Zusammenfassung „Verteilte Systeme 2" WS1995/1996

Michael Jaeger (michael.jaeger@in-flux.de)

29. Oktober 2001

Inhaltsverzeichnis

1 Einleitung **2**

2 Verteilte Dateisysteme **4**

2.1 Einleitung 4

2.1.1 Beispiel: Dateisystem unter UNIX 5

2.2 Entwurfsfragen 5

2.2.1 Namen und Transparenz 6

2.2.2 Zugriffssemantik 6

2.2.3 Caching (Performance) 6

2.2.4 Zuverlässigkeit 8

2.2.5 Replikation 8

2.2.6 Skalierbarkeit 9

2.3 Sun NFS 10

2.4 Andrew File System (AFS) 11

2.5 Vergleich NFS vs. AFS 13

2.6 Zusammenfassung 14

3 Verteilte Transaktionen **14**
3.1 Grundlagen 14
3.2 Verteilte Ausführung 18
3.3 DCE / Encina 19
3.4 Fehlerbehandlung 19
3.5 Sperren 22

4 Management **23**
4.1 Grundlagen 23
4.2 OSI Management 25
4.2.1 Informationsmodell 27
4.2.2 Organisationsmodell 31
4.2.3 Kommunikationsmodell 31
4.2.4 Funktionsmodell 33
4.3 Internet Management 34
4.3.1 Organisationsmodell 34
4.3.2 Informationsmodell 35
4.3.3 Kommunikationsmodell 37
4.3.4 SNMPv2 39
4.4 Vergleich OSI vs. Internet 39
4.5 ITU Telecommunications Management Network (TMN) 40
4.5.1 Informations-Architektur 44
4.5.2 Service Management 44

5 Prüfungsfragen **46**

1 Einleitung

Der Stand der Technik entwickelte sich vom Batch-Betrieb über das Time Sharing hin zum Client/Server-Computing. Die Auswirkungen auf die betriebliche Informationstechnik sind Reorganisation (**Downsizing**, **Upsizing**, **Rightsizing**).

Das **Client/Server-Modell** stellt hierbei ein **abstraktes Kooperationsmodell** für Programm-zu-Programm-Interaktion dar, wobei sowohl Client als auch Server

Hardware-Komponenten sind. Gleichzeitig stellt es ein **Organisationsmodell für moderne Informationsverarbeitung** dar. Im **Strukturmodell** für Client/Server-Anwendungen werden verschiedene **Schnitte** (**Tiers**) unterschieden, wie z.B. User Interface, Function und Data. Die Anzahl der Schnitte durch eine Anwendung ist dabei beliebig – Entwicklungswerkzeuge sollen hier Transparenz schaffen.

Aus der Diskussion von Client/Server-Systemen aus Verteilte Systeme 1 sind folgende Punkte erhalten:

Vorteile	Nachteile
Flexibilität und Skalierbarkeit	höhere Komplexität durch Verteilung und Heterogenität
'information at your fingertips'	Gesamtkosten nicht niedriger / schwer zu überschauen
attraktive Funktionalität	komplexe Netzinfrastrukturen
Ergonomie	unausgereifte Managementlösungen
niedrige Kosten für Hard- und Software	unausgereifte Sicherheitsmechanismen
Entkopplung durch Dezentralisierung	Trainingsaufwand
Entsprechung organisatorischer Strukturen	wichtige Anwendungen nicht verfügbar
	'legacy applications' (Integration von Altlasten)

Einen Schritt weiter gehen **Verteilungsplattformen** (**Middleware**), also **Software, die die Realisierung verteilter Anwendungen unterstützt**. Im Allgemeinen unterscheiden wir folgende Arten von Middleware:

Art	Beispiel
Message Queues (Send/Receive-Kommunikation)	MQS, Peerlogic,... (MOMA Standard)
Datei-Server, Dateitransfer (Fernzugriff auf gemeinsame Dateien)	Novell Netware, LAN Manager, Vines, NFS, ...
Remote Procedure Call (RPC) (Aufruf einer entfernten Prozedur)	OSF DCE, SUN ONC
Object Request Broker (ORB) (Objektaufrufe über das Netz)	OMG CORBA
Datenbankzugriff (auf entfernte Datenbanken)	ODBC, DRDA, SQLnet, ...
Transaktionsverarbeitung (Koordination verteilter Transaktionen)	Encina RPC, Tuxedo, ...
Groupware (Zusammenarbeit bei Gruppen)	Lotus Notes, ...

Art	Beispiel
Message Queues (Send/Receive-Kommunikation)	MQS, Peerlogic,... (MOMA Standard)
Datei-Server, Dateitransfer (Fernzugriff auf gemeinsame Dateien)	Novell Netware, LAN Manager, Vines, NFS, ...
Workflow (Organisation arbeitsteiliger Prozesse)	IBM Flowmark, ...

2 Verteilte Dateisysteme

2.1 Einleitung

Zunächst einige allgemeine Begriffsdefinitionen:

Datei (File) benamte Ansammlung von Daten auf permanentem Speichermedium.

Dateisystem (File System, LFS) Gesamtheit der Dateien mit definiertem Namensraum, Zugriffsmethoden, Speicherverfahren etc.

Verteiltes Dateisystem (Distributed File System, DFS) Dateisystem, das für den Benutzer wie ein konventionelles, nicht-verteiltes Dateisystem erscheint, dessen Dateien aber auf mehreren Knoten eines Rechnerverbundes liegen können.

Datei-Server (File Server) Knoten eines Rechnerverbundes, der Dateien verwaltet und Dateizugriffe über das Netz ermöglicht.

Dateisystem

Das **Dateisystem** eines Betriebssystems legt Folgendes fest:

- **Zugriffsschnittstelle** (z.B. System Calls, Funktionsbibliotheken)
- **Struktur des Dateiinhalts** (z.B. Bytefolge, Sätze fester Länge, Sätze variabler Länge)
- **Namen und Organisation des Namensraums** (z.B. flach, zweistufig, hierarchisch)
- **Zugriffsformen** (z.B. sequentiell, direkt, index-sequentiell)
- **Schutzvorrichtungen** (z.B. Zugriffslisten, Capabilities)
- **Abbildung der logischen Struktur auf die Hardware** (z.B. „Transparenz unterschiedlicher Plattenfabrikate”)

2.1.1 Beispiel: Dateisystem unter UNIX

In UNIX ist das Dateisystem z.B. wie folgt festgelegt:

- **Zugriffsschnittstelle**
  „Kernel Traps" sind als C-Funktion verpackt.
- **Struktur des Dateiinhalts**
  Der Namensraum ist hierarchisch, Namen können (fast) beliebige Länge besitzen, Verzeichnisse sind Knoten des Dateibaums, Dateien Blätter, es gibt relative und absolute Namen.
- **Namen und Organisation des Namensraums**
  Die Struktur des Dateiinhalts ist nicht festgelegt.
- **Zugriffsformen**
  Es kann direkt auf die Byte-Folgen in der Datei zugegriffen werden.
- **Schutzvorrichtungen**
  Einfache Zugriffslisten (user-group-all) bieten read-write-execute-Zugriff auf die Datei.

Die Verwaltung der Dateien erfolgt durch sog. **I-Nodes** (Index Nodes). Für jede Datei und jedes Verzeichnis gibt es einen i-node und beim Öffnen in den Hauptspeicher gelesen. Er enthält Angaben zu Dateiattributen und Speicheradressen der Datei auf der Platte sowie Dateiattribute (Rechte, Größe, letzter Zugriff, etc.). Der Dateiname selbst steht nur im Verzeichnis, nicht im i-node. Da viele Dateien sehr kurz sind, enthält der i-node einige direkte Blockadressen der Daten. Eine einfach/zweifach/dreifach indirekte Adressierung ermöglicht auch große Dateien.

2.2 Entwurfsfragen

Im Folgenden werden die Kern-Entwurfsfragen bei verteilten Dateisystemen verhandelt, die da sind:

- Namensraum und Transparenz
- Schnittstelle des Dateiservers
- Zugriffssemantik
- Performance
- Zuverlässigkeit und Verfügbarkeit
- Skalierbarkeit

2.2.1 Namen und Transparenz

Um eine Datei lokalisieren zu können, muss man sie eindeutig benennen können mit der **Adresse des Knoten** und der **Adresse im LFS**. Dabei sind **Transparenzforderungen** bzgl. folgenden Aspekten gefordert:

- **Ort** (Dateiname unabh. von „lokal" und „entfernt")
- **Zugriff** (Dateizugriff unabh. von „lokal" und „entfernt")
- **Replikation** (Anzahl der Replikate bleibt verborgen)
- **Migration** (Verlagerung der Datei bleibt verborgen)

Dabei existieren bereits 3 prinzipielle Arten der Integration von Namensräumen:

1. „host:filename" (keine Ortstransparenz)
2. netzweit eindeutiger Namensraum (Zugriff über immer gleichen Namen)
3. relativ zum lokalen Namensraum (durch „Mounten")

2.2.2 Zugriffssemantik

Der gleichzeitige Zugriff mehrerer Clients auf eine Datei kann zu **Zugriffskonflikten** und **Inkonsistenzen** führen. Dabei sind grundsätzlich zwei Arten von Zugriffssemantiken denkbar: die **UNIX-Semantik** (jede `read`-Operation sieht alle vorherigen `write`-Operationen) und die **Sitzungssemantik** (eine Sitzung ist eine Folge von `read`/`write`-Zugriffen, die erst amEnde der Sitzung für Andere sichtbar werden). Erstere ist in verteilten Systemen sehr schwer durchsetzbar und aufwendig zu realisieren, während es durchaus DFSe gibt, die sich mit letzterer Semantik zufrieden geben und den Anwendungen das Verwenden von Sperren etc. überlassen.

2.2.3 Caching (Performance)

Das **Caching** von Dateiinhalten erfolgt mit dem Ziel, den Zugriff zu beschleunigen und Kommunikationsvorgänge über das Netz möglichst zu vermeiden. Beim Einsatz von Caches stellen sich folgende Entwurfsfragen:

- In welcher **Granulartiät** werden Dateiinhalte im Cache gespeichert (ganze Datei, nur Dateiblöcke)?
- **Wann** werden Änderungen zum Server **zurückgeschrieben**?
- **Wie** werden Caches **konsistent** gehalten?

- Wie sieht die **Ersetzungsstrategie** bei Einlagerung neuer Daten aus?
- **Wo** befindet sich der Cache (Platte, Hauptspeicher)?

kschreiben des Cache

Soll der Inhalt des Cache **zurückgeschrieben** werden, so müssen alle Schreibvorgänge des Clients an den Server mit der Master-Kopie propagiert werden. Dies ist ein **Tradeoff zwischen Kommunikationsaufwand (Performance) und Aktualität der Master-Kopie**. Folgende Strategien werden im Allgemeinen eingesetzt:

- **write through**: Jedes Schreiben geht direkt an den Master.
  + Zuverlässigkeit, Aktualität
  − Overhead, oft unnötige Updates, read-only cache
- **delayed write**: Schreiben erfolgt auf dem Cache, späteres Update der Master-Kopie (evtl. periodisch)
  + schneller, vermeidet unnötige Updates
  − Problem bei Ausfällen des Clients
- **write on close**: Schreiben erfolgt auf dem Cache, Update der Master-Kopie beim Schließen der Datei
  + gut für Dateien, die lange geöffnet sind
  − Maste-Kopie nicht sehr aktuell

Beim Einsatz eines Cache müssen Clients die **Gültigkeit einer Cache-Kopie** überprüfen können, um zu entscheiden, ob der Cache noch aktuell ist. Prinzipiell gibt es hier zwei Vorgehensweisen:

- **Client-initiiert**: Der Client fragt nach dem Öffnen periodisch beim Server nach
  + kann nach Bedarf ausgeführt werde, z.B. bei jedem Zugriff
  − Performance
- **Server-initiiert**: Call back vom Server zu allen Clients, die eine Kopie der Datei im Cache haben (z.B. wenn eine Datei von einem Client zurückgeschrieben wird)
  + Server kann aktiv werden, wenn Client-Operationen im Konflikt stehen
  − komplexere Client/Server-Interaktion

Zusammenhang Zugriffssemantik und Caching-Strategie

Es bestehen folgende Zusammenhänge zwischen der Zugriffssemantik und der Caching-Strategie:

- **Sitzungssemantik+Caching ganzer Dateien+Write on close**
  Alle Operationen werden lokal im Cache durchgeführt und am Ende der Benutzung zurückgeschrieben

- **UNIX-Semantik+Caching+???**
  In einem DFS praktisch nicht zu erreichen, da alle Schreiboperationen sofort zum Master und allen Cache-Kopien propagiert werden müssen.
- **UNIX-Sematik+kein Caching**
  Wird häufig aus Gründen der Einfachheit und Kompatibilität vorgezogen.

2.2.4 Zuverlässigkeit

Bei Server-Design gibt es prinzipiell zwei Design-Arten:

- **zustandslos**: Jeder Zugriff enthält alle Informationen (z.B. File Handle, Lese-/Schreib-Position in der Datei), sinnvoll bei verbindungsloser Kommunikation.
- **zustandsorientiert**: Beim ersten Zugriff wird die Sitzung eröffnet, am Ende wird die Sitzung explizit geschlossen (z.B. bei verbindungsorienierter Kommunikation).

Ein Vergleich beider Design-Arten gibt folgende Tabelle:

Zustandslos	Zustandsorientiert
+ einfaches Wieder-Anlaufen eines Servers	+ bessere Performance
+ weniger Verwaltungsaufwand beim Server	− Ausfälle problematisch
− höheres Kommunikationsvolumen	− Client-Ausfälle blockieren Ressourcen

2.2.5 Replikation

Durch die **Replikation** von Dateien kann eine **höhere Verfügbarkeit** und **Performance** erreicht werden. Dabei sollte die Replikation vor dem Benutzer transparent bleiben (**Replikationstransparenz**). Replikation ist bei nur-Lese-Operationen unkritisch kann jedoch bei Schreib-Operationen zu Inkonsistenzen führen.

Primärkopie

Beim Einsatz einer **Primärkopie** wird ein Replikat gesondert behandelt, während alle andere sog. **Sekundär-Kopien** sind. Schreib-Operationen werden nur auf der Primär-Kopie ausgeführt, die dann die Sekundär-Kopien benachrichtigt. Dementsprechend können auf diesen nur Lese-Operationen erfolgen. Nachteil: die Primär-Kopie ist **central point of failure**.

Token Passing

Bei der **Token Passing** Methode wechselt die Verantwortlichkeit für die Master-Kopie mit einem **logischen Token**, das in einem virtuellen Ring von Station zu

Station weitergegeben wird. Vor- und Nachteile sind die bei der Verwendung von Tokens entstehenden.

Voting

Beim **Voting** gibt es eine Menge R von Rechnern, die ein Replikat besitzen und einen Replikationsgrad $n = |R|$ einer Datei. Die Anzahl der Simmen (**Votum**) wird mit V und das **Quorum** (die notwendige Anzahl von Stimmen für eine Aktion) mit Q bezeichnet. Vor jeder Aktion (Lesen oder Schreiben) muss eine Übereinstimmung von mindestens Q Replikaten gefunden werden. Auf diese Weise wird versucht, einen hohen Verteilungsgrad bei hoher Ausfallsicherheit zu gewährleisten. Bei Aktionen wird zwischen Lese- und Schreib-Operationen unterschieden: zum Lesen werden dann Q_R und zu Schreiben Q_W Stimmen benötigt, wobei $Q_R + Q_W > n$ sein muss. Im Fall **write-all-read-any** gilt $Q_R = 1$. Ausserdem können unterschiedlichen Rechnern auch unterschiedliche **Gewichte** bei der Abstimmung zugeordnet werden. Dabei ist S die Summe aller Stimmrechte mit

$$Q_R + Q_W > S \quad \text{und} \quad 2 \cdot Q_W > S.$$

Ausserdem gibt es noch **Zeugen** und **Geister**, die keine Datenkopie besitzen, jedoch an der Abstimmung teilnehmen. Sie helfen, eine aktuelle Kopie zu erkennen, belegen aber keinen Speicherplatz (z.B. Zeugen im Arbeitsspeicher bei plattenlosen Stationen).

Gitterverfahren

Beim sog. **Gitterverfahren** werden die Rechner in einem logischen Gitter angeordnet. Für einen Lese-Zugriff wird die Zustimmung von mind. einem Rechner in jeder Spalte benötigt, während für einen Schreib-Zugriff eine Zustimmung aller Rechner in einer Spalte und die Zustimmung eines Rechners von allen Spalten benötigt wird. Bei einem Gitter mit nur einer Spalte ist für ein **write-all-read-any** $Q_R = 1, Q_W = n$ und bei einem Gitter mit nur einer Zeile für ein **write-one-read-all** $Q_R = n, Q_W = 1$.

2.2.6 Skalierbarkeit

Durch die **Verlagerung der Verarbeitung** vom Server zum Client kann eine bessere Skalierung bei wachsender Anzahl von Clients ermöglicht werden. Eine andere Strategie ist das Vorhalten der Dateien möglichst nahe am Ort des Zugriffs (also bei den Clients).

Desweiteren können häufig auftretende Spezialfälle optimiert werden:

- viele kleine Dateien (Caching)
- vorrangig Lese-Zugriffe (entsprechende Quoren)
- häufiges Schreiben temporärer Dateien (delayed write)
- Datei-Sharing selten (Caching beim Client).

Dabei sollten häufige system-weite Veränderungen vermieden werden und die Prinzipien des **minimal trust** und **need to know** verfolgt werden.

2.3 Sun NFS

Die Hauptdirektive bei der Entwicklung des **SUN Network File System** (**NFS**) war die Erweiterung des lokalen Dateisystems. Dabei sollte es ermöglicht werden, entfernte Verzeichnisse an beliebiger Stelle im lokalen Dateibaum zu platzieren (mit der `mount`-Operation auf dem Client bzw. eines `export` auf Serverseite). Dabei kam ein zustandsloser Server mit Block-Caching beim Client mit **Block Read Ahead** zum Einsatz. Das regelmäßige Zurückschreiben modifizierter Blöcke (alle 30 Sek.) und das periodische Löschen von Daten/Verzeichnisblöcken aus dem Cache sollte Inkonsistenzen vermeiden. Die Aktualität einer Datei sollte beim Öffnen auf dem Server überprüft werden. Als **Kommunikationsmechanismus** kam RPC zu Einsatz. Das NFS etablierte sich bald zum de-facto Standard in UNIX-Umgebungen und vielen anderen Betriebssystemen.

Dabei gewährleistete NFS in folgenden Bereichen Transparenz:

- **Zugriffstransparenz**
  Die Zugriffsschnittstelle war identisch der beim lokalen Zugriff.
- **Ortstransparenz**
  Entfernte Dateien sind im lokalen Dateibaum an beliebiger Stele integrierbar.
- **Fehlertransparenz**
  Der zustandslose Server und (überwiegend) idempotente Dateioperationen erlauben einen problemlosen Restart des Servers – Client-Ausfälle wirken sich also nicht auf den Server aus.
- **Performancetransparenz**
  Durch Caching und andere Maßnahmen (**read ahead**, **delayed write**) wird eine hohe Performance erreicht, so dass der Verlust durch den Netzwerkzugriff minimiert wird.
- **Migrationstransparenz**
  Clients können mit verschiedenen Servern arbeiten, die das gleiche Dateisystem anbieten.

Die **Architektur** von NFS baut auf einem **Virtual File System** im UNIX-Kernel auf. Auf dieses wird über Standard-Schnittstellen zugegriffen und erst unterhalb dieser Schicht wird ein Unterschied zwischen einem entfernten und einem lokalen Zugriff gemacht. Soll ein Zugriff auf ein NFS-Share erfolgen, so greift der NFS-Client den Aufruf ab und leitet ihn über das NFS Protokoll an den NFS-Server auf dem Server-Rechner weiter, der den Aufruf von unten an das Virtual File System auf dem Server übergibt.

Innerhalb des Virtual File System wird zwischen einem lokalen und einem entfernten Zugriff unterschieden: entfernte Dateien werden durch **File Handles** und

lokale Dateien durch **i-nodes** repräsentiert. Das File Handle besteht dabei aus einer **filesystem ID**, **i-node number** und einer **i-node generation number**.

Der **Zugriffsschutz** und die **Authentisierung** erfolgen nur mit Hilfe des RPC. Da es sich um einen zustandslosen Dienst handelt, müssen bei jedem RPC-Aufruf alle Authentifizierungsinformationen mitgeschickt werden, was am besten verschlüsselt (ab Version 4.0) erfolgt.

Für die **Konsistenz** gibt es seit der Version 3 des NFS-Protokolls für die `write`-Operation zwei Optionen:

1. **Write-Through Caching:** Bevor dem Client eine Antwort geschickt wird, werden die Daten der `write`-Operation in den Speicher-Cache und auf Festplatte geschrieben. Der Client kann also davon ausgehen, dass die Daten persistent geschrieben wurden.
2. **Commit:** Die zu schreibenden Daten werden nur im Speicher gecached und erst dann auf Festplatte geschrieben, wenn eine `commit`-Operation für die Datei empfangen wird. Der Client kann sich also erst nach dem Aufruf eines `commit` sicher sein, dass die Daten persistent geschrieben wurden. Standardmäßig wird dieser Modus von NFS-Clients verwendet und ein `commit` beim Schliessen einer Datei aufgerufen.

Durch die Einführung der `commit`-Operation wurde ein Performance-Flaschenhals beseitigt, der bei der ersten Option schwer wiegen konnte.

Ein Caching auf Client-Seite ist ebenfalls sinnvoll. Dazu werden zu allen Datenblöcken im Cache die Zeitpunkte T_c der letzten Validierung und T_m der letzten Modifizierung auf dem Server gespeichert. Zu einem Zeitpunkt T ist ein Eintrag genau dann gültig, wenn gilt

$$(T - T_c) < t \quad \vee \quad T_m\,(\text{Client}) = T_m\,(\text{Server})\,,$$

wobei t das sog. **freshness intervall** beschreibt, für das ein Datenblock gültig ist.

2.4 Andrew File System (AFS)

Das **Andrew File System** (**AFS**) wurde im Rahmen des Andrew Projektes an der CMU entwickelt und später nach DCE als **DCE DFS** übernommen. Ein wichtiger Entwurfsaspekt war die gute **Skalierbarkeit** des Systems. Wie bei NFS werden Verzeichnisse an vorgegebener Stelle in den lokalen Dateibaum montiert, jedoch besitzen Server hier einen Zustand. Die Dateien werden als Ganzes beim Client gecached und der Server führt bei einer Änderung einer Datei einen Callback zum Client aus. Das System arbeitet also mit einer **Sitzungssemantik** und unterstützt die **read-only** Replikation.

Die gute Skalierbarkeit auch bei sehr vielen Clients wird durch das Cachen ganzer Dateien erreicht. Dabei besitzt AFS zwei unübliche Design Charakteristiken:

1. **whole-file serving:** Der gesamte Inhalt eines Verzeichnisses oder einer Datei wird übertragen (bei AFS-3 maximal 64 Kb).
2. **whole-file caching:** Hat ein Client eine Datei angefordert, so wird sie zu ihm übertragen und lokal gecached. Der Cache ist permament und überlebt auch Neustarts. Erst bei einem `close` wird die Datei wieder auf den Server zurückgeschrieben. Die Datei wird dann aber nicht gleich wieder aus dem Cache gelöscht für den Fall, dass der Client wieder auf die Daten zugreifen möchte.

Dem Design des AFS lagen folgende Annahmen zugrunde:

- Die meisten Dateien sind klein (<10Kb).
- `read`-Operationen sind wesentlich häufiger als `write`-Operationen.
- Sequenzieller Zugriff ist häufiger als zufälliger.
- Die meisten Dateien werden nur von einem Benutzer verwendet – wenn die Datei doch geteilt wird, so wird sie in der Regel nur von einem Benutzer gleichzeitig geändert.
- Dateien werden in Stößen referenziert: Wenn eine Datei kürzlich geöffnet wurde, so ist die Wahrscheinlichkeit sehr groß, dass sie bald wieder referenziert wird.

Wie man sieht passen Datenbanken nicht in dieses Schema, da sie häufig gemeinsam benutzt werden. Aus diesem Grund werden sie auch explizit von der Verwendung mit AFS ausgeschlossen.

Implementierung

Auf Client-Seite läuft der Prozess „**Venus**", der alle entfernten Dateizugriffe vom Unix-Kernel übermittelt bekommt. Er übersetzt Pfad-Namen in **fid**s (**file identifier**) und übermittelt diese an den Prozess „**Vice**" auf dem Server. Dieser liefert dann die gewünschte Datei an Venus zurück. Die **AFS Architektur** basiert auf Client- und Server-Prozessen (Venus und Vice), die auf dem UNIX-Kernel laufen und sich über das **AFS-Protocol** unterhalten.

Cache-Konsistenz

Zusammen mit der Datei übergibt Vice einen **callback promise** an Venus, das der übergebenen Datei zugeordnet ist und die Zustände „valid" und „cancelled" haben kann. Über einen **Callback** teilt dann der Server dem Client mit, wenn eine Datei geändert wurde, so dass dieser das **callback promise token** für die relevante Datei auf cancelled setzen kann. Damit dies gewährleistet werden kann, muss der Server zustandsbehaftet sein, was einen extra Verwaltungsaufwand erfordert (im Gegensatz zu NFS). Beim Öffnen eine Datei aus dem Cache wird genauso zunächst dieses Token geprüft, um sie dann u.U. neu vom Server anzufordern. Bei einem Neustart des Clients muss Venus seinen Cache mit Vice abgleichen, da Callbacks verpasst worden sein könnten. In diesem Fall werden

fids und Zeitstempel an Vice geschickt. Das gleiche gilt für Dateien, die eine festgelegte Zeit T nicht angefasst wurden.

Bei AFS kommen Cache-Konsistenz-Zusagen nur beim Öffnen oder Schliessen von Dateien zum Tragen. Desweiteren sind folgende Aspakte zu beachten:

- Unix Kernel-Modifications
- Location Database (Volume-names→Server)
- Threads
- Read-only Replikate
- Bulk-transfer (normalerweise 64 Kb, kann auf das Netzwerk optimiert werden)
- Partielles Datei-caching
- Skalierbarkeit und Performance (bei einem Benchmark nur 40% der Last gegenüber NFS)
- Wide-Areas support (Aufteilung in administrativ unabhängige Zellen)

DCE/NFS

Wie bereits erwähnt, wurde AFS in abgewandelter Form in DCE als **DCE/NFS** übernommen. Dabei wurde die Implementation derart verändert, dass zum Beschreiben einer Datei ein **write token** benötigt wird, um bestimmte Bereiche zu beschreiben. Jedes Token besitzt eine TTL, nach deren Ablauf es ungültig ist. Greift ein Client auf eine geänderte Datei zu oder schliesst ein Client eine Datei, so sind Callbacks vom Server die Folge.

2.5 Vergleich NFS vs. AFS

Vergleicht man die beiden Dateisysteme miteinander, so ergibt sich folgende Gegenüberstellung:

AFS	NFS
eideutig globaler Namensraum	lokaler Namensraum
Replikation	nicht unterstützt
Caching ganzer Dateien auf Client-Platte	Block-Caching
gute Skalierbarkeit durch Caching	begrenzte Skalierbarkeit
zustandsbehafteter Server	einfachere, zustandslose Server

2.6 Zusammenfassung

Beim Entwurf verteilter Dateisysteme sind folgende Design-Kriterien von besonderer Bedeutung:

- Effektives Caching bei den Clients,
- Wahrung der Konsistenz zwischen lokalen und entfernten Dateien, wenn diese geändert werden,
- Wiederherstellung nach Client-/Server-Absturz,
- Hoher Durchsatz bei Dateien veschiedener Grösse,
- Skalierbarkeit.

Ein weiterer Aspekt, der gerade heutzutage immer wichtiger wird ist die Einbindung mobiler Endgeräte, nämlich deren automatische Re-Integration, oder auch Quality of Service Garantien, persistente Abspeicherung und die Unterstützung von Multimedia-Streams und anderer zeitkritischer Daten.

3 Verteilte Transaktionen

3.1 Grundlagen

Unter einer **Transaktion** wird eine Folge logisch zusammenhängender Operationen, die automatisch ausgeführt wird, verstanden. Ziel ist eine höhere Zuverlässigkeit bei **möglichen Fehlern von Systemkomponenten** und **konkurrierendem Zugriff** auf gemeinsame Daten. Eine Transaktion besitzt immer einen Anfang und ein Ende – und kann gegebenenfalls auch abgebrochen werden.

Fehlermodell

Das **Fehlermodell** von Transaktionen kann sich auf unterschiedliche Bereich beziehen:

- **Kommunikation:** Unterbrochene Kommunikationsverbindungen, Netzpartitionen oder verlorene, duplizierte, beschädigte Nachrichten, oder auch eine falsche Reihenfolge.
- **Prozessor:** Beim `fail stop` hält der Prozessor ohne negative Auswirkungen auf die Daten, beim `fail fast` werden Fehler durch Ausnahmebedingungen angezeigt und danach hält der Prozessor. Genauso können ganze Knoten, einzelne Server, alle Server oder auch nur einzelne Module ausfallen.
- **Speicher:** Es wird zwischen flüchtigem, nicht-flüchtigem und stabilem Speicher unterschieden, wobei die Daten durch z.B. Paritätsbits oder Prüfsummen geschützt werden können.

Beim Zugriff mehrerer Prozesse auf gemeinsame Daten kann es zu verschiedenen Fehlerklassen kommen:

- **Verlorene Schreib-Operationen:** Prozess A und B lesen gleichzeitig einen Wert. B verändert ihn und schreibt ihn zurück. Jetzt verändert A den Wert und schreibt ihn auch zurück – damit ist die Änderung von B verloren.
- **Inkonsistenzen beim Lesen:** z.B. beim Parallelen Ausführen von Operationen (Buchungen tätigen und gleichzeitig den gesamten Geldbestand abfragen).

Transaktionsprimitiven

Wenn wir im Folgenden von Transaktionen sprechen, dann meinen wir, dass für deren Programmierung bestimmte **Transakionsprimitiven** benötigt werden:

- **BEGIN_TRANSACTION:** Markiert den Anfang einer Transaktion.
- **END_TRANSACTION:** Beendet die Transaktion und versucht einen `commit` durchzuführen.
- **ABORT_TRANSACTION:** Bricht eine Transaktion ab und versucht die alten Werte wieder herzustellen.
- **READ:** Liest aus einem Objekt.
- **WRITE:** Schreibt auf ein Objekt.

Serialisierbarkeit

Die **Serialisierbarkeit** erweist sich als wichtiges Mittel für Transaktionen. Sie soll durch Nebenläufigkeitskontrolle (**Concurrency Control**) sichergestellt werden und garantiert, dass wenn mehr als eine Transaktion gleichzeitig ausgeführt wird, das Endergebnis für alle anderen Prozesse so aussieht, als ob alle Transaktionen in einer bestimmten (systemabhängigen) Reihenfolge ausgeführt wurden (s. [Tanenbaum1995, S. 183]). Beispiele dafür sind **Sperren/Locks**, **optimistische Verfahren** (Konflikte werden ausgelassen und im Nachhinein bereinigt) und **Zeitstempel** (die Abfolge der Operationen wird anhand von Zeitmarkierungen auf mögliche Konflikte untersucht).

ACID

Als grundlegende Eigenschaften für Transaktionen gelten die **ACID-Eigenschaften**:

- **A**tomicity: Für die Außenwelt ist die Transaktion unteilbar.
- **C**onsistency: Eine Transaktion überführt ein System von einem konsistenten Zustand in einen neuen konsistenten Zustand.
- **I**solation: Parallele Transaktionen beeinflussen sich nicht gegenseitig.
- **D**urability: Nach erfolgreicher Beendigung einer Transaktion wird der Zustand dauerhaft gespeichert.

Dirty Read, Prema
te

Beim Einsatz von Transaktionen kann es zu Problemen kommen, die durch den Abbruch einer Transaktion entstehen können, wenn z.B. zwei Transaktionen parallel ausgeführt werden und eine Transaktion einen Wert schreibt, der dann von einer anderen gelesen und commited wird, wonach erstere abgebrochen wird (**Dirty Read**). Ein ähnliches Problem tritt auf, wenn eine Tansaktion einen Wert schreibt, den eine andere liest, wonach erstere wieder abgebrochen wird und dann letztere abgebrochen wird – in diesem Fall wird der Zustand zu Beginn der zweiten Transaktion wieder hergestellt, der den Schreibvorgang der ersten Transaktion beinhaltet (**Premature Write**).

Nebenläufigkeit

Die **Nebenläufigkeit** von Transakionen stellt eine neue Fehlerquelle dar. Insgesamt werden 4 Problemkategorien unterschieden, die durch Nebenläufigkeit auftreten können:

1. **Lost Update Problem:** Zwei Transaktionen lesen den alten Wert einer Variable und berechnen damit ihren neuen Wert. Bei der Bank, wenn zwei Konten z.B. um einen Anteil erhöht oder verringert werden sollen.
2. **Inconsistent Retrievals:** Bei der Bank wenn Buchungs-Operationen parallel zu Abfrage-Operationen laufen (z.B. branchTotal).
3. **Serial Equivalence:** Gleiche Anfangsbedingungen bewirken bei paralleler Ausführung mehrerer Transaktionen die gleiche Endsituation (**Serialisierbarkeit**).
4. **Conflicting Operations:** Zwei Operationen stehen in Konflikt zueinander, wenn das Endresultat von der Ausführungsreihenfolge abhängt. *„Damit zwei Transaktionen serialisierbar sind ist es notwendig und hinreichend, dass alle Paare von konfligierenden Operationen zweier Transaktionen in der gleichen Reihenfolge auf den Objekten, die sie betreffen, ausgeführt werden."* Zwei Operationen sind dabei konfligierend, wenn mindestens eine davon eine write-Operation ist.

Wiederherstellung

Ist eine Transaktion fehlgeschlagen, so muss der alte Zustand wieder hergestellt werden. Dabei kann es folgende Probleme geben:

- **Dirty Reads:** Die Isolations-Bedingung verlangt, dass keine Transaktion die Zwischenergebnisse einer anderen sieht, bevor diese abgeschlossen wurde. Führt eine Transaktion nun eine write-Operation durch, deren Wert eine andere liest und verwendet, so ist dies ein dirty read, wenn die erste Transaktion abgebrochen wird.
- **Wiederherstellbarkeit von Transaktionen:** Die Situation, dass eine Transaktion commited wird, während eine andere im Nachhinein abbricht, so dass die alte Situation nicht wiederherstellbar ist, kann dadurch verhindert werden, dass alle Operationen, die vielleicht zu einem dirty read führen so lange wie möglich „aufgeschoben" werden und erst commited werden, wenn die vorhergenden Transaktionen commited wurden.

- **Kaskadierende writes:** Damit bei der Wiederherstellungsmethode der **before images** korrekte Werte wieder hergestellt werden können, müssen Operationen so lange warten, bis vorhergehende Operationen, die das gleiche Objekt veränder haben, entweder commited oder aborted sind.

- **Strikte Ausführung von Transaktionen:** Damit die dirty reads und premature writes vermieden werden, müssen read und write Operationen so lange wie möglich verzögert werden. Ein Dienst der dieses zu Verfügung stellt und garantiert, dass alle Operationen, die ein Objekt verändern, das andere Operationen auch geändert haben, erst stattfinden, wenn die vorigen Operationen entweder commited oder aborted sind, heisst strikt.

- **Provisorische Versionen:** Damit der Server den alten Zustand eines Objektes nicht umständlich wieder herstellen muss, kann er provisorische Kopien der Objekte im Hauptspeicher erzeugen, diese manipulieren und bei einem commit wieder zurückschreiben.

Nested Transactions

Eine ganz neue Fehlerquelle stellen **geschachtelte Transaktionen** (**Nested Transactions**) dar. Hier laufen Sub-Transaktionen innerhalb von Transaktionen ab, wobei gilt, dass nur wenn alle Sub-Transaktionen mit einem `commit` enden, die Top-Level-Transaktion mit einem `commit` enden kann. Das kann zu Problemen führen wenn einige Sub-Transaktionen erfolgreich beendet werden und eine abbricht: In diesem Fall muss die gesamte Transaktion abgebrochen werden und somit müssen auch die erfolgreichen Sub-Transaktionen zurückgesetzt werden, die Dauerhaftigkeit ist hier also verletzt.

Optimistische Nebenläufigkeit

Bei dieser Methode werden alle Operationen auf Versuchsobjekten durchgeführt und dann am Ende commited. Dieser Prozess gliedert sich in 3 Schritte:

1. **Arbeitsphase:** Alle Operationen werden auf Versuchsobjekten durchgeführt – parallele Transaktionen arbeiten auf den gleichen Objekten, so dass dirty reads und premature writes nicht auftreten können.

2. **Validierungsphase:** Wenn die Transaktion beendet wurde wird geprüft, ob es zu Konflikten bei den Operationen kam.

3. **Update Phase:** Nur wenn es keine Konflikte gibt, wird die Transaktion commited, d.h. die Versuchsobjekte werden festgeschrieben. Read-only Transaktionen können gleich beendet werden, write Transaktionen müssen validiert werden.

Bestandteile eines Transaktionssystems

Als Bestandteile eines Transaktionssystems werden folgende Management-Bereiche ausgemacht:

- **Transaktionsmanagement:** Ist verantwortlich für die Bestimmung des Ergebnisses einer Transaktion.

- **Recovery Management:** Ist verantwortlich für die Wiederherstellung eines konsistenten Zustandes.
- **Puffermanagement:** Ist verantwortlich für den Transport der Daten zwischen flüchtigem und nicht-flüchtigem Speicher.
- **Log-Managment:** Ist verantwortlich für die Pflege der Transaktions-Logs.
- **Sperren-Management:** Ist verantwortlich für die Steuerung der Nebenläufigkeit.
- **Kommunikations-Management:** Ist verantwortlich für die Kommunikation zwischen Knoten.

3.2 Verteilte Ausführung

Verteilte Transaktionen

Wenn wir über **verteilte Transaktionen** reden, dann benötigen wir meist einen **Koordinator**, der sich um die ordnungsgemäße Abwicklung der Transaktion T kümmert (also die ACID-Eigenschaften sicherstellt) und alle Beteiligten konsistent benachrichtigt. Für die Serialisierbarkeit der Operationen muss sich wiederum jeder Server selbst kümmern. Die Rolle des Koordinators wird dabei in der Regel von dem Prozess übernommen, der die Transaktion startet. Dabei startet der Koordinator die Transaktionen auf anderen Servern, und diese registrieren (`join`) sich beim Koordinator, so dass beide Seiten eine Referenz aufeinander besitzen.

Atomische Commit-Protokolle

Eine Möglichkeit, verteilte Transaktionen durchzuführen stellen die sog. **atomischen Commit-Protokolle** dar:

- **Ein-Phasen-Commit:** Der Koordinator schickt ein `commit` oder ein `abort` solange an alle Beteilgten, bis alle mit einer Empfangsbestätigung geantwortet haben.
  Probleme entstehen, wenn Server voreilig in den `commit`-Zustand wechseln, obwohl die Gesamttransaktion nicht zu einem `commit` führt.
  Dieses Verfahren ist also nicht sinnvoll in Systemen mit unzuverlässigen Kommunikationskanälen und Systemausfällen.
- **Zwei-Phasen-Commit:** In der ersten Phase schickt der Koordinator allen Servern ein `prepare_to_commit` und bereitet sie so auf die Transaktion vor. Antworten alle Server mit einem `ok`, so schickt der Koordinator an alle Server ein `commit`. Dabei werden alle Interaktionen durch Time-Outs überwacht und Nachfragen gestellt, falls einmal keine Antwort kommt. Wenn ein Server mit `ok` geantwortet hat, so muss er die Transaktion später durchführen und kann diese Aussage nicht mehr zurücknehmen. Dazu muss er sowohl den neuen, als auch im Falle eines `abort` den alten Zustand speichern.

Mit dieser Methode können auch geschachtelte Transaktionen durchgeführt werden, die dann wie Bäume aussehen und aus Top- und Sub-Koordinatoren besteht.
Ein Problem kann das Handhaben von Server-Ausfällen und Kommunikationsproblemen sein.

Atomische Commit-Protokolle sind speziell für asynchrone Systeme geeignet, in denen Server und Kommunikationssystem ausfallen können. Dazu wird implizit vorausgesetzt, dass ein request-reply-Protokoll zur Verfügung steht, das beschädigte und doppelte Nachrichten verwirft – es wird davon ausgegangen, dass byzantinische Fehler nicht existieren. Um zu einem verteilten Konsens bzgl. der Transaktion zu kommen, können keine herkömmlichen verteilte Algorithmen zum Einsatz kommen, da diese Serverausfälle durch Ersetzen von Prozessen durch neue maskieren.

3.3 DCE / Encina

Die DCE-Architektur für verteilte Transaktionen heisst **Encina** und besteht aus einem **Transaction Manager** und **Ressource Managern** (den Servern). Unter Ressourcen verstehen wir z.B. Datenbanken, Dateisysteme oder Spooler. Die Dienste und Protokolle für das Transaktionsmanagement und das Recoverymanagement wurden im Rahmen der OSI festgelegt (**OSI TP**, **OSI Transaction Processing**).

Im Schichtenmodell sitzt Encina oberhalb des Betriebssystems, jedoch unterhalb der Anwendung (genau wie der IBM CICS Transaktionsmonitor) und kommuniziert mit der Ressource, die sich in der gleichen Schicht befindet.

3.4 Fehlerbehandlung

Fehlerbehandlung

Der **Recovery Manager** soll dafür sorgen, dass trotz Ausfällen von Beteiligten die ACID-Eigenschaften gelten – vor allem Atomicity und Durability. Dazu werden alle Aktionen einer Transaktion, die noch nicht abgeschlossen ist, protokolliert (Log-Datei), so dass nach dem Auftreten eines Fehlers der Recovery Manager dafür sorgen kann, dass wieder ein konsistenter Zustand eingenommen wird. Sowohl der Transaction Manager, als auch alle Resource Managers besitzen also einen eigenen Recovery Manager, mit dem sie kommunizieren.

Recovery

Durch seine Konzeption kann der Recovery Manager folgende Fehler behandeln:

- **Abbruch der Transaktion:** wenn ein Beteiligter ein `abort` schickt, müssen alle Transaktionen zurückgesetzt werden.
- **Ausfall des Servers:** Bei einem Fehler eines Servers müssen alle Transaktionen abgebrochen werden, an denen der Server beteiligt ist.

- **Ausfall eines Knoten:** Z.B. bei Stromausfall müssen alle Transaktionen aller Server abgebrochen werden.
- **Ausfall des Speichermediums:** Z.B. bei einem Platten-Crash müssen Daten wieder hergestellt werden.

Protokollierung

Jeder Server führt für seine Daten und die für ihn relevanten Transaktionen eine Liste der Modifikationen und Zustandsübergänge (**Journal**). Dazu werden:

- Alle geplanten Veränderungen an den Daten in einer Liste im nicht-flüchtigen Speicher vermerkt.
- Der Transaktionsmanager bestimmt `commit` oder `abort` für eine Transaktion.
- Bei einem `commit` werden die Änderungen der Daten in der Liste im nicht-flüchtigen Speicher ausgeführt.
- Bei einem `abort` wird die Liste gelöscht.

Für den Restart beim Two-Phase-Commit-Protokoll müssen folgende Schritte durchgeführt werden:

Rolle	Status	Aktion des Recovery Managers
Transaction Mngr.	bereit	Keine Entscheidung vor dem Ausfall → Abbruch an alle schicken
Transaction Mngr.	commited	Ausfall nach positiver Entscheidung → `commit` an alle schicken, falls nicht bereits geschehen
Resource Mngr.	commited	Ausfall nach positiver Entscheidung → `ack` an Transaction Mngr., falls nicht bereits geschehen
Resource Mngr.	ungewiss	Ausfall vor Entscheidung → Anfrage an Transaction Mngr. zum Ausgang der Transaktion
Resource Mngr.	bereit	Ausfall vor der Stimmabgabe → kann Abbruch erzwingen
Transaction Mngr.	fertig	Ausfall nach erfolgreichem Ende → keine Aktion erforderlich

Dieser Ablauf soll in Abbildung 1 verdeutlicht werden.

Logging

Für die Benutzung von **Recovery Files** gibt es zwei verschiedene Ansätze: Logging und Schatten-Versionen.

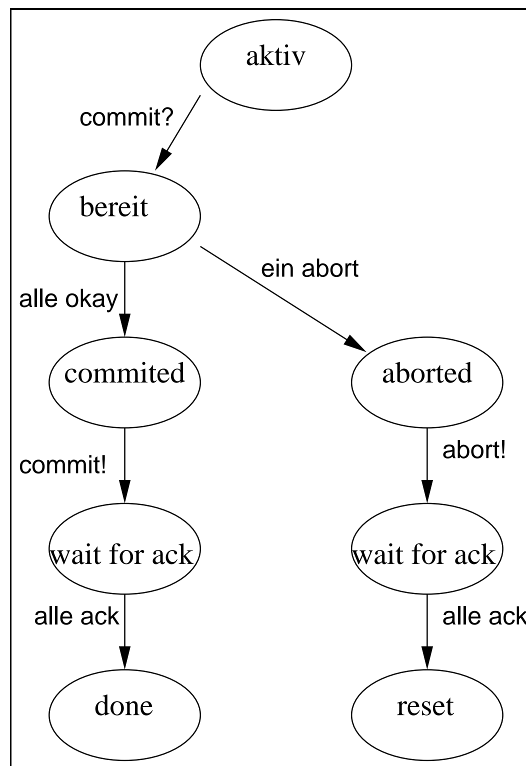

Abbildung 1: Two-Phase-Commit Protokoll

Beim **Logging** wird in einem recovery File ein Protokoll aller Transaktionen mitgeschrieben, die auf dem Server durchgeführt wurden. Das Protokoll besteht aus den Werten von Objekten, Transaktions-Status-Einträgen und einer Intentionsliste für Transaktionen in der Reihenfolge, in der sie durchgeführt wurden. Wenn ein Server einer Transaktion zustimmt (prepared), so werden alle der Transaktion zugehörigen Objekte der Intentionsliste im Log angehängt. Da sequentielles Schreiben auf die Platte schneller ist als der zufällige Zugriff, ergibt sich so zusätzlich ein Performance-Gewinn. Die gesamte Wiederherstellung muss dabei **idempotent** sein – kann also beliebig oft bei gleichem Ergebnis durchgeführt werden.

Durch das Setzen sog. **Checkpoints** wird verhindert, dass das Log-File über alle Grenzen wächst und die Wiederherstellung unnötig lange braucht. Sie müssen direkt nach dem commit einer Transaktion und dem prepare einer anderen gesetzt werden.

Schatten-Versionen

Im Gegensatz zum Logging wird beim Einsatz von **Schatten-Versionen** ein **Version Store** verwendet, in dem Objekte abgespeichert werden. Wenn nun eine Transaktion ein `prepare to commit` sendet, so werden die entsprechenden Objekte in den version store aufgenommen. Wird eine Transaktion commited, so

wird ein neuer version store erzeugt und die festgeschriebenen Objekte entfernt. Während sich das Logging beim Protokollieren als schneller erweist, ist der Einsatz von Schatten-Versionen bei der Wiederherstellung performanter.

3.5 Sperren

Sperren

Um eine Serialisierbarkeit beim Zugriff auf Objekte zu erreichen, kann dass **Sperren** von Objekten sinnvoll sein, d.h. auf ein Objekt kann nur dann zugegriffen werden, wenn es nicht von Sperren belegt ist. Ein Parameter beim Einsatz von Sperren ist die **Granularität**: Ab welcher Größe soll gesperrt werden? Desweiteren sind beim Sperren sowohl das **two-phase locking**, als auch das **strict two-phase locking** als Strategie möglich. Während es bei ersterem eine wachsende und eine schrumpfende Phase gibt, in der Sperren angefordert bzw. freigegeben werden, werden bei letzterem die Sperren erst dann freigegeben, wenn die Transaktion abgeschlossen ist.

Um Konflikte zwischen parallelen Transaktionen zu verhindern werden i.d.R. zwei Arten von Locks eingesetzt: **read locks** und **write locks**. Während zwei parallele read locks durchaus möglich sind, ist ein weiterer Lock, sobald ein write lock im Spiel ist, nicht mehr möglich. Auf diese Art werden lost updates und premature writes verhindert. Bei der Implementierung ist das Konzept eines **Lock Manager**s sinnvoll, der die Sperren zentral verwaltet und u.U. Deadlocks erkennt.

Auch verschachtelte Transaktionen sind mit Sperren möglich, indem alls Sub-Transaktionen ihre Sperren an die Vorfahren vererben, so dass am Ende die Top-Level-Transaktion entscheiden kann, ob die Transaktionen festgeschrieben werden oder nicht. Dafür dürfen allerdings Eltern- und Kind-Transaktionen nicht parallel laufen – das gilt jedoch nicht für Sub-Transaktionen auf der gleichen Ebene.

Erhöhte Nebenläufgkeit durch Sperr-Schemata

Durch ein anderes **Sperr-Schema** kann die Nebenläufigkeit von Transaktionen noch erhöht werden:

Two-version locking Bei diesem optimistischen Schema werden write-Operationen auf temporären Objekten durchgeführt, die dann auch von anderen Transaktionen gelesen werden können. Das Objekt wird bei einem commit geschrieben, allerdings darf es auch nur dann geschrieben werden – sobald es noch eine Transaktion gibt, die auch nur lesend darauf zugreift, kann es noch nicht geschrieben werden. Aus diesem Grund gibt es auch drei Arten von Sperren: read, write und commit Sperren.

Hierarchic locks Werden bei unterschiedlichen Applikationen verschiedene Granularitäten für Locks benötigt, so kann es sinnvoll sein eine Hierarchie von Sperren unterschiedlicher Granularität zu verwenden. Dadurch wird die Verwendung von Sperren ökonomischer.

eile beim Einsatz von
en

Natürlich haben Sperren auch **Nachteile**:

- Die Verwaltung von Sperren bringen einen **Overhead**, da sogar `read`-Operationen verwaltet werden müssen.
- Sperren können zu **Deadlocks** führen.
- Um verschachtelte Abbrüche zu vermeiden können Sperren erst am Ende einer Transaktion wieder freigegeben werden – ein Nachteil bzgl. der **Nebenläufigkeit**.

Sperren

Bevor eine Transaktion ein Objekt lesen oder schreiben kann, benötigt sie eine Lese- bzw. Schreib-Sperre. Beim Zwei-Phasen-Commit wird nach der Freigabe einer Sperre keine weitere Sperre mehr angefordert. Damit ist das Zwei-Phasen-Sperren sehr restriktiv und kann zu Deadlocks führen. Als Varianten mit besserer Performance wurden z.B. **semantische Sperren**, die das Wissen über die Applikationssemantik ausnutzen, und **frühes Freigeben von Sperren**, was evtl. aufwendige Korrekturmaßnahmen erfordert, wenn optimistische Annahmen nicht erfüllt werden.

4 Management

4.1 Grundlagen

> „Das Management vernetzter Systeme umfasst in seiner allgemeinsten Definition alle Maßnahmen, die einen effektiven und effizienten, an den Zielen des Unternehmens ausgerichteten Betrieb der Systeme und ihrer Ressourcen sicherstellen. Es dient dazu, die Dienste und Anwendungen des vernetzten Systems in der gewünschten Güte bereitzustellen und ihre Verfügbarkeit zu gewährleisten. Steht das Management des Kommunikationsnetzes und seiner Komponenten im Vordergrund, spricht man von **Netzmanagement**, liegt der Schwerpunkt auf den Endsystemen, bezeichnet man es als **Systemmanagement**. Das **Anwendungsmanagment** ist für verteilte Anwendungen und verteilt realisierte Dienste zuständig." [Hegering et al. 1999]

Zunächst einige Definitionen, das Management von Systemen betreffend:

Management Alle Maßnahmen für einen unternehmenszielorientierten effektiven und effizienten Betrieb eines verteilten Systems mit seinen Ressourcen.

OSI Network Management *„The facilities to control, coordinate and monitor resources which allow communications to take place in an OSI environment."*

ITU-T Management *„... to support the management requirements of administrations to plan, provision, install, maintain, operate and administer telecommunication networks and services."*

Isoliertes Management Jedes Management-Werkzeug steht für sich und ist isoliert in Bezug auf Hersteller, Funktionsbereich, Betrachtungsebene etc.

Koordiniertes Mangement Werkzeuge können kombiniert werden in der Art und Weise, dass die Ausgaben eines Werkzeuges als Eingabe eines anderen genommen werde können und so einfache Steuerungs-Skripte erstellt werden können.

Integriertes Management (nach [Hegering et al. 1999]) *„... Summe aller Verfahren und Produkte zur Planung, Konfigurierung, Steuerung, Fehlerbehebung sowie Verwaltung von Rechnernetzen und verteilten Systemen."* Demnach ist ein integrierter Ansatz nur dann gegeben, wenn die zu managenden Komponenten in einer herstellerunabhängigen Weise zu interpretieren sind. Genauso müssen Informationen über wohldefinierte Schnittstellen und Protokolle frei zugänglich sein.

Damit eine **Managementplattform** für integriertes Management in einer heterogenen Umgebung geeignet ist, müssen folgende Aspekte herstellerübergreifend spezifiziert sein:

- **Informationsmodell:** Beschreibung von Managementobjekten.
  Legt den Modellierungsansatz und eine eindeutige Syntax für die Beschreibung von Managementinformationen fest. ISO und OMG haben z.B. einen objektorientierten Ansatz gewählt, während IAB einen Datentypenansatz verfolgen.

- **Organisationsmodell:** Behandlung und Unterstützung von Organisationsaspekten, Rollen und Kooperationsformen.
  Proxies können eingesetzt werden um unterschiedliche Organisationsmodelle aufeinander abzubilden.

- **Kommunikationsmodell:** Beschreibung der Kommunikationsvorgänge zu Managementzwecken.
  Kommunikation kann dabei 3 verschiedene Zielsetzungen haben: Austausch von Steuerinformationen, Statusabfragen und (asynchrone) Ereignismeldungen. Dabei müssen Partner und Kommunikationsmechanismen festgelegt werden, Syntax und Semantik der Protokoll-Datenstrukturen definiert und Managementprotokolle in die Dienstarchitektur bzw. Protokollhierarchie eingebettet werden.

- **Funktionsmodell:** Strukturierung der Managementfunktionalität. Zergliedert den Gesamtaufgabenkomplex in Management-Funktionsbereiche wie Konfigurationsmanagement, Fehlermanagement, Abrechnungsmanagement und versucht generische Managementfunktionen festzulegen. Es ist also die Basis für Bibliotheken von Management-Teillösungen und für die Delegation von Teilaufgaben an Agenten.

Ein Rahmenwerk, dass diese Aspekte standardisiert wird **Managementarchitektur** genannt.

Die drei Dimensionen, die das Management umfasst illustriert Abbildung 2.

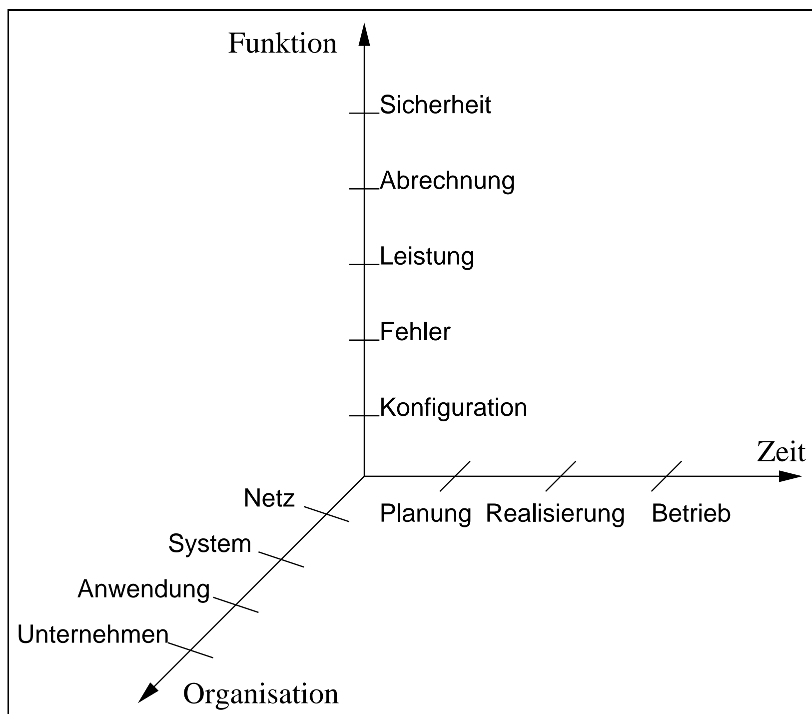

Abbildung 2: Die drei Dimensionen des Management.

4.2 OSI Management

Das **OSI Resource Management** (**OSI RM**) ist ein Standard für **Grundlagen des Managements eines Netz von offenen Systemen**. Dazu werden in vielen weiteren Standards Einzelheiten spezifiziert (z.B. System Management Overview, Common Management Information Protocol Specification / Service Specification). Dies war die erste Architektur, die alle 4 Teilmodelle ausgeprägt hat und kann somit als eine Art Referenzarchitektur dienen. Im Gegensatz

zum Internet-Management ist es komplexer, bietet dafür aber auch mächtigere Modellierungsmöglichkeiten.

Das OSI Management zerlegt das Problem in 4 Teilmodelle:

1. **Informationsmodell:** Eine objektorientierte Modellierung zur Abstraktion der Ressourcen dient der Strukturierung der Management Infomation Base (MIB).
2. **Organisationsmodell:** Es werden Rollen und Aufgaben der beteiligten Akteure unterschieden (Manager, Agent), wobei grundsätzlich von einem verteilten kooperativen Management ausgegangen wird.
3. **Kommunikationsmodell:** Kommunikationsvorgänge zur Abwicklung des Managements. Hier werden drei Mechanismen zum Austausch von Management-Informtionen definiert: die Kommunikation zwischen Management-Anwendungsprozessen der Schicht 7, die Kommunikation zwischen schichtspezifischen Managementinstanzen (Layer Management) und eine Managementkommunikation zwischen normalen Protokollinstanzen (Layer Operation).
4. **Funktionsmodell:** Die 5 Funktionsbereiche des Managements: Configuration, Fault, Performance, Accounting, Security. Desweiteren wird versucht generische Managementfunktionen abzuleiten.

> „Obwohl der Abstraktionsgrad des OSI-Managements sehr groß ist, ist er besser für die Bedürfnisse eines Betreibers eines umfangreichen Netzen mit komplexen und teuren Komponenten geeignet. Der objektorientierte Ansatz mit multipler Vererbung und Allomorphismus, das Trennen von Vererbungs-, Containment- und Registrierungshierarchien, flexible Steuermöglichkeiten über Diskriminatoren und Filtermechanismen sowie schließlich semantisch komplexe Systems Management Functions bieten dem Designer eines Managementsystems sehr viel Gestaltungsfreiraum. Dies ist der Grund, warum die OSI-Ansätze gerade im Bereich öffentlicher Netze großen Anklang gefunden haben.“ [Hegering et al. 1999, Kapitel 5]

Grundlegendes Managementmodell

Im **grundlegenden Managementmodell** führt der **Manager** Operationen auf dem **Agenten** in dem **Managed Object** aus, der wiederum Operationen auf der MIB des Managed Objects ausführt. Diese liefert Ereignisse an den Agenten zurück, die dieser wieder an den Manager weiterleitet (siehe auch Abbildung 4 auf Seite 32).

Der Manager ist ein **Common Management Information Service (Element)** (**CMIS(E)**), das über das **Common Management Information Protocol** (**CMIP**) mit anderen CMISEs kommuniziert. Unter einer **MIB** wird die **Managament Information Base** verstanden, in der die Management-Informationen eines Managed Objects gespeichert sind.

4.2.1 Informationsmodell

Das **Informationsmodell** im OSI Management modelliert eine Resource als **Managed Object** (**MO**), auf das über ein **Interface** zugegriffen werden kann. Es besitzt **Attribute**, die den Zustand der Ressource anzeigen, **Operationen**, die Abfragen und Operationen auf dem Objekt erlauben, und **Notifikationen** , die partielle Ereignismeldungen des Objektes liefern. Alle MOs eines offenen Systems zusammengefasst ergeben seine **Management Information Base** (**MIB**).

Vererbung

Gleichartige Objekte werden zu Klassen zusammengefasst, die durch Vererbung Eigenschaften an Unterklassen weitergeben. Durch **Allomorphie** (oder polymorphes Verhalten) kann sich ein Objekt wie die Instanz einer (beliebigen) anderen Klasse verhalten. Dadurch können gleich ganze Klassen von Objekten (z.B. aufwärtskompatible Software) gemanaged werden.

GDMO

Die **Guideline for the Definition of Managed Objects** (**GDMO**) spezifiziert MOs durch einfache Template-Notation, die mit dem ASN.1-Makromechanismus defniert werden. Dabei werden 9 Arten von Templates definiert:

- **Managed Object Class:** Klassendefinitionen
- **Package:** Zusammenfassung von Definitionen
- **Parameter:** Operationsparameter
- **Attribute:** Objektattribut, gibt die Eigenschaften und den Status des Managementobjektes an. Attributtypen können z.B. Zähler, Pegel, Schwellwerte sein. Für Attribute können erlaubte Operationen und Wertebereiche angegeben werden.
- **Attribute Group:** Gruppierung von Attributen, auf die dann mit einer attributbezogenen Operation zugegriffen werden kann.
- **Behaviour:** Verhaltensspezifikation, die Semantik von Attributen, Verhalten und Operationen i.d.R. informell natürlichsprachlich beschreibt.
- **Action:** Aufrufbare Operationen auf einem Objekt, die durch das Action Template festgelegt werden. Bereits definiert sind `createMO` und `delteMO`.
- **Notification:** Asynchrone Meldung vom Objekt
- **Name Binding:** Zur Benennung der Objektinstanzen

Ein Beispiel für ein Template ist folgendes Template „Managed Object Class":

```
<class-label>     MANAGED OBJECT CLASS
[DERIVED FROM     <class-label>           -- Oberklassen
                  [<class-label>]*;       -- optional
]
```

```
[CHARACTERIZED BY <package-label>            -- Merkmale
                  [<class-label>]*;
]
[CONDITIONAL PACKAGES
                  <package-label> PRESENT IF -- bedingte
                  <condition definition>     -- Merkmale
                  [<package-label> PRESENT IF
                  <condition definition> ]*;
]
REGISTERED AS     object identifier;         -- MO-Klassen-
                                             -- identifikator
                                             -- im ISO-Regis-
                                             -- trierungsbaum
```

Alle vordefinierten MOC- und sonstige Template-Beschreibungen, die wiederverwendet werden können sollen, werden im ISO-Registrierungsbaum abgelegt. Dabei haben neben der ISO und der ITU auch andere Standardisierungs-Konsortien MOC-Kataloge definiert.

Grafisch kann ein Template wie in Abbildung 3 dargestellt werden.

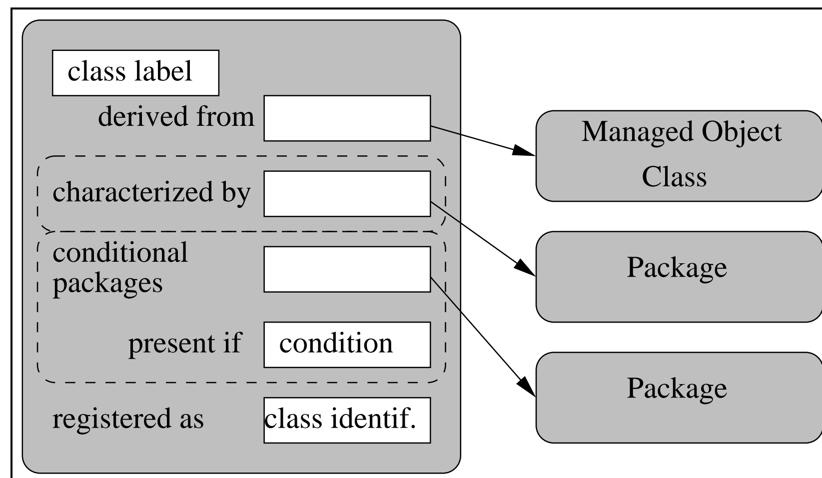

Abbildung 3: Grafische Darstellung eines Template

Bei den ASN.1 Object Identifiers wurde ein registriertes „Objekt” bezeichnet (z.B. ISO Standard oder ASN.1 Modul). Sie bestanden aus einer Folge nichtnegativer Integer-Werte,die optional mit einem Namen versehen waren und wurden zentral in einer hierarchischen Anordnung vergeben. Als Beispiel für GDMO soll hier die MO-Klasse für ein Modem angegeben werden. Sie kann sukzessive verfeinert werden, ist hier aber sehr einfach gehalten:

```
modem  NAMED OBJECT CLASS
   DERIVED FROM "ISO/IEC 10165-2 : 1992" : top;
   CHARACTERIZED BY   modemPackage;
REGISTERED AS    (iso(1) smi(3) classes(5) modem(7));
```

Das Package für ein Modem könnte dann folgendermassen aussehen:

```
modemPackage PACKAGE
   BEHAVIOUR      modemfunktion;
   ATTRIBUTES     modemNumber        GET,
                  anzFehler          GET;
   NOTIFICATIONS protocolFehler;
REGISTERED AS    (iso(1) ms(9) smi(3) package(4) modem(1));
modemfunktion    BEHAVIOUR
    DEFINED AS   Beschreibung des Modemverhaltens in Textform;
protocolFehler   NOTIFICATION
   BEHAVIOUR      ...                -- behaviour package
   WITH INFORMATION SYNTAX ...       -- Syntax des Inhalts
   WITH REPLY SYNTAX ...             -- erwartete Reaktion
```

Eine Objekt-Instanz wird mittels eines **Name Binding Template** benannt. Dazu müssen seine Attribute eindeutig sein und als Namensattribute gekennzeichnet werden. Dadurch wird eine „Enthalten-sein-in"-Hierarchie definiert.

```
modemNameBinding   Name BINDING
   SUBORDINATE OBJECT CLASS  NAME BINDING
   NAMED BY
   SUPERIOR OBJECT CLASS  "ISO/IEC 10165-2:1992":commSystem;
   WITH ATTRIBUTE modemNumber;  -- das eindeutige Attribut
   BEHAVIOUR
      modemContainmentBehaviour BEHAVIOUR
         DEFINED AS  maximal 3 Modems duerfen in einem
                     commSystem sein ; ;
   CREATE ...     -- Angaben, was beim Create passieren soll
   DELETE ...     -- Angaben, was beim Delete passieren soll,
                  -- z.B. dass alle Subord. gelöscht werden
```

Vererbung und Name Binding definieren zwei völlig getrennte Hierarchien, nämlich die Klassenhierarchie und die Objekthierarchie (so kann ein Computer z.B. ein `commSystem` enthalten, das wiederum ein `modem` enthält, während in der Klassenhierarchie der Computer und das `commSystem` auf einer Stufe stehen).

In der Klassenhierarchie gibt es folgende **generische, vordefinierte Objektklassen**:

- **TOP:** Wurzel der Vererbungshierarchie
- **SYSTEM:** Wurzel der Enthaltungshierarchie (Naming Tree)
- **DISCRIMINATOR:** ein MO, das Kriterien zur Steuerung beinhaltet
- **LOG:** dient zur Speicherung und Buchführung

In den OSI-Schichten gibt es folgende vordefinierte Objektklassen:

- **APPLICATION PROCESS:** Anwendungsprozess
- **COMMUNICATION ENTITY:** Protokollinstanz
- **PROTOCOL MACHINE:** Protokollautomat
- **CONNECTION-LESS PROTOCOL MACHINE**
- **CONNECTION-ORIENTED PROTOCOL MACHINE**
- ...

Genauso gibt es vordefinierte abstrakte Attribut-Typen:

- **COUNTER:** Zähler
- **GAUGE:** Anzeige
- **THRESHOLD:** Schwellenwert
- **TIDE:** Pegel
- ...

Im Falle des COUNTER gibt es folgende Eigenschaften:

- die **Struktur** ist eine einfacher Wert
- der **Wertebereich** umfasst alle natürlichen Zahlen beginnend bei Null, Inkrement=1
- es gibt die **Operationen** Lesen und Zurücksetzen
- es existieren **Beziehungen** zwischen Attributen wie evtl. ein Zähler-Schwellenwert, also ein Ereignis, das den Zähler zurücksetzt
- unter der **Anwendung** wird die Beschreibung der Verwendung des Zählers verstanden

MOs stehen nicht für sich alleine, sondern sind im Kontext zu sehen: Attributtypen können miteinander in Beziehung stehen (wie beim Zähler mit Schwellwert), auf andere MOs verweisen oder mittels Namebinding in einer Enthaltenseinsrelation zueinander stehen. Die Relationen zwischen MOs können in dem **General Relationship Model** (**GRM**) als **managed relationship** beschrieben werden.

4.2.2 Organisationsmodell

Im **Organisationsmodell** wird die **Management Domain** weiter unterteilt, um sie einfacher verwalten zu können. Dies kann zu **organisatorischen Domänen** (z.B. Sicherheitsmanagement und Abrechnungsmanagement) oder zu **administrativen Domänen** (die Verwaltungsautoritäten besitzen, die eine organisatorische Domäne verwalten) führen. Dabei werden zwei Rollen für Systeme unterschieden: die **Managerrolle** und die **Agentenrolle**, wobei OSI-Systeme grundsätzlich beide Rollen wahrnehmen und sogar dynamisch zwischen ihnen wechseln können. Dem liegt ein asynchrones asymmetrisches Kooperationsmodell zugrunde, in der Kooperation mit Hilfe der Management-Schnittstelle CMIS (Abbildung 4 auf der nächsten Seite) geschieht.

Unter der **Management Domain** und der **Management Policy** wird verstanden:

- Der Domäne ist ein Gruppenkriterium zugeordnet (z.B. Location)
- Einer Domäne ist eine benennbare **Management Policy** zugeordnet (also eine Menge von Regeln und Richtlinien für Management-Aktionen)
- Die Management Policies können definiert werden für z.B. Sicherheit, Benennung, Adressierung, Abrechnungsstrategie
- Eine Management Policy ist ein selbst änderbares Managementobjekt
- Zur Definition von Management Domains und Management Policies existieren entsprechende GDMO Schablonen

Allgemein wird zwischen Organisations- und Verwaltungsdomänen unterschieden. In **Organisationsdomänen** (**Functional Domains**) werden MOs nach funktionellen Gesichtspunkten, Funktionsbereichen oder nach temporärer Rollen-Zuordnung organisiert. In **Verwaltungsdomänen** (**Administrative Domains**) werden Organisationsdomänen eingerichtet, um sie manipulieren und den Steuerfluß zwischen den Domänen kontrollieren zu können.

Domänen selbst sind wieder Gegenstand von Management-Entscheidungen und somit Managementobjekte, die mittels einer MOC beschreibbar sind. Die Management-Policy darf das Verhalten der MOs beschränken, jedoch nicht ihrem Verhalten widersprechen.

4.2.3 Kommunikationsmodell

Ziel des Management ist es die Ressourcen betriebszielorientiert zu verwalten. Damit nun in einem offenen System Management-Informationen ausgetauscht werden können, muss ein einheitliches Kommunikationsmodell geschaffen werden. Im OSI-Managementmodell werden 3 Managementkategorien unterschieden: das schichtübergreifende Management (**Systems Management**, **SM**), das

Schichtenmanagement (**Layer Management**, **LM**) und das Protokollmanagement (**Layer Operation**). Dabei ist von der ISO nicht festgelegt, wie diese drei Management-Kategorien zusammenarbeiten.

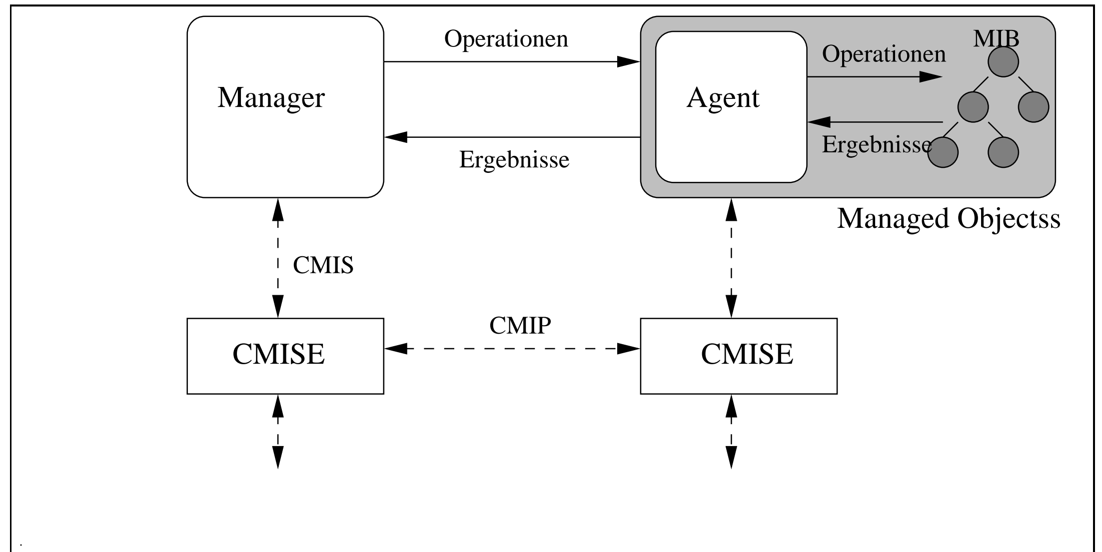

Abbildung 4: Common Management Information Service (CMIS)

Der **Common Management Information Service** (**CMIS**) ist ein **verbindungsloser Dienst**, der auf ACSE und ROSE aufbaut. Er erlaubt den Zugriff auf entfernte Managed Objects und es werden verschiedene **Dienstgruppen** unterschieden:

- **Assoziationsverwaltung** (initialisieren, terminieren, abbrechen)
  `M-INITIALIZE`, `M-TERMINATE`, `M-ABORT`
- **Ausführen von Operationen**
  `M-GET` (Lesen von MO-Attributen)
  `M-SET` (Modifikation von MO-Attributen)
  `M-ACTION` (Auslösen einer MO-Operation)
  `M-CREATE` (Kreieren eines MOs)
  `M-DELETE` (Löschen eines MOs)
- **Mitteilen von Ereignissen**
  `M-EVEN-REPORT` (Übertragen einer MO-Notification)

Dabei können Dienste sowohl bestätigt als auch unbestätigt sein.

Bei CMIS können Objekte wie folgt ausgewählt werden:

- Durch die **Benennung von einzelnen Objekten** gemäß der Enthalten-Sein-Hierarchie im MIT (Namebinding)
- Durch **Scoping**, bei dem ausgehend von einem benannten Basisobjekt „Objekt-Bereiche" für eine Verarbeitung benannt werden (z.B. alle Objekte unterhalb eines Basisobjekts)
- Durch **Filtering**, bei der aus der Menge der spezifizierten Objekte gewisse Objekte durch Filter-Attribute ausgewählt werden können (z.B. alle Objekte mit Zähler =0)
- Durch **Synchronisation**, so dass beim Zugriff auf eine Menge von Objekten bestimmte **Synchronisationsbedingungen** angegeben werden können (z.B. atomic, best effort)

4.2.4 Funktionsmodell

Das OSI Management wird in folgende **5 Funktionsbereiche** (**OSI Systems Management Functional Areas**, **SMFA**) aufgeteilt:

- **Konfigurationsmanagement:** Definieren und organisieren von Ressourcen, Einstellen und Änderung von Attributen, Sammeln von Zustandsinformationen.
- **Fehlermanagement:** Erkennen, Lokalisieren und Beheben von Störungen.
- **Leistungsmanagement:** Kontrolle des Leistungsverhalten.
- **Abrechnungsmanagement:** Benutzerverwaltung und Abrechnung von Nutzungsentgelten.
- **Sicherheitsmanagement:** Verwaltung von Sicherheitsvorkehrungen, Generierung und Austeilen kryptografischer Schlüssel.

Dabei werden **Funktionsbereiche** nach **Funktionalität** (Sinn und Zweck des Bereichs), **Prozeduren**, die notwendig sind, um Bereichsfunktionalität zu erbringen, und **Managed Object Classes**, die für den Bereich wichtig sind, aufgeteilt.

Beispiele für **System Management Functions** (**SMF**s) sind:

- **Object Management Function:** Erzeugen und Löschen von MOs sowie Lesen und Ändern von MO-Attributen
- **State Management Function:** Allgemeine Operationen zum Zustandsmanagment der MOs und Definition eines Satz von Operationen zur Steuerung der Zustandübergänge

- **Atributes for Representing Relationships:** Unterstützung für das Einrichten und Manipulieren von Beziehungen zwischen MOs durch ein eigenes Relationship Template
- **Alarm Reporting Function:** Generische Klassifikation von Alarmen nach ihrer Ursache
- **Event Report Management Function:** Zusammenstellung eingehender Meldungen zu Berichten und deren Weiterleitung über Filterfunktionen
- **Log Control Function:** Sammeln und Ablegen von Ereignismeldungen
- **Security Alarm Reporting Function**: s. Alarm Reporting, diesmal angepasst auf spezifische Sicherheitsbelange
- **Workload Monitoring Function**, etc.

Das Funktionsmodell wird laufend erweitert, um immer mehr formalisierte Managementfunktionalität in generischer Art und Weise zu definieren und damit als wiederverwendbarer Anwendungsbaustein für Managementlösungen zu dienen.

4.3 Internet Management

An der Erstellung von Internet Standards sind das **IAB** (**Internet Advisory Board**) und innerhalb dieser die **IRTF** (**Internet Research Task Force**) und die **IESG** (**Internet Engineering Steering Group**) beteiligt. In der IRTF gibt es mehrere **IRSG** (**Internet Research Steering Group**), die sich in verschiedenen **Research Group**s zusammenfinden. In der IESG gibt es innerhalb verschiedener **Area**s wieder verschiedene **Working Groups**. Standards werden in sog. **RFC**s (**Request For Comments**) veröffentlicht.

Die Internet-Managementarchitektur kennt kein sehr ausgeprägtes Organisations- oder Funktionsmodell und ist im Vergleich zum OSI-Modell einfacher und nicht objekt-orientiert. Hier war Einfachheit die Entwicklungsvorgabe, so dass die Flexibilität hier nicht im Informationsmodell steckt, sondern über die Managementanwendung erzielt werden muss. Dabei setzt SNMP auf UDP auf.

4.3.1 Organisationsmodell

Im Internet wird zwischen **Manager**, **Agent** und **Proxy Agent**, der sog. **Foreign Device**s anbinden kann, unterschieden. Jeder Agent hat Zugriff auf die **MIB** des zu verwaltenden Objektes.

4.3.2 Informationsmodell

Entwurfsziel Nummer eins war **Einfachheit**. Dabei wurden „Objekte" wie bei OSI definiert, man ging aber von einem anderen Objektmodell aus. Es gibt nämlich **keine Vererbung**, nur wenig Datentypen und nur sehr eingeschränkte Zusammensetzungen. Die Objekte selbst wurden mit ASN.1 Makros (**Structure of Management Information**, **SMI**) definiert. Dabei wurden Unterscheidungen getroffen zwischen **einfachen Typen** (nur 4 ASN.1 Basistypen zulässig), **applikationsweiten Typen** (für Zwecke des Managements definiert) und **einfach zusammengesetzten Typen** (Listen und Tabellen). Dazu folgendes Zitat:

> „The impact of adding network management to managed nodes must be minimal, reflecting a lowest common denominator."

Im **Simple Network Management Protocol** (**SNMP**) werden einfache Basistypen zur Verfügung gestellt: **Integer**, **Object String**, **Object Identifier** und **Null**. Die Kodierung wird dann mit ASN.1 **Basic Encoding Rules** (**BER**) vorgenommen. Ein von der IAB definiertes ASN.1-Makro definiert, wie Knoten mit „echten" Managementfunktionen beschrieben werden können: Name und Objektidentifikator, Syntax der Managementinformationen als ASN.1-Datentyp, Informelle Beschreibung der Managementinformationen, Statusinformationen.

Beispiele für applikationsweit definierte Typen sind z.B. `IpAddress`, `NetworkAddress`, `Counter`. Es können aber auch **Zusammengesetzte Typen** erzeugt werden:

```
<list> ::= SEQUENCE { -- Spalten in einer Tabelle
  <type 1>            -- <type> nur primitiver Typ!
  <type 2>
  ...
  <type N>
}
<table> ::= SEQUENCE OF { -- nur zwei-dimensional
  <list>
}
-- Es kann stets nur auf einzelne Datenelemente
-- zugegriffen werde, nicht auf ganze Spalten
-- oder Tabellen.
```

Es folgt ein Beispiel einer Definition von Objekten mit dem ASN.1 Makro „`OBJECT-TYPE`":

```
OBJECT-Type MACRO ::=
BEGIN
   TYPE NOTATION ::= "SYNTAX" type (TYPE ObjectSyntax)
                     "ACCESS" Access
```

```
                       "STATUS" Status
   VALUE NOTATION ::= value  (VALUES ObjectName)
   Access ::= "read-ony"
              |   "read-write"
              |   "write-only"
              |   "not-accessible"
   Status ::= "mandatory"
              |   "optional"
              |   "obsolete"
END
```

Hier ein paar Beispiele für die Definition von Objekten:

```
ipInReceives OBJECT-TYPE
   SYNTAX  Counter
   ACCESS  read-only
   STATUS  mandatory
   DESCRIPTION "Number of all received IP datagrams"
::= { ip 3}  -- object identifier
ipRouteTable OBJECT-TYPE
   SYNTAX  SEQUENCE OF IpRouteEntry
   ACCESS  not-acccessible
   STATUS  mandatory
   DESCRIPTION "This machine's routing table"
::= { ip 21 }  -- object identifier
IpRouteEntry ::= SEQUENCE {
   ipRouteDest     IpAddress -- Zieladresse
   ipRouteIfIndex  INTEGER   -- Nummer des Interfaces
   ...
   ipRouteNextHop  IpAddress -- nächster Knoten
   ipRouteType     INTEGER   -- z.B. direct, remote, ...
   ...
   ipRouteAge      INTEGER
   ...
}
```

Da eine Vererbung nicht möglich ist und somit ein hoher Grad an Redundanz bei MIBs entsteht, wurden einige allgemeine MIBs z.B. zum Host-Monitoring, X.500 Monitoring etc. definiert. Damit wurde eine Öffnung des Internet-Managements auch zu anderen Anwendungsbereichen hin signalisiert.

Nach der ersten Version wurde 1990 der Internet Standard **MIB-2** verabschiedet. Im Vergleich zum Vorgänger bringt er Verbesserungen und Erweiterungen. Alle Geräte, die „Internet Management"-konform sein wollen, müssen MIB-2 implementieren. In der **MIB-2** gibt es folgende **Objektgruppen**:

1. **System** (Contact, Name, UpTime, Services, Location etc.)
2. **Interfaces** (InterfaceTable, Index, Speed, Phys. Address, Status etc.)
3. **AT** (Adress Translation)
4. **IP** (Counters, AddresTable, RoutingTable etc.)
5. **ICMP**
6. **TCP** (Counters, ConnectionTable etc.)
7. **UDP** (Counters, UDPPortsTable etc.)
8. **EGP**
9. **OIM** (Managemenobjekte zu CMOT, CMIP over TCP)
10. **Transmission** (Managementinformationen zu verschiedenen Übertragungsprotokollen)
11. **SNMP** (Counters etc.)

Die SNMP MIB Objekte werden wieder in einer Baumstruktur **registriert**, deren Aufbau teilweise festgelegt ist. Über die **Adressierung von MIB Objekten** macht die SMI keine Angaben. Dies wird als Teil des Managementprotokolls angesehen. Das OSI Management vergibt Namen schon bei der Definition der Klassen (über die Containment Hierarchie), während SNMP **Objekt-IDs** benutzt, um auf Instanzen zu verweisen.

4.3.3 Kommunikationsmodell

Das SNMP geht davon aus, dass die **Management Station** ausreichend Verarbeitungskapazität hat, während **Managed Nodes** sehr einfache Geräte sein können. Als Protokoll auf Transportebene wird UDP eingesetzt, wobei die **well-known-Ports 161 und 162** Verwendung finden (162 nur für traps, also asynchrone Meldungen). Das ganze ist **inhärent Polling-orientiert** (einfache Agents!) und bietet nur triviale Sicherheitsfunktionen an (Kar-Text-Authentifizierung, einfache Authentifizierungsmechanismen beim Zugriff auf Objekte). Ausserdem kann immer nur auf einzelne Datenelemente zugegriffen werden.

Als **Operationen** stehen **GET** (Lesen von Managementdaten), **Get-Next** (Zusammenhängendes Lesen von Managementdaten), **Set** (Schreiben von Managementdaten) und **Trap** (Ereignismeldung) zur Verfügung. Eine **SNMP Nachricht** hat folgendes Format:

```
Message ::= SEQUENCE {
    -- PDUs für Operationen 1-3 haben gleiche Struktur,
    -- Traps haben eigenes Format
```

```
   version INTEGER         -- Version des Protokolls
      { version-1(0), version-2(1) },
   community OCTET STRING, -- Name einer Gruppe
   data ANY                -- für PDUs etc.
}
```

Eine **SNMP PDU** ist wie folgt gemäß der ASN.1-Basic Encoding Rules definiert:

```
PDU ::= SEQUENCE { -- alle PDUs haben gleiche Struktur
   request-id INTEGER,
   error-status INTEGER {
     noError(0), tooBig(1), noSuchName(2),
     badValue(3), readOnly(4), genErr(5)
     },
   error-index INTEGER,
   variable-bindings VarBindList
}
VarBindList ::= SEQUENCE OF VarBind
VarBind ::= SEQUENCE {
   name ObjectName,
   value ObjectSyntax
}
```

Der **Zugriff auf Objektinstanzen** ist durch das Informationsmodell vorgeschrieben, da es definiert, dass es in einer MIB von jedem Objekttyp nur eine Instanz geben darf. Diese wird dann mit „**Object Identifier.Suffix**" adressiert. Als Regeln gelten dabei:

1. Nur Blätter des Baumes können adressiert werden.
2. Falls das zu adressierende Objekt nicht in einer Tabelle enthalten ist, d.h. eine einfach MIB-Variable ist, dann erhält es den Suffix 0.
3. Falls das zu adressierende Objekt Teil einer Tabelle ist, dann muss die MIB-Definition einen eindeutigen Index benennen, der zur Adressierung benutzt wird.

Sicherheit

Der Sicherheitsmechanismus von SNMP verpackt die zu übertragenden Protokolleinheiten in sog. **SNMP-Messages**, die neben der PDU noch eine Versionsnummer und einen **Community String** enthält, der als eine Art Passwort für den MIB-Zugriff dient, da von dem Agenten nur die Nachrichten mit seinem Community String verarbeitet werden. Diese Form der Authentisierung heisst **Trivial Authentication Algorithm** und kann einfach durch das Abhören des Netzwerkverkehrs umgangen werden (von Integrität ganz zu schweigen). Ab SNMPv2p sind der Einsatz von MD5-Prüfsummen und DES im CBC-Modus vorgesehen, um mehr Sicherheit in das Protokoll zu integrieren.

4.3.4 SNMPv2

Da SNMP große Schwächen in den Bereichen Funktionalität und Sicherheit hatte, wurde 1993 SNMPv2 veröffentlicht. Nachteile bei der Funktionalität von SNMPv1 waren:

- unmögliche Übertragung größerer Datenmengen
- nur Polling vom Manager ausgehend möglich; indirekt kann der Agent aber auch über Traps Polling auslösen
- nur eine primitive Unterstützung asynchroner Ereignisse
- Gefahr von Datenverlusten wegen Abstützung auf UDP

Zu den wichtigsten Neuerungen von SNMPv2 (auch SNMPv2 Classic oder SNMPv2p) zählten **Datentypen** (Counter64, NsapAddress, BIT STRING), **Operationen** (GetBulk, Inform) und **Sicherheitskonzept** (Authentisierung, Vertraulichkeit, Zugangskontrolle). Das neue Protokoll sieht ausserdem eine **Manager-zu-Manager-Kommunikation** vor (**Manager-to-Manager MIB**), für die die **inform Operation** eingeführt wurde.

In dem **Sicherheitskonzept** nehmen interagierende Instanzen die Rolle einer „**Party**" ein, wobei pro Instanz mehrere Parties möglich sind. Dementsprechend enthalten SNMPv2-Nachrichten Sicherheitsinformationen (im Nachrichtenkopf), die im Kontext der interagierenden Parties interpretiert werden. Mögliche Kombinationen sind:

1. nicht gesichert (`non-secure`)
2. authentisiert/nicht vertraulich, mit Digest und Zeitstempel (`authinfo`)
3. nicht authentisiert/vertraulich
4. authentisiert und vertraulich

Die **Authentisierung** basiert dabei auf **geheimen Schlüsseln** (DES) und der rechtzeitigen Übertragung (**Alterungsmechanismus**). Das Sicherheitskonzept verlangt relativ aufwendige lokale Speicherung von Uhrzeiten, Schlüsseln, Rollen, Kontexten in einer lokalen **Party MIB**. Die Kommunikation zwischen Managern wird durch Manager-to-Manager-MIBs (**M2M-MIB**) ermöglicht.

4.4 Vergleich OSI vs. Internet

Folgende Tabelle vergleicht das Management der OSI und des Internet miteinander:

Internet	OSI
einfach und effizient	komplex und sehr mächtig
einfaches Informationsmodell mit Variablen und Tabellen	mächtiges Informationsmodell, Objektorientierung, Objektmethoden, ...
Benennung von Instanzen über OIDs, nur eindeutig innerhalb eines Systems, kompakte Kodierung	X.500-artige Benennung der Instanzen, global eindeutige Namen, längliche Kodierung
Polling orientiert, Traps sind Ausnahmen	Polling und Ereigniss gleichberechtigt
verbindungsloses Protokoll	verbindungsorientiertes Protokoll
maximale Nachrichtenlänge: 484 Oktets	konzeptionell unbegrenzte Länge
nur bestätigte Dienste	bestätigte und unbestätigte Dienste
keine Sicherheit	Sicherheit geplant
keine speziellen Filtermechanismen	Scoping und Filter, 'beliebig' komplex
MOs nur für Netzressourcen	MOs für beliebige Ressourcen
keine Managementfunktion	Definition eines Rahmenwerks für typische Managementfunktionen
nur direkte Beziehungsinformation über OIDs in Tabelleneinträgen	Beziehungen können über Attribute oder auch durch eigene MO Klassen dargestellt werden.
sehr weit verbreitet, allgemeine Akzeptanz	kaum Implementierungen, umstritten wegen Komplexität

4.5 ITU Telecommunications Management Network (TMN)

Die **ITU** (früher **CCITT**) ist der Dachverband der Telekommunikationsanbieter. Seine Geschäftsfelder haben sich sehr stark von reiner Sprach- und Datenkommunikation (in getrennten Netzen) hin zu „Intelligenten Netzen" und „Integrierten Diensten" gewandelt. Die ITU erkannte früh, dass eine Standardisierung des Managements dieser Netze notwendig ist und definierte so das **Telecommunications Management Network** (**TMN**):

> „... to support the management requirements of administration to plan, provision, install, maintain, operate and administer telecommunication network and services."

Das **Telecommunication Management Network** (**TMN**) der ITU-T, dem als Basis das OSI-Management zugrundeliegt, wurde speziell auf die Bedürfnisse

der Betreiber öffentlicher Netze zugeschnitten und soll ein integriertes Management dieser Netze unterstützen. Aus Sicht der zu managenden Dienstnetze ist TMN einfach ein **Overlay Network**, in dem 6 Management-Ebenen unterschieden werden:

1. **Business Management**,
2. **Customer Management**,
3. **Service Management**,
4. **Network Management**,
5. **Subnetwork Management**,
6. **Network Element Management**, das sich um das Management der **Network Elementes** in der untersten Ebene kümmert.

Ziele

Ziel bei der der Entwicklung war eine gute Skalierbarkeit, da Telekommunikationsnetzwerke in der Regel sehr groß sind mit tausenden von Komponenten, die wiederum aus einer Vielzahl von MOs bestehen. Der OSI-Ansatz ist hier besonders geeignet, da er durch seine objekt-orientierte Modellierungsweise einen Top-down-Entwurf und Wiederverwendung, sowie die Modellierung komplexer Ressourcen und standardisierten Austausch von Informationen ermöglicht. Das Domänenkonzept bietet zudem die Möglichkeit Verantwortungsbereiche festzulegen.

Architektur

Die TMN-Architektur orientiert sich an sog. **Funktionsblöcken** (**Function Blocks**), die auf einem oder verteilt auf mehreren Trägersystemen erbracht werden und zwischen denen Schnittstellen (**Reference Points**) definiert sind:

- **Telecommunication Network (TN):** Teilnetze auf der Ebene der Network Elements
- **Network-Element (NE), Network Element Function (NEF):** Komponente, die Teilnehmern Dienste zur Verfügung stellt
- **Operations System (OS), Operations System Function (OSF):** Komponente, die Management-Informationen verarbeitet; um TNs zu steuern, im OS findet die Datenanalyse und globale Steuerung statt
- **Mediation Device (MD), Mediation Function (MF):** Komponente, die Management-Informationen zwischen NE(F) und OS(F) weiterleitet; Daten sammeln, vermitteln, aufbereiten, selektieren und identifizieren mit Netzwerkkomponenten
- **Workstation (WS), Workstation Function (WSF):** Komponente für den TMN-Zugang für menschliche Benutzer (außerhalb des TMN)

- **Data Communications Network (DCN), Local Communications Network (LCN), Data Communications Function (DCF):** Komponente, die Kommunikation mit anderen TMN-Einheiten ermöglicht; Transportnetze für Managementinformationen

Da das TMN auf dem OSI-Management aufbaut, werden die OSI-Managementkonzepte übernommen. Auf generischer Ebene werden allerding TMN-MOCs definiert, die als Oberklasse für spezifische MOs quer über alle TK-Technologien, -dienste und -architekturen Anwendung finden (z.B. Netz, Netzzugangspunkt, Netzkomponenten, Übertragungsstrecken, Beziehungen zwischen Vermittlungsknoten).

Da das OSI-Management nur Kommunikationsaspekte aber keine APIs beschreibt, sind Produkte weitgehend nicht portierbar und proprietär. Architekturen wie CORBA und TINA-C versuchen dieses Problem mit Hilfe von OMG-Konzepten zu umgehen.

Beispiele für Objekt des TMN sind:

- Telefonnetze, ISDN, Mobilfunknetze, VPN, Datennetze
- Sprachendgeräte, Rechner, Vermittlungsrechner, Multiplexer, Übertragungskanäle
- Intelligente Netze, Dienste, Anwendungen, Benutzer, Anbieter
- Verzeichnissysteme, Vermittler
- Managementstationen, TMN selbst, nicht TMN-Managementstationen
- ...

TMN-Empfehlungen der ITU-T sind in vielen `M.xxx` Empfehlungen abgelegt, die von den Prinzipien bis zu den den Mangement Functions alles beschreiben.

In der Struktur des TNM ist das Datenkommunikationsnetz eine Spezialisierung des Telekommunikationsnetz. Es enthält Operatingsystems und Workstations.

Die Aufgliederung der TMN Architektur erfolgt dabei in 3 Teile:

- **Funktionale Architektur:** sie beschreibt allgemeine TMN Funktionskomponenten und deren Schnittstellen untereinander
- **Physische Architektur:** beschreibt physische Komponenten und Schnittstellen, die für TMN gebraucht werden
- **Informations-Architektur:** Anwendung des OSI Management Information Model auf TMN

Dabei bestehen folgende **Anforderungen an die Funktionale Architektur**:

- **Austausch** von Management-Informationen zwischen Telekommunikationsnetz und TMN
- **Konvertierung** von Management-Information von einem Format in ein anderes
- **Austausch** von Management-Informationen zwischen den Funktionskomponenten des TMN
- **Präsentation** der Management-Information für den Benutzer des Managementsystems
- **Sicherung** des Zugangs zu Management-Information für autorisierte Benutzer des Managementsystems

Das TMN wird in 6 funktionale Blöcke aufgeteilt:

- **Operations System Function (OSF)**
  Verarbeitung von Management-Information, um etwas „zu managen".
- **Network Element Function (NEF)**
  Repräsentiert die Elemente des Telekommunikationsnetzes und kommuniziert mit dem TMN.
- **Workstation Function (WSF)**
  Kommunikation und Präsentation zum Benutzer.
- **Data Communication Function (DCF)**
  Kommunikation zwischen den TMN Komponenten.
- **Mediation Function (MF)**
  Übergang zwischen NEF (oder QAF) und OSF: Übersetzung, Filterung, Komprimierung etc. von Management-Informationen.
- **Q Adapter Function (QAF)**
  Übergang zu Nicht-TMN-Komponenten.

Damit gibt es die in Abbildung 5 auf der nächsten Seite aufgezeigten Referenzpunkte in der funktionalen Architektur.

Die Physische Architektur ist mit Schnittstellen beschrieben:

- **Qx** (Anschluss einfacher Netzkomponenten)
- **Q3** (Anschluss komplexer Komponenten z.B. mit OSI Protokollen CMIS/CMIP oder FTAM)

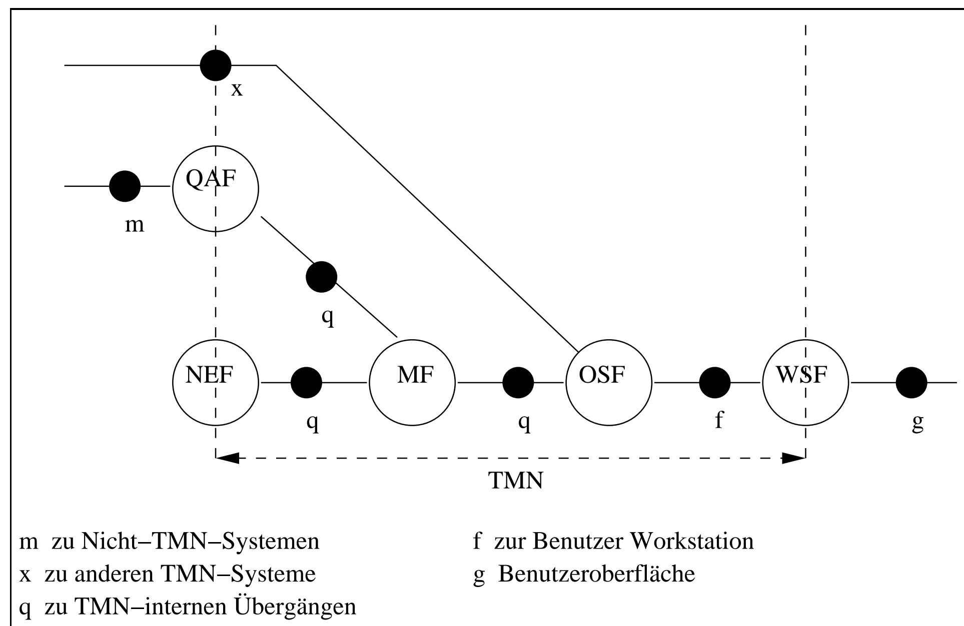

Abbildung 5: Referenzpunkte der funktionalen Architektur

- **X** (Anschluss anderer TMNs z.B. mit OSI-Protokollen, aber höhere Sicherheitsanforderungen)
- **F** (Kommunikation zur Workstation)

4.5.1 Informations-Architektur

Die **Informations-Architektur** basiert auf einer objektorientierten Modellierung wie im OSI Management, die in einer Management Information Base zusammengefasst wird. Dabei muss es eine 1:1 Zuordnung zwischen einer Ressource und einem MO geben – eine Ressource kann jedoch durch mehrere MOs repräsentiert werden.

Es können nur Ressourcen gemanaged werden, die nach aussen sichtbar sind. Damit TMN selbst gemanaged werden kann, muss es selbst MOs definierten. Beispiele für definierte MO-Klassen sind Network, Managed Element, Termination Point, Cross-Connection, Transmission Element.

4.5.2 Service Management

Der Begriff **Service** wird bei der ITU-T nochmal in frei Bereiche aufgeteilt:

- **bearer service:** reine Kommunikationsdienste (z.B. Sprachkanäle)

- **teleservice:** Anwendung, die auf Kommunikationsdiensten aufbauen (z.B. Videokonferenz)
- **supplementary service:** Zusatzdienste (z.B. für spezielle Anwendungsbereiche)

Für die **Quality of Service** (**QoS**), die beim Service Management eine entscheidende Rolle spielt, wurden ebenfalls einige Dinge festgelegt:

- Beispiele für QoS-Parameter in OSI-Standards (Durchsatz, Transitverzögerung, Fehlerrate etc.)
- Beispiele für QoS-Parameter in ATM-Netzen (durchschnittliche Ankunftsrate der Zellen, Spitzenrate etc.)
- Beispiele für QoS-Parameter bei Multimedia-Übertragungen (Bild- und Tonqualität etc.)

Dabei hat in geschichteten Architekturen jede Ebene eigene QoS-Parameter. Ausserdem hängt dort die Qos von Ebene n rekursiv von der QoS von Ebene $n-1$ ab.

Aufgaben den QoS Management

Zu den **Aufgaben des QoS Management** gehören:

- **Spezifikation und Abbildung**
  Beim Entwurf werden QoS-Parameter definiert und ihre Abhängigkeiten von darunterliegenden QoS-Parametern spezifiziert.
- **Verhandlung und Ressourcenzuteilung**
  Mit dem Dienstanbieter wird die Dienstgüte und die reservierte Ressourcen-Kapazität ausgehandelt.
- **Überwachung und Steuerung**
  Bei der Diensterbringung (d.h. zur Laufzeit) wird die QoS überwacht und ggf. gegengesteuert.
- **Statistik und Planung**
  Der Verlauf der QoS-Parameter wird statistisch erfasst, ausgewertet und in zukünftige Ressourcen-Planung mit einbezogen.

RACE Nemesys Projekt

Im **RACE Nemesys Projekt** wurde ein Prototyp eines Service Managers für Breitbandnetze entwickelt. Es war in mehrere Unterprojekte aufgeteilt und wurde mit dem Ziel **Service und Traffice Management in ATM-Netzen mit fortschrittlichen Informatik-Methoden** entwickelt.

ATM

Der **Asynchronous Transfer Mode** (**ATM**) besteht aus mehreren virtuellen Kanälen. Eine ATM-Zelle umfasst 48 Bit Nutzlast und 5 Bit für den Kopf. Auf ATM setzt wiederum das **B-ISDN Referenzmodell** auf, das aus den Schichten

- Management Pane (Control Pane, User Pane)
- Higher Layer Protocols
- ATM Adaption Layer
- ATM Layer
- Physical Layer

besteht.

Aufgaben des Nemesys Projekt waren:

- **Verkehrsfluss-Management** (**Traffic Management**): Rufannahme, Bandbreitenzuteilung für virtuelle Pfade
- **Dienst-Management** (**Service Management**): Dienst und Benutzeradministration, QoS Monitoring und Steuerung, Ereignisbehandlung und Statistik, Erstellung von Dienstprofilen, Lastevaluation und Vorhersage

Nemesys Benutzerdienste waren z.B. PCM Telefon, TASI Telefon, interaktiver Datentransfer, Mengendaten-Transfer, Video-Konferenz. Dagegen sind Nemesys Netzdienste z.B. konstante oder variable Bitrate Sprachkanal, Daten, Videokanal. Eine Video-Konferenz wurde z.B. 1 konst. Bitrate Sprache und einem Netzdienst Video zugewiesen. Wobei das System dann von jedem Dienst zwei zur Verfügung stellen muss.

QoS Parameter für Service Associations sind z.B. Aufbauverzögerung, Signalverzögerung, Signalverzögerungsvarianz, Abbauverzögerung, Signalqualität. Für ATM-Netzverbindungen sind QoS Parameter z.B. Zellen-Verlustrate, Aufbauverzögerung, Signalverzögerung, Signalqualität. Die Aufbauverzögerung wird dann z.B. dem Maximum aller Aufbauverzögerungen der einzelnen Verbindungen gesetzt.

5 Prüfungsfragen

- **Welche Transparenzarten unterstützt NFS?**
  Zugriffs-, Orts- und Mobilitätstransparenz. (lediglich eine Umkonfigurierung der Mount-Points notwendig). Weiter Features sind Skalierbarkeit (Probleme bei sog. „Hotspots“, dann besser Replikation wie bei Coda), Replikation (nur read-only auf mehreren Servern), Hardware- und Betriebssystemheterogenität, Fault Tolerance (Zustandslos und idempotent, soft und hard mounting), Konsistenz (durch regelmäßiges Update), Sicherheit (Kerberos) und Effizienz (read ahead, delayed write).

Literatur

[Tanenbaum1995] A. S. Tanenbaum: „Verteilte Systeme", Prentice Hall 1995.

[Coulouris et al. 2001] G. Coulouris, J. Dollimore, T. Kindberg: „Distributed Systems – Concepts and Design", Addison Wesley 2001.

[Hegering et al. 1999] H.G. Hegering, S. Abeck, B. Neumair: „Integriertes Managment vernetzter Systeme", dpunkt.verlag 1999.