nur im Einbenutzermodus benutzt werden können.

6.2 Exportieren und Importieren von Dateisystemen

Der TFS-Server verwaltet eine Liste der aktuell exportierten Dateisysteme und der zugehörigen Zugriffsbeschränkungen. Die Einträge in der Liste werden bei jedem Erzeugen neuer oder Löschen nicht mehr benötigter TFS-Dateisysteme aktualisiert.

Jedes exportierte TFS-Dateisystem kann von Klienten importiert werden. Da TFS-Dateisysteme auf dem Klienten in jedem Verzeichnis importiert werden können, ist es möglich, Teile von existierenden Dateibäumen zu verdecken.

Das TFS-Dateisystem kann in der speziellen Betriebssystemdatei ("/etc/fstab") des Klienten aufgeführt sein oder explizit mit dem Kommando mount importiert werden.

Benutzung der Datei "/etc/fstab"

Wurde ein Eintrag in der Datei fstab eingefügt, so importieren die TFS-Klienten das betreffende Dateisystem jedesmal beim Systemstart. Die Einträge unterscheiden sich nicht von den "normalen" NFS-Einträgen. Ihre Syntax lautet:

Server:Dateisystem Verzeichnis Typ <Optionen> 0 0

- *Server* bezeichnet den Namen des NFS-Servers.
- *Dateisystem* ist ein Pfadname der Objektspeicherdatei in dem TFS-Dateisystem. Dies muß ein absoluter Pfadname sein. Im Gegensatz zu Sun-NFS ist es nicht möglich, ein Unterverzeichnis eines vom TFS-Server exportierten Dateisystems zu importieren.
- *Verzeichnis* gibt an, in welchem Verzeichnis das TFS-Dateisystem importiert werden soll (*Mount-Point*).
- Als *Typ* des Dateisystems muß *nfs* angegeben werden.
- Das Optionsfeld kann eine Vielzahl von NFS-spezifischen Mount-Optionen enthalten. Die Optionswerte werden im nächsten Abschnitt erklärt.

Ein Beispieleintrag für das TFS-Dateisystem *tfs.ts* sieht wie folgend aus:

dbis12:/users/dbis1/gorzynsk/tfs.ts /users/tfs nfs rw,bg,soft,retrans=6,timeo=11 0 0

Benutzung von mount

Einträge in der Datei fstab sind nützlich, um eine langlebige NFS-Umgebung herzustellen. Manchmal muß jedoch ein Dateisystem während der laufenden Arbeit importiert werden oder es wird nur kurzfristig benötigt, um Dateien daraus zu kopieren. Mit dem Kommando mount kann man ein TFS-Dateisystem so importieren, daß es nur bis zum expliziten Entfernen mittels unmount oder bis zum Abschalten der Maschine importiert bleibt. Das Kommando mount erwartet in der Kommandozeile den Namen des Servers und des Dateisystems in einer Syntax ähnlich der von fstab-Einträgen. Mit dem nachfolgenden Befehl kann das Dateisystem aus dem Objektspeicher *tfs.ts* dynamisch importiert werden:

mount dbis12:/users/dbis1/gorzynsk/tfs.ts /users/tfs nfs -o rw,hard 0 0

Für das Importieren von TFS-Dateisystemen können folgende Optionen spezifiziert werden:

rw/ro: Diese Option beschreibt, ob ein Dateisystem zum Lesen und Schreiben (*rw*) oder nur zum Lesen importiert wird (*ro*). Die Option *rw* ist die Standardeinstellung.

retry: Kann ein Klient ein TFS-Dateisystem nicht innerhalb der vorgesehenen RPC-Ausführungszeit importieren, wiederholt er die RPC-Operation so oft, wie es bei der Option *retry* angegeben wurde.

bg: Wurde die Option *bg* angegeben, so startet mount einen neuen Prozeß, der das Dateisystem im Hintergrund importiert. Ohne diese Option blockiert mount und wartet so lange, bis der Dateiserver antwortet oder die maximale Anzahl der Wiederholungen erreicht ist (siehe *retry*-Option).

hard/soft: Die Option *hard* bewirkt, daß NFS-Anfragen so lange wiederholt werden, bis sie vom Server bestätigt werden. Wurde die Option *soft* angegeben, so liefert ein NFS-RPC-Aufruf einen Zeitüberschreitungsfehler, falls die bei *retrans* angegebene Anzahl überschritten wird. Die Datenkonsistenz nach einem Systemfehler kann nur dann garantiert werden, wenn das TFS-Dateisystem mit der Option *hard* importiert wurde.

retrans: Die Option *retrans* wird im Zusammenhang mit der Option *soft* verwendet. Sie spezifiziert, wie oft eine RPC-Anforderung wiederholt wird, bevor ein Zeitüberschreitungsfehler zurückgeliefert wird.

timeo: Mit dem Parameter *timeo* wird die Zeitdauer für eine RPC-Zeitüberschreitung (*timeout*) in Zehntelsekunden angegeben.

intr: NFS-Operationen sind nicht unterbrechbar und laufen so lange weiter, bis ein RPC-Fehler auftritt oder sie erfolgreich abgeschlossen sind. Mit der Option *intr* kann der Benutzer NFS-RPC-Aufrufe unterbrechen und die RPC-Schicht zwingen, einen Fehler zurückzumelden.

port: Die Option *port* legt fest, an welchen Port des Dateiservers die NFS-Anfragen verschickt werden. Da der NFS-Dämon die Standard-Port-Nummer verwendet, entfällt die Angabe dieser Option bei der Verwendung der aktuellen Version des TFS. Sie kann aber nützlich sein, wenn auf den Einsatz des die NFS-Anfragen weiterleitenden NFS-Dämons verzichtet werden sollte.

rsize und wsize: Die Optionen *rsize* und *wsize* dienen der Festlegung der Größe der Schreib- und Lesepuffer.

7. Zusammenfassung und Ausblick

Das in dieser Arbeit vorgestellte *Tycoon File System* ist mit dem Ziel entworfen worden, entfernte Zugriffe auf Tycoon-Objektspeicher zu ermöglichen. Das TFS ist als eine Dateischnittstelle zu persistenten Objektspeichern unter Verwendung des *Network File Systems* und des *Tycoon Store Protocols* realisiert. Es erlaubt den Zugriff auf die in Objektspeichern gespeicherten Daten und ihre Manipulation mit Hilfe von Standard UNIX-Befehlen. Somit können existierenden Anwendungen wie Editoren, Filter, WWW-Browser, Audio-/Video- und Suchwerkzeuge direkt auf Objektspeicherobjekte zugreifen. Über die Tycoon-Schnittstelle können Tycoon-Programme auf diese in speziellen TFS-Strukturen gespeicherten Objekte als Zeichenketten zugreifen oder neue Objekte in die TFS-Dateisystemhierarchie einfügen.

Die in dieser Arbeit vorgestellte Implementation von TFS ist noch hinsichtlich der folgenden Aspekte erweiterbar:

Sicherheit von NFS: Die Sicherung eines Dateisystems besteht aus zwei Teilen: Kontrolle des Zugangs zu den Dateien und der Operationen auf ihnen sowie begrenzte Freigabe der Dateiinhalte. Das TFS-Gateway beschränkt sich auf die Überprüfung der Rechte der Klienten-Rechner beim Importieren von TFS-Dateisystemen.

Übermittlung der NFS- und Mount-Anfragen: In der vorliegenden Version von TFS werden die NFS- und Mount-Anfragen mittels der NFS- und Mount-Dämonen an den TFS-Server zur Abarbeitung weitergegeben. Es ist denkbar, auf die Dienste des NFS-Dämonen zu verzichten. Dafür müßte aber der Mount-Dämon dahingehend erweitert werden, außer den Mount-Anfragen für das TFS-Dateisystem auch die Anfragen bezüglich anderer Dateisysteme bearbeiten zu können.

Strukturierte Dateidaten: Zur Zeit können zwischen Tycoon und TFS nur unstrukturierte Daten in Form von Zeichenketten (Tycoon-Typ *String*) ausgetauscht werden. Diese Tatsache trägt zur Einfachheit der die Daten transformierenden Komponente und der Lese- und Schreiboperationen des TFS-Gateways bei. Allerdings sollten auch strukturierte Daten in das TFS-Dateisystem integriert oder von dem Tycoon-System übernommen werden können.

Funktionen der *tfsenv*-Bibliothek: Mit vorhandenen Tycoon-Funktionen lassen sich TFS-Dateisysteme erzeugen oder löschen und Datenobjekte zwischen Tycoon und TFS austauschen. Die Bibliothek könnte um Funktionen zur Manipulation und Verwaltung der TFS-Dateibäume ergänzt werden. Durch die Erweiterung um das Konzept des aktuellen Verzeichnisses könnten zum Beispiel die Pfade auf Dateisystemobjekte in den Tycoon-Funktionen relativ zum aktuellen Verzeichnis angegeben werden.

Gemeinsame Nutzung der Datenobjekte: Der Austausch von Daten zwischen Tycoon und TFS hat Referenzsemantik. Die Manipulationsoperationen auf den Datenobjekten können aber dazu führen, daß Kopien von Objekten entstehen, die unabhängig voneinander verwaltet werden. Das Ziel wäre, keine Kopien entstehen zu lassen, damit die im Tycoon-System an einem Datenobjekt vorgenommenen Änderungen im TFS-Dateisystem sichtbar sind. Auch beim Editieren von TFS-Dateisystemobjekten sollte die Konsistenz von Daten gewahrt bleiben.

Gleichzeitiger Zugriff auf Objektspeicher: Die Möglichkeit der gleichzeitigen Nutzung der Objektspeicher durch Tycoon und TFS und damit verbundene Probleme müßten untersucht werden.

A. Schnittstellen des NFS- und Mount-Protokolls

Dieser Anhang liefert eine Übersicht aller NFS- und Mount-Prozeduren. Die hier bei der Festlegung der Prozedurparameter verwendeten Datentypen werden im Anhang B definiert.

A.1 Schnittstellenbeschreibung des NFS-Protokolls

```
program NFS_PROGRAM {
    version NFS_VERSION {
        void NFSPROC_NULL(void) = 0;
        /* does no work */
        attrstat NFS_PROC_GETATTR(nfs_fh) = 1;
        /* gets file attributes */
        attrstat NFS_PROC_SETATTR(sattrargs) = 2;
        /* sets file attributes */
        void NFS_PROC_ROOT(void) = 3;
        /* gets filesystem root, no longer used */
        diropres NFS_PROC_LOOKUP(diropargs) = 4;
        /* looks up file name */
        readlinkres NFS_PROC_READLINK(nfs_fh) = 5;
        /* reads from symbolic link */
        readres NFS_PROC_READ(readargs) = 6;
        /* reads from file */
        void NFS_PROC_WRITECACHE(void) = 7;
        /* writes to cach, to be used in the next protocol version */
        attrstat NFS_PROC_WRITE(writeargs) = 8;
        /* writes to file */
        diropres NFS_PROC_CREATE(createargs) = 9;
        /* creates file */
        nfsstat NFS_PROC_REMOVE(diropargs) = 10;
        /* removes file */
```

```
        nfsstat NFS_PROC_RENAME(renameargs) = 11;
        /* renames file */
        nfsstat NFS_PROC_LINK(linkargs) = 12;
        /* creates link to file */
        nfsstat NFS_PROC_SYMLINK(symlinkargs) = 13;
        /* creates symbolic link */
        diropres NFS_PROC_MKDIR(createargs) = 14;
        /* creates directory */
        nfsstat NFS_PROC_RMDIR(dirpargs) = 15;
        /* removes directory */
        readdirres NFS_PROC_READDIR(readdirargs) = 16;
        /* reads from directory */
        statfsres NFS_PROC_STATFS(nfs_fh) = 17;
        /* gets filesystem attributes */
    } = 2;
} = 100003;
```

A.2 Schnittstellenbeschreibung des Mount-Protokolls

```
program MOUNT_PROGRAM {
    version MOUNT_VERSION {
        void MOUNT_PROC_NULL(void) = 0;
        /* does no work */
        fhstatus MOUNT_PROC_MNT(dirpath) = 1;
        /* looks up the root file handle, adds mount entry */
        mountlist MOUNT_PROC_DUMP(void) = 2;
        /* returns mount entries */
        void MOUNT_PROC_UMNT(dirpath) = 3;
        /* removes mount entry */
        void MOUNT_PROC_UMNTALL(void) = 4;
        /* removes all mount entries */
        exports MOUNT_PROC_EXPORT(void) = 5;
        /* returns export list */
        exports MOUNT_PROC_EXPORTALL(void) = 6;
        /* returns export list */
    } = 1;
} = 0x20000005;
```

B. NFS- und Mount-Datenstrukturen

Die folgenden Datenstrukturen werden als Argumente und Resultate für die NFS- und Mount-Prozeduren verwendet (siehe Anhang A):

```
enum nfsstat {
    NFS_OK = 0,
    NFSERR_PERM = 1,
    NFSERR_NOENT = 2,
    NFSERR_IO = 5,
    NFSERR_NXIO = 6,
    NFSERR_ACCES = 13,
    NFSERR_EXIST = 17,
    NFSERR_NODEV = 19,
    NFSERR_NOTDIR = 20,
    NFSERR_ISDIR = 21,
    NFSERR_FBIG = 27,
    NFSERR_NOSPC = 28,
    NFSERR_ROFS = 30,
    NFSERR_NAMETOOLONG = 63,
    NFSERR_NOTEMPTY = 66,
    NFSERR_DQUOT = 69,
    NFSERR_STALE = 70,
    NFSERR_WFLUSH = 99
};
typedef enum nfsstat nfsstat;

enum ftype {
    NFNON = 0,
    NFREG = 1,
    NFDIR = 2,
    NFBLK = 3,
```

```
    NFCHR = 4,
    NFLNK = 5,
    NFSOCK = 6,
    NFBAD = 7,
    NFFIFO = 8
};
typedef enum ftype ftype;

typedef char fhandle[FHSIZE];

struct nfstime {
    unsigned int seconds;
    unsigned int useconds;
};
typedef struct nfstime nfstime;

struct fattr {
    ftype type;
    unsigned int mode;
    unsigned int nlink;
    unsigned int uid;
    unsigned int gid;
    unsigned int size;
    unsigned int blocksize;
    unsigned int rdev;
    unsigned int blocks;
    unsigned int fsid;
    unsigned int fileid;
    nfstime atime;
    nfstime mtime;
    nfstime ctime;
};
typedef struct fattr fattr;

struct sattr {
    unsigned int mode;
    unsigned int uid;
    unsigned int gid;
    unsigned int size;
    nfstime atime;
    nfstime mtime;
};
```

```
typedef struct sattr sattr;

struct attrstat {
     nfsstat status;
     union {
          fattr attributes;
     } attrstat_u;
};
typedef struct attrstat attrstat;

struct sattrargs {
     fhandle file;
     sattr attributes;
};
typedef struct sattrargs sattrargs;

struct diropargs {
     fhandle dir;
     char *filename;
};
typedef struct diropargs diropargs;

struct diropokres {
     fhandle file;
     fattr attributes;
};
typedef struct diropokres diropokres;

struct diropres {
     nfsstat status;
     union {
          diropokres diropres;
     } diropres_u;
};
typedef struct diropres diropres;

struct readlinkres {
     nfsstat status;
     union {
          char *data;
     } readlinkres_u;
};
```

```
typedef struct readlinkres readlinkres;

struct readargs {
    fhandle file;
    unsigned int offset;
    unsigned int count;
    unsigned int totalcount;
};
typedef struct readargs readargs;

struct readokres {
    fattr attributes;
    struct {
        unsigned int data_len;
        char *data_val;
    } data;
};
typedef struct readokres readokres;

struct readres {
    nfsstat status;
    union {
        readokres reply;
    } readres_u;
};
typedef struct readres readres;

struct writeargs {
    fhandle file;
    unsigned int beginoffset;
    unsigned int offset;
    unsigned int totalcount;
    struct {
        unsigned int data_len;
        char *data_val;
    } data;
};
typedef struct writeargs writeargs;

struct createargs {
    diropargs where;
    sattr attributes;
```

```
};
typedef struct createargs createargs;

struct renameargs {
    diropargs from;
    diropargs to;
};
typedef struct renameargs renameargs;

struct linkargs {
    fhandle from;
    diropargs to;
};
typedef struct linkargs linkargs;

struct symlinkargs {
    diropargs from;
    char *to;
    sattr attributes;
};
typedef struct symlinkargs symlinkargs;

typedef char nfscookie[NFS_COOKIESIZE];

struct readdirargs {
    fhandle dir;
    nfscookie cookie;
    unsigned int count;
};
typedef struct readdirargs readdirargs;

struct entry {
    unsigned int fileid;
    char *filename;
    nfscookie cookie;
    struct entry *nextentry;
};
typedef struct entry entry;

struct dirlist {
    entry *entries;
    bool_t eof;
```

```
};
typedef struct dirlist dirlist;

struct readdirres {
    nfsstat status;
    union {
        dirlist reply;
    } readdirres_u;
};
typedef struct readdirres readdirres;

struct statfsokres {
    unsigned int tsize;
    unsigned int bsize;
    unsigned int blocks;
    unsigned int bfree;
    unsigned int bavail;
};
typedef struct statfsokres statfsokres;

struct statfsres {
    nfsstat status;
    union {
        statfsokres reply;
    } statfsres_u;
};
typedef struct statfsres statfsres;

struct fhstatus {
    nfsstat fhs_status;
    union {
        fhandle fhs_fhandle;
    } fhstatus_u;
};
typedef struct fhstatus fhstatus;

struct mountbody {
    char *ml_hostname;
    char *ml_directory;
    mountbody *ml_next;
};
typedef struct mountbody mountbody;
```

```
struct groupnode {
    char *gr_name;
    groupnode *gr_next;
};
typedef struct groupnode groupnode;

struct exportnode {
    char *ex_dir;
    groupnode *ex_groups;
    exportnode *ex_next;
};
typedef struct exportnode exportnode;
```

C. Schnittstelle der Tycoon-Bibliothek *tfsenv*

```
interface TFS
    (*
    File:      TFS.ti
    Author:    Lukas Gorzynski
    Date:      04.10.1995
    Purpose: Interface to Tycoon File System
    *)

export

    createFileSystem() :Ok
    (* create a file system in the current store *)

    removeFileSystem() :Ok
    (* remove a file system from the current store *)

    newFile(fileName    :String
            directory   :String
            data        :String
            permission  :Int    ) :Ok
    (* create a directory entry "fileName" for "data" in "directory",
       the "permission" for the created file must be specified *)

    getFile(filePath :String) :String
    (* get data of the file with "filePath",
       "filePath" must be absolute *)

end;
```

Literaturverzeichnis

Muel91 R. Müller. *Sprachprozessoren und Objektspeicher: Schnittstellenentwurf und Implementierung.* Diplomarbeit, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, November 1991.

Matt93 F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993.

MMM93 B. Mathiske, F. Matthes, and S. Müßig. *The Tycoon System and Library Manual.* DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Dezember 1993.

MMS92 F. Matthes, R. Müller, and J.W. Schmidt. *Object Stores as Servers in Persistent Programming Environments - The P-Quest Experience.* FIDE Technical Report FIDE/92/48, Fachbereich Informatik, Universität Hamburg, Juli 1992.

MMS95 F. Matthes, R. Müller, and J.W. Schmidt. *Towards a Unified Model of Untyped Object Stores: Experience with the Tycoon Store Protocol.* To appear in Fully Integrated Data Environments, Springer-Verlag, 1995.

MaSc93 F. Matthes and J.W. Schmidt. *System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways.* In Proceedings of Euro-Arch'93 Congress. Springer-Verlag, Oktober 1993.

Ster93 H. Stern. *NFS und NIS.* Addison-Wesley Publishing Company, 1993.

Sun2 Sun Microsystems, Inc. *Network Interfaces Programmer's Guide.* 1994.

Post80 J. Postel. *User Datagram Protocol.* RFC 768, USC Information Sciences Institute, September 1981.

Sand85 R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. *Design and Implementation of the Sun Network File System.* USENIX Proceedings, 119-130, June 1985.

StAh87 M. Stein, and S. Ahnger. *Tutorial #M5: The Network File System.* USENIX Technical Conference and Exhibition, Juni 1987.

Sun4 Sun Microsystems, Inc. *NFS: Network File System Protocol Specification.* RFC 1094, March 1989.

Sun5 Sun Microsystems, Inc. *RPC: Remote Procedure Call Protocol Specification.* RFC 1057, June 1988.

Sun6 Sun Microsystems, Inc. *XDR: External Data Representation Standard.* RFC 1014, June 1987.

Corb91 J.R. Corbin. *The Art of Distributed Applications.* Springer-Verlag, 1991.

Bach86 M.J. Bach. *The Design of the Unix Operating System.* Prentice-Hall International, Inc., 1986.

Tan92 A.S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall International, Inc., 1992.

PeSi88 J.L. Peterson, and A. Silberschatz. *Operating System Concepts.* Addison-Wesley Publishing Company, 1988.

Sun1 Sun Microsystems, Inc. *TCP/IP Network Administration Guide.* 1994.