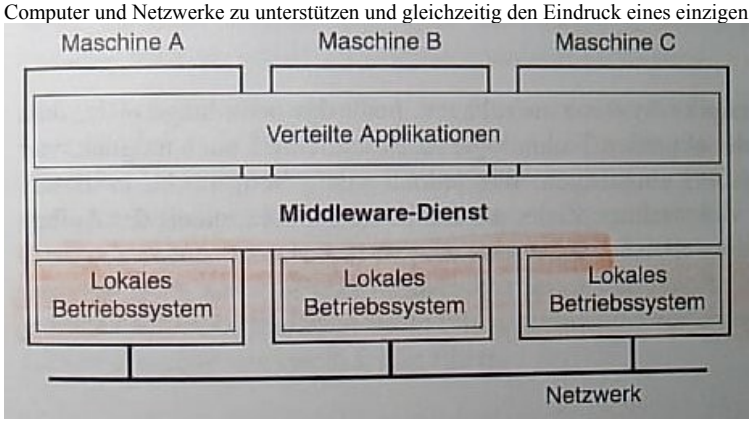
KE1

1.1 Einleitung

Def.:
Ein verteiltes System ist eine Menge voneinander unabhängiger Computer, die dem Benutzer wie ein einzelnes, kohärentes System erscheinen.
Eigenschaften:

- Unterschiede zwischen Computern und Kommunikation muß verborgen werden.
- Benutzer und Applikationen können auf konsistente und einheitliche Weise mit einem verteilten System zusammenarbeiten.
- Relativ einfach zu erweitern und zu skalieren

Implementierung einer Softwareschicht=**Middleware**, um heterogene
Computer und Netzwerke zu unterstützen und gleichzeitig den Eindruck eines einzigen Systems zu repräsentieren.



1.2 Ziele
Vier wichtige Ziele:

1. **Benutzer und Ressourcen verbinden**
   Billiger und sinnvoller (zB gemeinsame Nutzung von Supercomputern)
2. **Transparenz**
   Präsentation als ein einziges Computersystem

Transparenz	Verbirgt..
Zugriff	…Unterschiede in Datendarstellung (Little/Big Endian) und wie Zugriff auf Ressourcen
Position	..wo sich eine Ressource physisch befindet
Migration	..dass eine Ressource an andere Position verschoben werden kann.
Relokation	..dass eine Ressource während Nutzung an andere Position verschoben werden kann.
Replikation	…dass Ressource repliziert ist-Positionstransparenz erforderlich
Nebenläufigkeit	..dass eine Ressource von mehreren konkurrierenden Benutzern gleichzeitig genutzt werden kann.
Fehler	..den Ausfall und Wiederherstellung einer Ressource
Persistenz	..ob eine Software-Ressource sich im Speicher oder auf Festplatte befindet

   Transparenzgrad: Nicht immer sinnvoll, alle Aspekte zu verbergen, zB. Zeitzonenunterschiede, Verzögerungen durch Übertragung Auch ist es sinnvoller, Serverausfälle nicht zu maskieren, um wiederholte Kontaktversuche zu vermeiden.
3. **Offenheit**
   Ein offenes verteiltes System ist ein System, das Dienste den Standardregeln entsprechend anbietet, die die Syntax und die Semantik dieser Dienste beschreiben.
   Spezifikation in **IDL** (**Interface Definition Language**).
   Saubere Spezifikationen sind vollständig (alles Erforderliche wurde wirklich spezifiziert) und neutral (schreiben nicht vor, wie eine Implementierung auszusehen hat).
   Vollständigkeit und Neutralität sind erforderlich für Interoperabilität (Ausmaß, in dem zwei unterschiedliche Implementierungen von Systemen oder Komponenten nach gemeinsamem Standard zusammenarbeiten) und Portabilität (Ausmaß für Ausführung einer Applikation auf einem anderen System mit selber Schnittstelle).
   Trennung von Strategie und Mechanismus
   Flexibilität durch kleine austauschbare und anpassbare Komponenten, Schnittstellen nicht nur auf oberster Ebene, sondern auch zu internen Komponenten.
   Optimale Strategie nötig, je nach Benutzer und Applikation.
   Beispiel: Web-Caching: Benutzer sollten selbst (dynamisch) entscheiden können, wie lange Dokumente gespeichert werden.
4. **Skalierbarkeit**
   Skalierbar nach
   a. Größe (Ressourcen)
   b. Geographie
   c. Administrativ- einfach zu verwalten trotz mehrerer unabhängiger admin. Organisationen
   d.

Probleme/Beschränkungen bei der Skalierbarkeit:

a)**Größe**

Konzept	Beispiel	Nachteil
Zentrale Dienste	Einziger Server	Engpaß
Zentrale Daten	Einziges Online-Telefonbuch	Engpaß
Zentrale Algorithmen	Routing	Überlastung durch Nachrichten

b)**Geographie**

-Verzögerung in WAN

- Unzuverlässigkeit
- Ebenfalls Probleme mit Zentralität (ein Mailserver für ein ganzes Land

c)**administrativ**

Konflikte bei Nutzung der Ressourcen, Verwaltung und Sicherheit (erstreckt sich ein verteiltes System auf eine andere Domäne, müssen diese sich voreinander schützen).

Skalierungstechniken:

a. **Verbergen von Kommunikationslatenzzeiten-** durch asynchrone Kommunikation, Client kann andere Arbeiten (evtl auch serverspezifische, zB Überprüfen von Eingaben) erledigen bis zur Antwort des Servers

b. **Verteilung-** Komponenten werden zerlegt und über System verteilt. Bsp DNS: DNS-Namensraum als Domänenbaum, für jeden Teilnamen ist ein Server zuständig. Auch eine URL ist meist physisch über mehrere Server verteilt.

c. **Replikation-**Kopien, evtl geographisch näher.

cc. **Caching**: Besondere Form der Replikation, Kopie wird iA in Nähe des Clients angelegt, Unterschied zu Replikation: Client entscheidet darüber!

Bei beiden aber *Konsistenzprobleme*!

1.3 Hardwarekonzepte

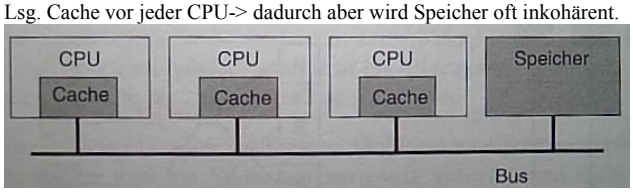
1.3.1 Multiprozessoren

Ein gemeinsam genutzter Speicher.

Kohärenter Speicher= Werte im Speicher sind immer aktuell.

Problem: Überlastung des Bus bei schon 4 Prozessoren.

Lsg. Cache vor jeder CPU-> dadurch aber wird Speicher oft inkohärent.



Problem bei Bussen: Beschränkte Skalierbarkeit.

Lösungen: **Kreuzungspunktschalter** (n² Kreuzungspunktschalter nötig) und **Omega-Netzwerk**

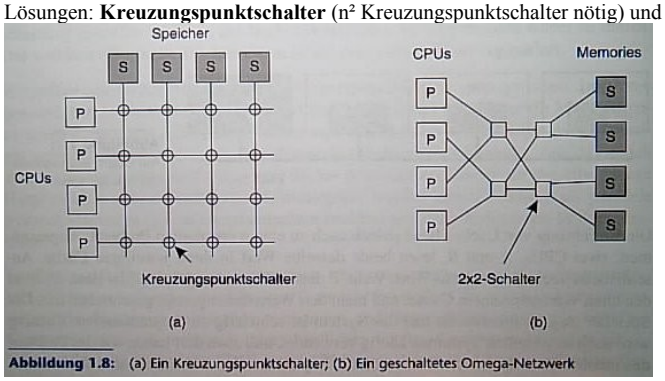


Abbildung 1.8: (a) Ein Kreuzungspunktschalter; (b) Ein geschaltetes Omega-Netzwerk

Hierarchische Systeme: Jeder CPU ist ein gewisser Speicher zugeordnet, Zugriff auf eigenen Speicher schnell, auf den anderer CPUs langsam (**NUMA-Maschine-NonUniform Memory Access**)

1.3.2 Homogene Multicomputersysteme

Nur ein verbindendes Netzwerk , überall dieselbe Technologie.

SAN (System Area Network): Knoten in Schrank und über ein einziges Netzwerk verbunden.

Schalterbasiert: Nachrichten zwischen CPUs werden geroutet (kein Broadcast). Topologien: Maschen und Hypercubes.

1.3.3 Heterogene Multicomputersysteme

Computer unterscheiden sich in Prozessortyp, Speichergrößen und I/O-Bandbreiten, evtl auch in Verbindungsnetzwerken.

1.4 Softwarekonzepte

1.4.1 Verteilte Betriebssysteme

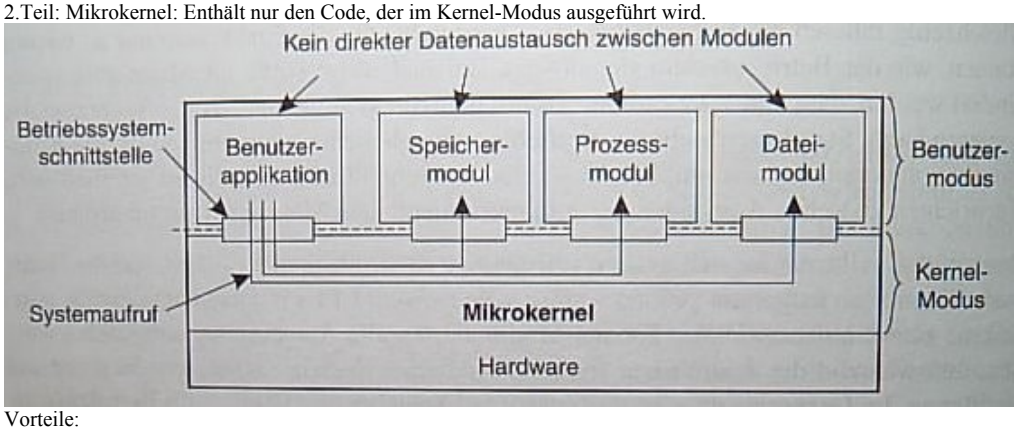
Zwei Typen: Multiprozessor-BS und Multicomputer-BS

System	Beschreibung	Wichtigstes Ziel
DOS (Distributed Operating System)	Streng gekoppeltes BS für Multiprozessoren und homogene Multicomputer	Hardware-Ressourcen verbergen und verwalten
NOS (Network Operating System)	Locker gekoppeltes BS für heterogene Multicomputer (LAN und WAN)	Anbieten lokaler Dienste für entfernte Clients
Middleware	Zusätzliche Schicht über dem NOS, die allgemeine Dienste implementiert.	Verteilungstransparenz erzielen.

Einprozessor-BS:

Zusammenfassung 1678 Verteilte Systeme

BS implementiert virtuelle Maschine, so dass es Applikationen erscheint, als hätten sie ihre eigenen Ressourcen. Anwendungen werden voneinander geschützt.
Zwei Betriebsmodi:
Kernel-Modus: Alle Anweisungen dürfen ausgeführt werden.
Benutzermodus: Einschränkungen beim Zugriff auf Speicher und Register.
Ausführung von BS-Code nur im Kernel-Modus.
Einzige Möglichkeit vom Schalten aus Benutzermodus in Kernel-Modus: Systemaufrufe (vom BS implementiert).
Monolithische BS (üblich und verbreitet) führen Code innerhalb eines einzigen Adressraumes aus, bei Anpassungen Herunterfahren oder sogar Neukompilierung nötig.
Flexibler: BS in 2 Teile anordnen:
1.Teil: Verwaltung der Hardware im Benutzermodus (nur MMU-Registersetzen im Kernelmodus)
2.Teil: Mikrokernel: Enthält nur den Code, der im Kernel-Modus ausgeführt wird.



Vorteile:

- Flexibilität- Module leicht ersetzbar
- Module auf Benutzerebene können leicht auf anderen Maschinen platziert werden

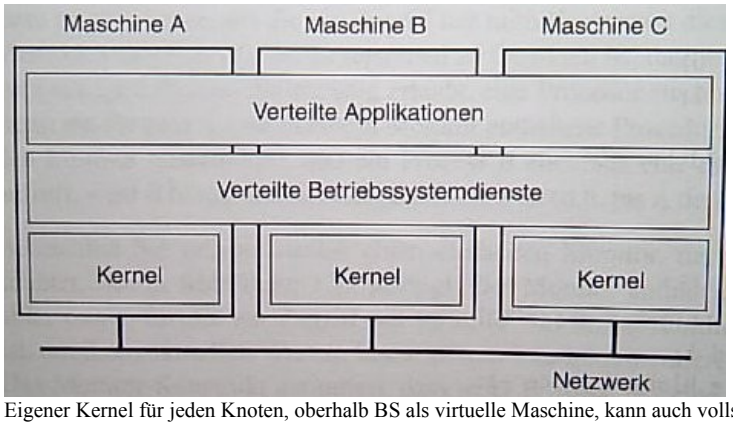
Nachteile:

- Unterscheidung zu aktuellen BS
- Zusätzliche Kommunikation=Leistungsverlust

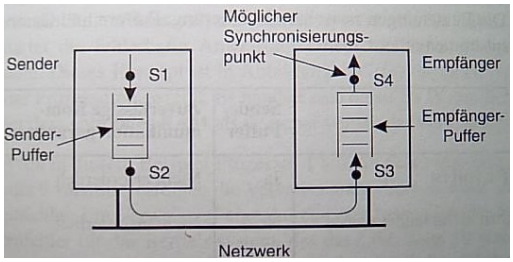
Mehrprozessor-Betriebssysteme
Alte BS enthalten nur einen einzigen Steuerthread, so dass mehrere CPUs nicht unterstützt werden können, moderne BS können dies und zwar transparent.
Nötig: Konsistenzgarantie durch Schutz der gemeinsam genutzten Daten gegen gleichzeitigen Zugriff:

- Durch **Semaphor:** Besteht aus int count und Prozessmenge W. Ops: down und up. Count= Zahl der freien Betriebsmittel, will ein Prozeß ein Betriebsmittel, ruft er down auf, ist Count>0 so dekrementiert er Count und bekommt das Mittel, ist Count=0, so wird er zur Prozessmenge W hinzugefügt. Up wird aufgerufen, wenn Prozeß frei wird und inkrementiert Count, Prozeß aus W kann dann wieder down aufrufen.
- Durch **Monitor:** (Stichwort synchronized): Führt Prozeß A eine im Monitor enthaltene Prozedur aus (A tritt in den Monitor),so wird B, der auch diese Prozedur ausführen will, blockiert, bis A fertig ist. Realisierung durch Sprache, nicht durch den Programmierer.

Multicomputer-Betriebssysteme



Eigener Kernel für jeden Knoten, oberhalb BS als virtuelle Maschine, kann auch vollständige Software-Implementierung des gemeinsam genutzten Speichers bereitstellen (ohne dies können den Applikationen nur Funktionen für Nachrichtenübergabe bereitgestellt werden.).
4 Blockierpunkte und 2 Pufferpunkte möglich:



S1: Sendepuffer (expl. erforderlich) ist voll, keine zuverlässige Kommunikation nötig
S2: Nachricht wurde gesendet, kein Sendepuffer und keine zuverlässige Kommunikation nötig
S3: Nachricht angekommen, Sendepuffer nicht sinnvoll, zuverlässige Kommunikation. nötig!
S4: Nachricht ausgeliefert, Sendepuffer nicht sinnvoll, zuverlässige Kommunikation. nötig!

Verteilte Systeme mit gemeinsam genutztem Speicher

DSM (Distributed Shared Memory): seitenbasierter verteilter gemeinsam genutzter Speicher- Paging mit entferntem Speicher.
Verweist CPU1 auf Seite 10: *Seitenfehler*- BS verschiebt dann Seite 10 von Maschine 2 auf Maschine 1:

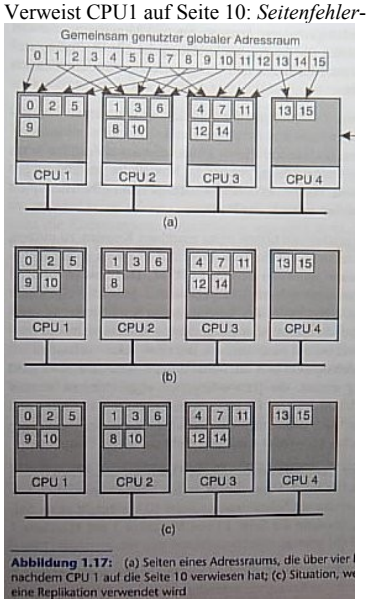
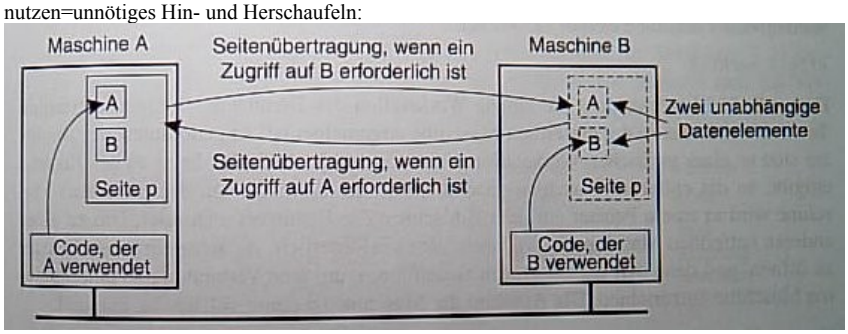


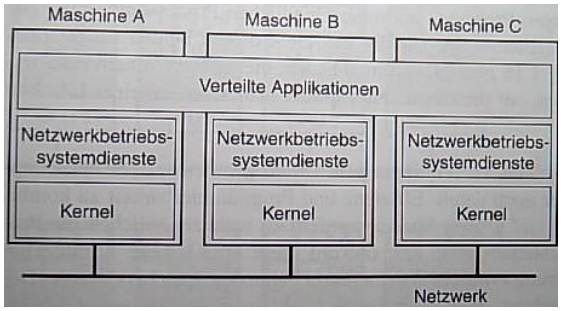
Abbildung 1.17: (a) Seiten eines Adressraums, die über vier
nachdem CPU 1 auf die Seite 10 verwiesen hat; (c) Situation, w
eine Replikation verwendet wird

Verbesserung der Leistung durch schreibgeschützte Replikationen- noch besser ist, alle Seiten zu replizieren, dann Konsistenzmaßnahmen (zB vor Schreiben alle Kopien ungültig machen).
Kosten, Übertragung einzurichten bestimmt die Transportkosten, nicht die Datenmenge! Also möglichst große Seiten wählen.
Andererseits kann es damit zu falscher gemeinsamer Nutzung kommen, wenn 2 unabhängige Prozesse Datenelemente einer Seite gemeinsam nutzen=unnötiges Hin- und Herschaufeln:



1.4.2 Netzwerkbetriebssysteme

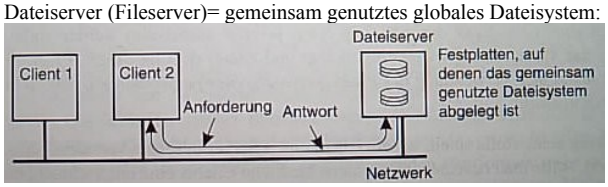
Maschinen und BS können sich unterscheiden und sind nur durch ein Netzwerk miteinander verbunden:



Mögliche Dienste:

- Login bei entfernter Maschine rlogin machine
- Kopieren rpc machine1:datei1 machine2:datei2

Dateiserver (Fileserver)= gemeinsam genutztes globales Dateisystem:



Dateiserver haben hierarchische Dateisysteme (Hauptverzeichnis enthält untergeordnete Verzeichnisse und Dateien). Clients können diese importieren und in ihre eigene Dateistruktur beliebig –an unterschiedlichen Positionen- installieren.
Nachteil von Netzwerkbetriebssystemen:

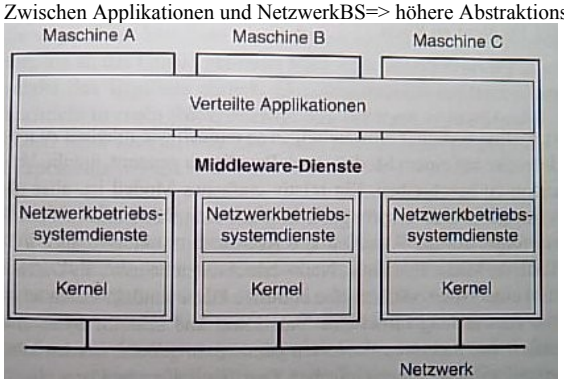
- Nicht transparent
- Änderung von Zugriffsberechtigungen und Passwörtern muß auf jeder Maschine einzeln ausgeführt werden

Vorteile:

- Einfaches Hinzufügen (Internet: Mit DNS-Eintrag) und Entfernen von Maschinen

1.4.3 Middleware

Sinn: Heterogenität der verschiedenen Plattformen verbergen und Verteilungstransparenz verbessern.
Positionierung
Zwischen Applikationen und NetzwerkBS=> höhere Abstraktionsebene



Heterogenität vor Anwendungen verbergen: Middleware-Systeme bieten vollständige Sammlung an Diensten an, soll nicht umgangen werden.
Middleware-Modelle

- Middleware basierend auf verteilten Dateisystemen: Verteilungstransparenz nur für traditionelle Dateien->gut skalierbar
- Middleware basierend auf RPCs
- Middleware basierend auf verteilten Objekten

Middleware-Dienste

- Kommunikationsfunktionen, die die Low-Level-Nachrichtenübergabe verbergen
- Namensgebung (vgl Telefonbuch)
- Persistenz (spez. Funktion zum Speichern)
- Verteilte Transaktionen
- Funktionen für die Sicherheit

Middleware und Offenheit
Moderne verteilte Systeme sind iA als Middleware für ganzen Bereich von BS aufgebaut.
Unabhängigkeit der Appl vom BS erkauft durch Abhängigkeit von der Middleware
Unvollständige SS-Definitionen führen zu eigens eingeführten SS=>nicht mehr portabel von einem aufs andere System
Vergleich zwischen den Systemen

Aspekt	Verteilte BS		NetzwerkBS	Middleware-basiertes BS
	Multiprozessoren	Multicomputer		
Transparenzgrad	Sehr hoch	Hoch	Gering	Hoch

Dasselbe BS auf allen Knoten	Ja	Ja	Nein	Nein
Anzahl der Kopien des BS	1	N	N	N
Kommunikationsbasis	Gemeinsam genutzter Speicher	Nachrichten	Dateien	Modellspezifisch
Ressourcenverwaltung	Global,zentral	Global,verteilt	Pro Knoten	Pro Knoten
Skalierbarkeit	Nein	Moderat	Ja	Variiert
Offenheit	Geschlossen	Geschlossen	Offen	Offen

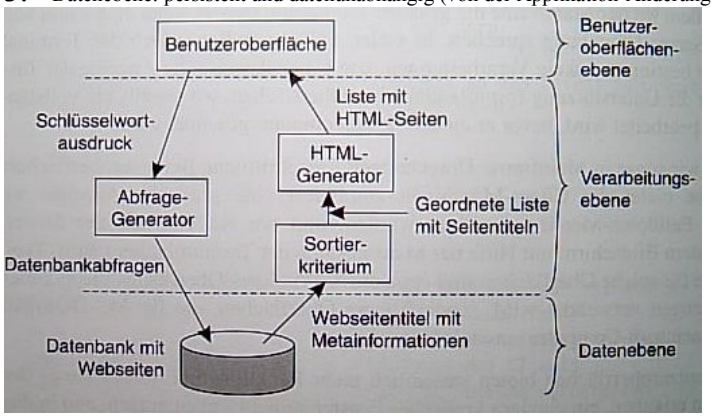
1.5 Client-Server-Modell

Client= Prozeß, der Dienst von einem Server anfordert, indem er Anforderung sendet und auf Antwort wartet.

	Verbindungsloses Protokoll:	
Vorteil	Effizienz	Zuverlässigkeit
Nachteil	Anforderung/Antwort kann verlorengehen	Einrichtung kostspielig

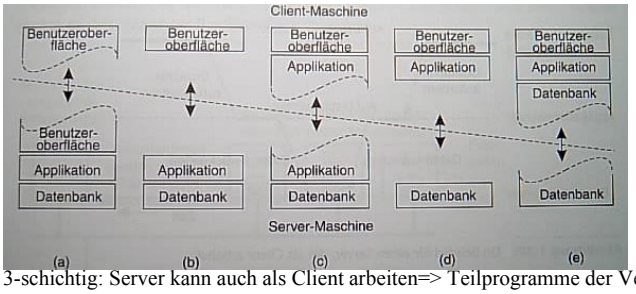
1.5.2 Anwendungsschichten

1. Ebene der Benutzeroberfläche (normal von Clients implementiert)
2. Verarbeitungsebene (enthält iA Kernfunktionalität einer Applikation, Verbindung Oberfläche-Datenbank)
3. Datenebene: persistent und datenunabhängig (von der Applikation-Änderungen haben keine Einfluß auf Applikation)

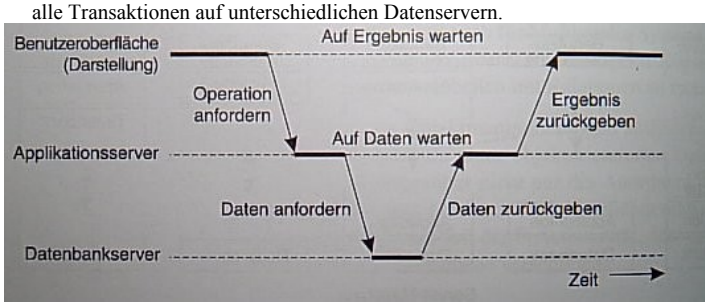


1.5.2 Architekturen

- Einfach: Zwei Maschinentypen-Client:Oberfläche, Server: Rest. Problem: Keine wirkliche Verteilung, Client=dummer Arbeiter
- lMehrschichtig (Multitier)
  - 2-schichtig

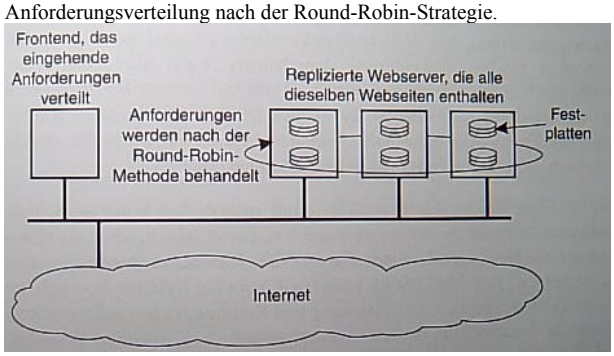


  - 3-schichtig: Server kann auch als Client arbeiten=> Teilprogramme der Verarbeitungsebene auf separatem Server. Bsp: Transaktionsverarbeitung->separater Prozeß (Transaktionsmonitor)-kontrolliert alle Transaktionen auf unterschiedlichen Datenservern.



- Moderne Architekturen:

- Vertikale Verteilung: logisch unterschiedliche Komponenten auf unterschiedlichen Maschinen.
  Begriff von „vertikaler Fragmentierung“in verteilten relationalen Datenbanken: Tabellen nach Spalten unterteilt und auf unterschiedliche Maschinen verteilt.
- Horizontale Verteilung: Client/Server physisch in logisch äquivalente Teile unterteilt, jeder Teil mit vollständigerDatenmenge zB mehrere Webserver
  Anforderungsverteilung nach der Round-Robin-Strategie.



Clientverteilung: Auch möglich, dass es keinen Server gibt (Zusammenarbeit von einfachen Anwendungen)=**Peer-to-Peer**-Verteilung.

7.Fehlertoleranz

Grundlegende Konzpte

Anforderungen an **verlässliche Systeme**:

1. **Verfügbarkeit:** Wahrscheinlichkeit, dass ein System zu einem best. Zeitpunkt korrekt funkioniert
2. **Zuverlässigkeit:** Eigenschaft, dass ein System fortlaufend fehlerfrei ausgeführt wird (definiert für Zeitintervall)
3. **Sicherheit:** System funktioniert nicht->nichts Schlimmes darf passieren
4. **Wartbarkeit:** Schwierigkeit der Reparatur

Fehlermodelle

Fehlertyp	Beschreibung
Absturzfehler	Korrekt funktionierender Server wird unterbrochen
Auslassungsfehler Empfangsauslassung Sendeauslassung	Server reagiert nicht auf Anforderungen Server erhält keine eingehenden Anforderungen Server sendet keine Nachrichten
Timing-Fehler	Antwortzeit eines Servers liegt außerhalb eines festgel. Intervalls
Antwortfehler Wertfehler Statusübergangsfehler	Antwort des Servers ist falsch Wert der Antwort ist falsch Server weicht vom korrekten Steuerfluß ab
Zufälliger Fehler	

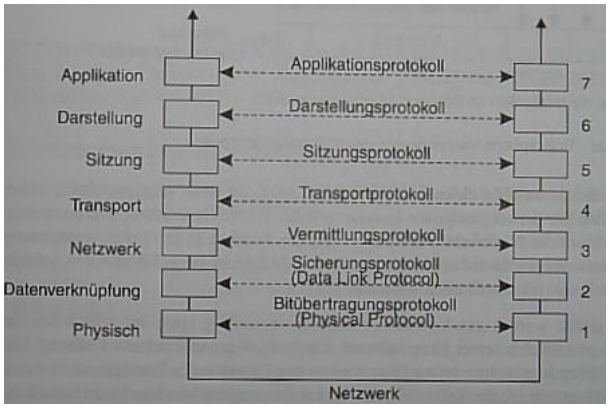
Fehlermaskierung durch Redundanz

3 Arten:

1. Informationsredundanz: zusätzliche Bits werden eingefügt (zB Hamming-Code zur Wiederherstellung bei Rauschen)
2. zeitliche Redundanz: Aktion wird ausgeführt und ggf wiederholt (zB Transaktionen)
3. Physische Redundanz: Zusätzliche Ausrüstung oder Prozesse zur Kompensation von Fehlern (Zwei Augen, 4 Triebwerke)
   TMR (Triple Modular Redundancy-3fach Entscheider**):** 3 Eingänge, mind. 2 müssen übereinstimmen, damit Ausgabe zustande kommt

KE2 Geschichtete Protokolle

Verbindungsorientierte Protokolle: Verbindung einrichten, über Protokoll einigen, abbauen
Verbindungslose Protokolle: Keine Verbindungseinrichtung erforderlich
7 Schichten (Die verschiedenen Protokolle in einem System heißen **Protokollstapel**):



Unterste Schicht

Vorteil der Schichten: Jede kann ausgetauscht/modernisiert werden, ohne dass die anderen Schichten geändert werden müssen!

1. Bitübertragungsschicht: Aufgabe: 0 und 1 übertragen- wie viel Volt für 0 und 1, wie viele Bit pro Sekunde, Übertragung in beide Richtungen gleichzeitig. Protokoll: Standardisierung der SS, so dass 0 und 1 auch auf anderen Maschinen 0 und 1 sind.
2. Sicherungsschicht: Gruppierung der Bits in Einheiten (**Frames**). Am Anfang und Ende jedes Frames spez. Bitmuster (Addition aller Bytes)=**Prüfsumme**. Stimmt diese nicht überein, wird der Frame x (alle Frames sind nummeriert) erneut angefordert.
3. Vermittlungsschicht: Wichtigste Aufgabe: **Routing.** Routing-Algos versuchen, beste Route zu ermitteln (Verzögerung, Verkehrsaufkommen…).
   Protokolle: IP (verbindungslos), Virtueller Channel (verbindungsorientiert-ATM-Netzwerke)

Transportprotokolle

Konzept: Ermöglicht der Applikationsschicht, eine Nachricht an die Transportschicht weiterzugeben, mit der Sicherheit, dass diese ohne Verluste ausgeliefert wird=>dazu zerlegt sie diese in kleine Teile und weist ihnen eine laufende Nummer zu. Da diese beim Empfänger in unterschiedlicher Reihenfolge ankommen können, ist es Aufgabe der Transportschicht, diese in die richtige Reihenfolge zu bringen.
TCP (Transmission Transport Protocol)-verbindungsorientiert, **UDP** (Universal Datagram Protocol)-verbindungslos, entspricht im Wesentlichen IP+kleine Zusatzfunktionen. **RTP** (Realtime Transport Protocol)-Framework-Protokoll, spezifiziert Paketformate für Echtzeitdaten ohne garantierte Datenauslieferung.

Client-Server-TCP

Normale TCP-Arbeitsweise: Initiierung mit 3-Phasen-Handshake (Einigung über Folgenummerierung der Pakete), dann Anforderung etc., viel Aufwand für Verwaltung der Verbindung.
Deshalb T/TCP (Transaktionales TCP): Billiger, da Initiierung gleich mit Senden der Anforderung und die Antwort mit dem Schließen der Verbindung kombiniert wird:

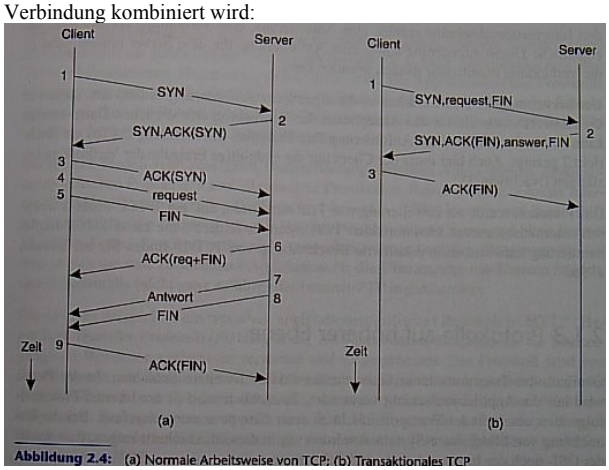


Abbildung 2.4: (a) Normale Arbeitsweise von TCP; (b) Transaktionales TCP

Protokolle auf höherer Ebene

Sitzungs- und Darstellungsprotokoll

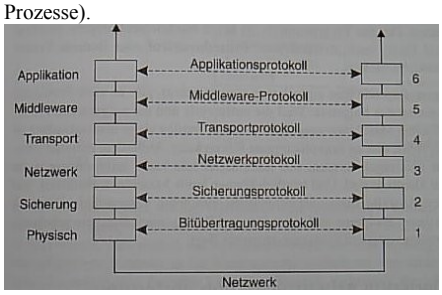
Sitzungsschicht: Wird selten unterstützt, Erweiterung der Transportschicht, unterstützt Dialogsteuerung, hat Funktionen zur Synchronisierung (Prüfpunkte für lange Übertragungen, bei Absturz nur Rückkehr zum letzten Prüfpunkt)
Darstellungsschicht: Beschäftigt sich mit der Bedeutung der Bits: Hier können bestimmte Datensätze mit Feldern definiert werden, die dem Empfänger das Format mitteilen.

Applikationsprotokolle

Protokolle: FTP (File Transfer Protocol)-Übertragung von Dateien von Client- auf Servermaschine, http (HyperTextTransferProtocol)-Übertragung von Websites, auch von Java RMI verwendet für Aufruf von entfernten, durch Firewall geschützten Objekten (http-Tunneling)

MiddlewareProtokolle

Middleware ist eine Applikation, die sich logisch auf der Applikationsschicht befindet, aber viele allgemeine Protokolle enthält, die ihre eigenen Schichten rechtfertigen, zB. **Authentifizierung** (Beweis für Identität), **Autorisierung** (Berechtigung für Ressourcen),**Commit-**(in Gruppe von Prozessen haben entweder alle Zugriff, oder keiner),**Sperrprotokolle** (Schutz gegen gleichzeitigen Zugriff durch mehrere Prozesse).



2.2 Entfernte Prozeduraufrufe (RPC,Remote Process Call)

RPC:Prozess auf Maschine A ruft Prozedur auf Maschine B auf=>keine Nachrichtenübergabe für den Programmierer sichtbar.

Client- und Server-Stubs

RPC soll transparent sein- nicht zu erkennen, dass Prozeduraufruf nicht lokal ist, zB Aufruf von read: Bei entfernter Prozedur wird eine andere Version von read, ein Client-Stub, aufgerufen.
Schritte:

1. Client-Prozedur ruft Client-Stub auf
2. Client-Stub erzeugt Nachricht und ruft lokales BS auf
3. BS Client sendet Nachricht an BS Server
4. BS Server gibt Nachricht an Server-Stub
5. Server-Stub entpackt Parameter+Aufruf Server
6. Server rechnet, Rückgabe des Ergebnisses an Stub
7. Server-Stub verpackt Ergebnis in Nachricht, Aufruf BS
8. BS Server gibt Nachricht an BS Client
9. BS Client gibt Nachricht an Client Stub
10. Stub entpackt Ergebnis, Rückgabe an Client

2.2 Parameterübergabe

Marshaling=Verpacken der Parameter in eine Nachricht
Probleme: Unterschiedliche Zeichencodes, Integerdarstellung, Little/Big Endian

Übergabe von Referenzparametern

Problem hier: Adresse nur gültig für Client.
Lösung:

Ersetzen von call-by-reference durch copy/restore=>Objektübergabe in Nachricht

Bei komplexen Objekten Zeigerübergabe an Server-Stub=>Rückanforderung der Daten an Client

Parameterspezifikation und Stub-Erstellung

Bei RPC sollten beide dasselbe Protokoll verwenden
Einigkeit über

- Definition des Nachrichtenformats
- Darstellung einfacher Datenstrukturen
- Transportdienst (verbindungsorientiert/-los)

Implementation der stubs:
Häufig nur Unterschied in der Schnittstelle zu den Applikationen.
Spezifikation der SS mithilfe einer IDL (Interface Definition Language), eine derart spezifizierte SS wird dann zu einem Client- oder Server-Stub compiliert.

Erweiterte RPC-Modelle

Doors*:Kompormiß zu RPCs, die sich zusammen auf derselben Maschine befinden-schneller als RPC.*

Door =generischer Name für Prozedur im Adressraum eines Serverprozesses, die von Prozessen aufgerufen werden kann, die sich auf derselben Maschine wie der Server befinden.
Eine Door muß vom Server registriert werden, Rückgabewert ist eine ID, dieser ID kann ein Name zugeordnet werden.

Client ruft auf mit Systemaufruf door-call(id,..parameter..)->BS ruft einen weiteren Server-Prozeß auf, der die Door registriert hat.
Wichtigster Vorteil: Doors erlauben die Verwendung eines einzigen Mechanismus zur Kommunikation in einem verteilten System
Nachteil: Applikationsentwickler muß wissen, ob Aufruf lokal innerhalb des Prozesses, lokal für anderen Prozeß oder für entfernten Prozeß erfolgt.

Asynchroner RPC

Client blockiert nicht, bis Antwort vom Server kommt, sondern bekommt sofort eine Bestätigung vom Server und fährt fort. Bsp: Überweisung->nicht nötig, auf Antwort zu warten.
Verzögerter synchroner RPC: 2 Aufrufe: Client-Anforderung|Bestätigung|…|Server-Antwort|Client-Bestätigung
Einwege-RPC:Client wartet nicht auf Bestätigung,->kann sich auch nicht sicher sein, dass Server die Anforderung bekommen hat

2.2.4 Beispiel DCE-RPC

DCE= Distributed Computing Environment, entwickelt von OSF (Open Software Foundation)
DCE=Middleware, zugrunde liegend das Client-Server-Modell.
Dienste als Teile von DCE:

- Verteilter Dateidienst: transparente Möglichkeit, auf dieselbe Weise auf beliebige Dateien im System zuzugreifen
- Verzeichnisdienst: zum Beobachten aller Ressourcen (Maschinen, Drucker, Daten…) im System. Erlaubt einem Prozess, Ressourcen anzufordern, ohne zu wissen wo.
- Sicherheitsdienst: Einschränkung des Zugriffs
- Zeitdienst: Synchronisierung der Uhren

Ziele von DCE-RPC

1. Ermöglicht Client Zugriff auf Remotedienst, nur durhc Aufruf von lokaler Prozedur.
2. Einfach, große Mengen existierenden Codes in verteilter Umgebung auszuführen,, ohne große Änderungen vornehmen zu müssen.

Binden=Kommunikationseinrichtung zw. Server und Client
RPC-System verbirgt Details vor den Clienten:

- Findet automatisch den richtigen Server und richtet Kommunikation ein
- Kann Nachrichtenübertragung in beide Richtungen vornehmen+Zerlegen/Zusammensetzen
- Autom. Datentypumwandlungen zw. Client und Server, auch bei unterschiedl. Architekturen und Endians.
- ⇨ Client und Server extrem unabhängig voneinander iBa Sprachen, Hardware, BS,Unterstützung von vielen Protokollen.

Client und Server schreiben

IDL (Interface Definition Language) definiert die Syntax der Aufrufe (nicht die Semantik)
In IDL-Datei: Global eindeutige Bezeichner für die spezifizierte SS, bestehend aus 1. Position, 2. Erstellungsdatum
Schritte bei der Entwicklung einer Client-Server-Applikation:

1. uuidgen aufrufen->Erzeugung einer Prototyp-IDL-Datei mit SS-ID=eindeutig
2. IDL-Datei bearbeiten: Namen der entfernten Prozeduren+Parameter eintragen
3. IDL-Compiler aufrufen, Ausgabe:
   - ⇨ Header-Datei mit eindeutiger ID, Definition von Typ u. Konstanten, Funktionsprototypen
   - ⇨ Client-Stub enthält Prozeduren, die das Client-Programm aufruft: Parameter sammeln, Nachrichten verpacken/versenden
   - ⇨ Server-Stub enthält Prozeduren, die vom Laufzeitsystem auf der Server-Maschine bei Nachricht aufgerufen werden können
4. Appl. Entwickler schreibt Client-/Servercode
5. Client/Server+Stubs compilieren
6. Binden der Client und Client-Stub-Objektdateien zu Laufzeitbibliothek, erzeugt ausführbare Programmdatei,analog Server

Client zu Server binden

Erforderlich: Registrierung des Servers+Vorbereitung auf eingehende Anrufe
Schritte:

1. Suchen der Servermaschine
2. Suchen des Servers (korrekten Prozesses)

DCE-Dämon: Prozess in DCE, der eine Tabelle mit (Server-,Port)-Paaren verwaltet.
Dann Registration dieses Ports beim DCE-Dämon und im Verzeichnisdienst (wie DNS)

RPC ausführen

Client verpackt Nachricht unter Verwendung des beim Binden gewählten Protokolls.
Beim Server wird sie abhängig vom Port an den korrekten Server weitergeleitet.
Weitergabe an Server-Stub, dieser entpackt Parameter und ruft Server auf.
DCE-Optionen:

- ⇨ **Höchstens-einmal-Operation** (Standard): Nur ein Aufruf, auch bei Serverabsturz
- ⇨ Markierung einer entfernten Prozedur als **idempotent**: Mehrfache Wiederholung ohne Probleme möglich.
- ⇨ **Broadcasting**

2.3 Entfernter Objektaufruf

Wichtig für verteilte Systeme: SS auf einer Maschine, Objekt selbst auf der anderen.

Bindet sich ein Client zu einem verteilten Objekt, wird ein Proxy=Implementierung der Objekt-SS in den Adressraum des Clients geladen. Proxy ähnlich zu Client-Stub in RPC.

Arbeit des Proxy: Verpackt Methodenaufrufe in Nachrichten, entpackt Antwort, Rückgabe des Ergebnisses an Client

Skeleton=Server-Stub entpackt Nachrichten vom Proxy+retour.

Charakteristisch für verteilte Objekte: Status nicht verteilt, befindet sich auf einer einzigen Maschine.

Compile- oder Laufzeitobjekte

Compilezeitobjekte: von objektorientierten Sprachen unterstützte Objekte- Einfach, verteilte Applikationen zu erstllen, aber abhängig von Programmiersprache
Laufzeitobjekte: Erzeugung der Objekte zur Laufzeit- Applikation kann aus unterschiedlichen Sprachen zusammengesetzt werden.
Wichtig: Implementierungen mpssen als Objekt erscheinen, Hilfsmittel hier: Objektadapter (bilden Hülle um das Objekt, um ihm das Erscheinungsbild eines Objekts zu geben)-> binden sich dynamisch zu Implementierung und stellen aktuellen Status dar.

Persistente und transiente Objekte

Persistentes Objekt: Existiert auch unabhängig vom Adressraum des Servers (ist im sekundären Speicher)
Transientes Objekt: Existiert nur solange wie der Server.

2.3.2 Client zu Objekt binden

Unterschied zu RPC-Systemen und normalen, die verteilte Objekte unterstützen: Letztere unterstützen systemübergreifende Objektreferenzen (zB für Parameterübergabe).
Implizites Binden: Client wird Mechanismus bereitgestellt für Aufruf der Methoden unter Verwendung nur einer Referenz (zB durch überladene Methoden)

Implementierung von Objektreferenzen

Objektreferenz muß beinhalten: Adresse, Port des Servers und Objektangabe. Besser als Adresse ist aber Adresse eines Positionsservers (Dämon)-flexibler.
Auch Implementierungshandle kann in Referenz enthalten sein->verweist (ZB als URL) auf Implementierung eines Proxies, das der Client dynamisch laden kann.

2.3.3 Statische im Vergleich zu dynamischen entfernten Methodenaufrufen

Nach Binden Methodenaufruf möglich: RMI (Remote Method Invocation).
Unterschied zu RPC: RMI unterstützen systemübergreifende Objektreferenzen.
Statischer Aufruf über vordefinierte SS-definitionen: Müssen bekannt sein: obj.methode()
Dynamischer Aufruf: Applikation wählt zur Laufzeit, welche Methode sie auswählt (invoke (obj,id(methode)…)) –nötig zB bei Benutzereingabe

2.3.4 Parameterübergabe

Bei entfernten Objekten wird die Referenz übergeben, bei lokalen die Objektkopie als Wert.

2.3.5 Beispiel1: Entfernte DCE-Objekte

Einzige Möglichkeit zur verteilten Objekterzeugung: Implementierung auf einem Server, der dafür verantwortlich ist, C++-Objekte lokal zu erzeugen und Methoden für entfernte Client-Maschinen bereitzustellen.
Zwei Typen:

1. Verteilte dynamische Objekte: Server erzeugt Objekt lokal für einen Client, Zugriff nur durch diesen.
2. Verteilte benannte Objekte: Gemeinsame Nutzung von mehreren Clients

Entfernter Objektaufruf in DCE: Durch RPC, Parameter Objekt-ID, SS-ID der Methode usw.
Kein Mechanismus für transparente Objektreferenzen,kann höchstens Bindle-Handle (enthält ID der SS des Objekts, das Transportprotokoll und Serveradresse+Port) als String übergeben.

2.3.6 Beispiel 2. Java RMI

Unterschiede zwischen entfernten und lokalen Objekten:

1. entfernte und lokale Objekte werden auf unterschiedliche Weise geklont-nur Server kann entfernte Objekte klonen, Clients müssen sich neu daran binden.
2. Semantik beim Blockieren: Blockieren in Java nur über Proxies- Sperrtechniken nötig.

Entfernter Objektaufruf in Java: Lokale Objekte werden kopiert und als Wert übergeben, entfernte als Referenz. Entfernte Objekte setzen sich aus 2 Klassen zusammen: Serverseitiger Code und Clientseitiger, diese enthält Implementierung eines Proxies. Einfachste Form: Proxy wandelt nur jeden Methodenaufruf in eine Nachricht um, die an serverseitige Implementierung gesendet wird.

Proxy ist in Java serialisierbar, kann also als Referenz auf entferntes Objekt benutzt werden. Beim Verpacken eines Proxies wird letztlich ein Implementierungshandle erzeugt, der genau angibt, welche Klassen benötigt werden, um den Proxy zu erzeugen-Code muß dann evtl heruntergeladen werden.
Status wird beim Binden an einen Client kopiert, zur Konsistenzsicherstellung prüft jeder Aufruf, ob sich Status auf dem Server geändert hat, ggf Aktualisierung

Nachrichtenorientierte Kommunikation

2.4.1 Persistenz und Synchronität in der Kommunikation

- Persistente Kommunikation: Nachricht wird vom Kommunikationssystem solange gespeichert, bis sich ausgeliefert werden kann.
- Transiente Kommunikation: Nur solange Speicherung, wie sendende und empfangende Applikation ausgeführt werden.
- Asynchrone Kommunikation: Sender wird unmittelbar fortgesetzt, nachdem er seine Nachricht zu Übertragungen weitergegeben hat.
- Synchrone Kommunikation: Sender wird blockiert, bis eine Nachricht in einem lokalen Puffer auf dem empfangenden Host abgelegt ist oder an Empfänger ausgeliefert wurde.

Kombinationen:

- Persistent asynchron: Speicherung persistent auf lokalem Host oder 1.Kommunikationsserver (email)
- Transient asynchron: Datagramm-Dienste auf Transportebene zB. UDP oder Einwege-RPCs
- Transient synchron: Sender blockiert, bis Bestätigung über Speicherung oder Antwort vom Empfänger.

2.4.2 Nachrichtenorientierte transiente Kommunikation

Berkeley-Sockets:
Socket=Kommunikationsport, an den eine Applikation Daten schreiben kann, die über das zugrunde liegende Netzwerk versendet werden sollen, und von dem eingehende Daten gelesen werden sollen.
Liegt auf Transportebene(hier TCP).
4 elementare Funktionen: Socket, Bind, Listen, Accept.
MPI (Message-Passing Interface)
Sockets nicht geeignet für Hochgeschwindigkeits-Netzwerke- SS nötig für Formen des Pufferns und der Synchronisierung->Definition eines Standards: MPI.
MPI ist auf parallele Applikationen ausgelegt und damit auf die transiente Kommunikation!
Voraussetzung für MPI: Kommunikation innerhalb einer bekannten Prozessgruppe- jeder Gruppe und jedem Prozeß wird eine ID zugewiesen, Quelle oder Ziel ist dann eindeutig als ein Paar (gruppenID,prozessID).
Nicht unterstützt wird nur die empfangsbasierte, transiente, synchrone Kommunikation.

Nachrichtenorientierte persistente Kommunikation

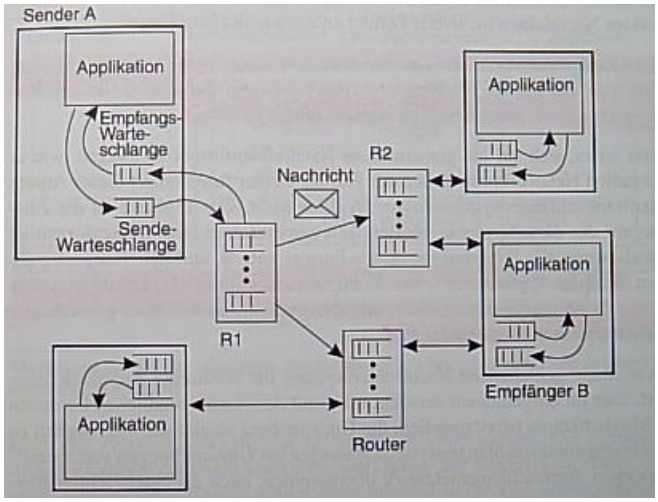
Warteschlangen oder MOM (Message oriented Middleware) sind im Unterschied zu Sockets und MPI darauf ausgelegt, Nachrichtenübertragungen zu unterstützen, die mehrere Minuten dauern.
Sollen persistente Kommunikation zwischen den Prozessen ermöglichen.
Unterstützte Anwendungen zB: Email, Workflow,Groupware,Batch-verarbeitung,Integration von verstreuten Datenbanken in eine Multi-Datenbank (Abfragen werden entsprechend weitergeleitet).

Nachrichtenwarteschlange-Modell

Nachrichtenübergabe über spezielle Kommunikationsserver an Ziel. Dieses kann inaktiv gewesen sein bei Nachrichtenabsendung.
Wichtiger Aspekt: Sender erhält iA nur die Garantie, dass eine Nachricht irgendwann in die Warteschlange des Empfängers gestellt wird.
Warteschlangen erlauben iA, dass ein Prozess eine Verarbeitungsroutine als Callback-Funktion installiert, die automatisch aufgerufen wird, wenn eine Nachricht in die Warteschlange gestellt wird- zB Starten eines Prozesses um Nachrichten aus der Warteschlange zu holen, wenn gerade kein Prozess ausgeführt wird.

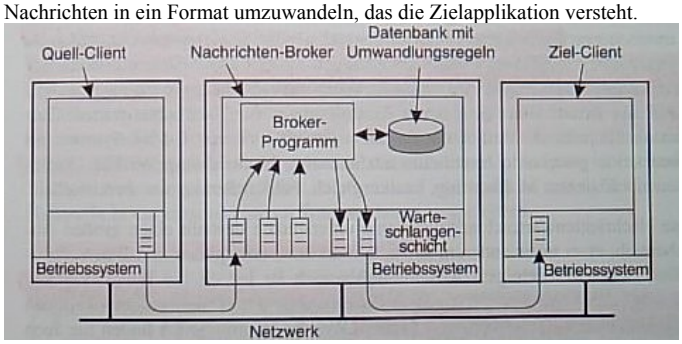
Allgemeine Architektur eines Nachrichtenwarteschlangensystems

Quellwarteschlange: Warteschlange auf derselben oder höchstens benachbarten Maschine.=lokal für den Sender, nur dort und nicht weiter kann Sender seine Nachricht einstellen. Enthält Adresse der Zielwarteschlange.
Warteschlangen werden von Warteschlangenmanagern verwaltet- dieser kommuniziert normalerweise direkt mit der Applikation, die eine Nachricht sendet oder empfängt, können aber auch als Router oder Relays arbeiten: Geben Nachrichten an andere Warteschlangenmanager weiter:



Nachrichten-Broker

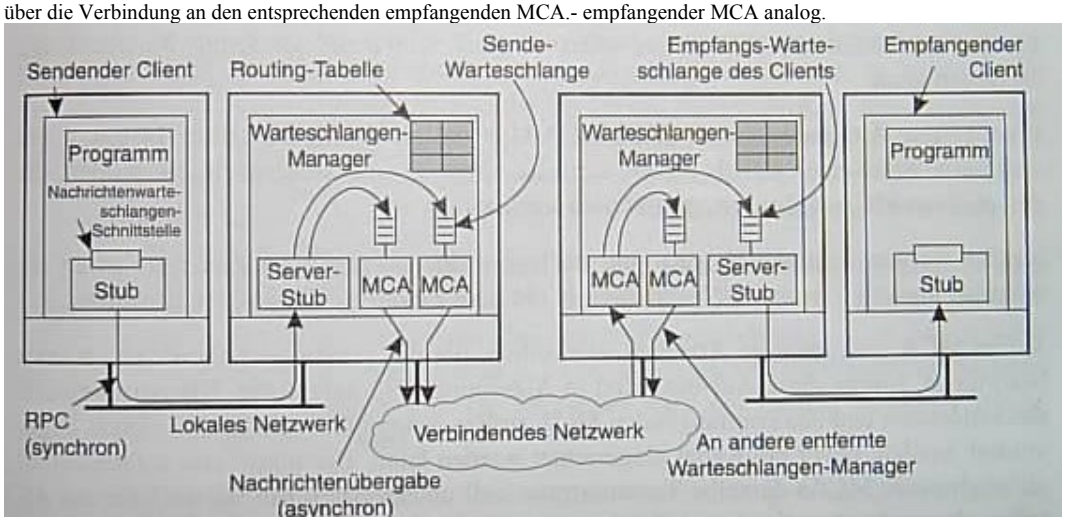
Nachrichten-Broker agieren als Gateway auf Applikationsebene in einem Nachrichtenwarteschlangensystem, hat die Aufgabe, eingehende Nachrichten in ein Format umzuwandeln, das die Zielapplikation versteht.



2.4.3 Beispiel: IBM MQSeries

Überblick

Warteschlangen-Manager sind paarweise durch Nachrichten-Channels miteinander verbunden. Jedes der beiden Enden eines Nachrichten-Channels wird durch einen MCA (Message Channel Agent) verwaltet.
Aufgabe sendender MCA: Überprüft Sende-Warteschlangen auf Nachrichten, verpackt sie in ein Paket für Transportebene und sendet sie über die Verbindung an den entsprechenden empfangenden MCA.- empfangender MCA analog.



Kommunikation über RPC, wenn Warteschlangen-Manager und Applikation auf verschiedenen Maschinen sind.

Channels-Nachrichtenkanäle

Übertragung über den Kanal nur möglich, wenn sowohl der sendende als auch der empfangende MCA aktiv sind-können auch über Trigger oder Steuernachrichten gestartet werden.
Abzustimmende Attribute zwischen MCAs:

Transporttyp	Legt das Transportprotokoll fest
FIFO-Auslieferung	Nachrichtenauslieferung in Sende-Reihenfolge
Nachrichtenlänge	Maximale Länge einer einzelnen Nachricht
Setup Wiederholungszähler	Max. Anzahl der Wiederholungen, den entfernten MCA zu starten
Auslieferungswiederholungen	Max. Versuche des MCA, eine empfangende Nachricht in die Warteschlange zu stellen

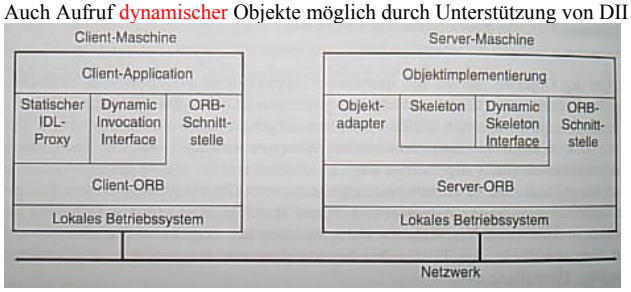
Nachrichtenübertragung

Zur Übertragung Übertragungsheader eingefügt: Zielwarteschlangen-Manager|Zielwarteschlange|Route (Zieladresse)
Routen werden in Warteschlangen-Managern in Routing-Tabellen gespeichert, hier wird bei jeder auszuliefernden Nachricht nachgeschlagen, an welchen benachbarten Warteschlangen-Manager die Nachricht weitergegeben werden soll.

9.1 Corba

Überblick

CORBA(Common Object Request Broker Architecture), entwicklet von OMG (Object Management Group)=objektorientierte Middleware, die verschiedene Plattformen und Sprachen unterstützt (ähnlich RMI, nur unterstützt diese nur Java)
Der ORB (Object Request Broker) ist für die Kommunikation zwischen Clients und Objekten verantwortlich und daneben auch für die Facilities (Vertikal=domänenspezifisch-E-Commerce,Banking, Horizontal=von Applikationsdomänen unabhängige High-Level-Dienste-Benutzeroberflächen…)
Objektmodell
Modell: Entfernte Objekte, Spezifikation der Objekte und Dienste in IDL.
Statisch: IDL-Compiler erzeugt Proxy und Skeleton, die mit der Programmlogik ergänzt werden, Proxy gibt Daten dann an ORB weiter, dieser an ORB des Servers etc.
Auch Aufruf dynamischer Objekte möglich durch Unterstützung von DII (Dynamic Invocation Interface)-Aufruf mit invoke-Op.



Schnittstellen- und Implementierungsrepository
In der Schnittstellen-Repository werden alle Schnittstellendefinitionen gespeichert, bei Compilierung weist IDL-Compiler der SS eine Repository-ID (abgeleitet aus Namen und Methoden) zu. Damit kann die SS-definition aus dem Repository geladen werden.
Die Implementierungs-Repository beinhaltet alles, was für Implementierung und Aktivierung von Objekten benötigt wird-nötig zB für Objekt-Adapter, die sich für eine Objektreferenz an das Implementierungs-Repository wenden, um genau festzustellen, was getan werden muss.
CORBA-Dienste
Einige Dienste:

Naming Service, ermöglicht Serverobjekten, mittels eines festgelegten Namens angesprochen zu werden. Der Namensdienst liefert dann die IOR zu einem registrierten Objektnamen. Der Naming Service ist eine Art "Telefonbuch" für Corba Objekte.

Der **Trading Service** ermöglicht es ebenfalls, Objekte zur Laufzeit zu finden. Allerdings werden Objekte hier über ihre Eigenschaften identifiziert und nicht durch einen Namen. Das Ergebnis einer solchen Suche können auch mehrere Objekte sein.

Der **Event Service** ermöglicht lose, gekoppelte oder ereignisbasierte n:n Kommunikation. Die Aufrufe erfolgen nicht mehr synchron. Beim Push-Modell senden die Server-Objekte die Ergebnisse zum Client, beim Pull-Modell fragen die Clients explizit nach Ereignissen.

Der **Life Cycle Service** stellt Operationen zum Kopieren (Migrieren), Verschieben und Löschen von Objekten.

9.1.2 Kommunikation

Modelle für den Objektaufruf

Anforderungstyp	Fehlersemantik	Beschreibung
Synchron	Höchstens-einmal	Aufrufer blockiert, bis eine Antwort zurückgegeben oder eine Ausnahme geworfen wurde
Ein-Wege	Bestmögliche Auslieferung	Aufrufer kehrt zurück, ohne auf Antwort vom Server zu warten
Verzögert synchron	Höchstens-einmal	Aufrufer kehrt unmittelbar zurück und kann später blockieren,

		bis Antwort ausgeliefert wurde

Höchstens-einmal: Wird korrekte Antwort gegeben, garantiert CORBA, dass die Methode genau einmal aufgerufen wurde.

Event- und Notification-Dienste

2 Modelle für Auftreten von Ereignissen:

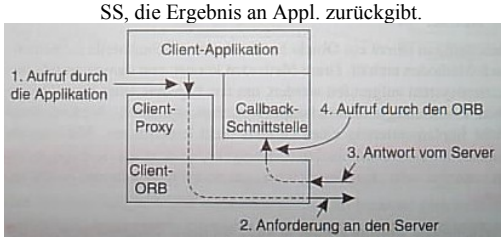
- Push-Modell:Erzeuger schiebt Ereignis durch Ereigniskanal, der es an Verbraucher verteilt.
- Pull-Modell:Verbraucher fragen Ereigniskanal ab.

Notification-Dienst bietet Filtermöglichkeiten.

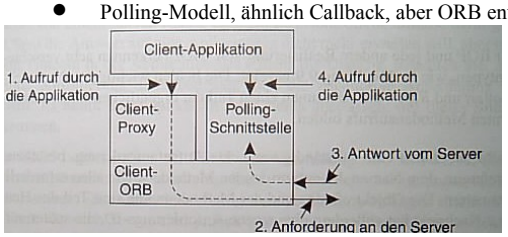
Nachrichten

Modelle zur persistenten Speicherung:

- Messaging-Dienst: Warteschlangen
- Callback-Modell: Client ist dafür verantwortlich, dass der synchrone in einen asynchronen Aufruf umgewandelt wird, Server sieht die Anforderung synchron. 2 neue Schnittstellen: 1.Methodenspezifikation der SS, ohne Rückgabe, 2. Callback-SS, die Ergebnis an Appl. zurückgibt.



- Polling-Modell, ähnlich Callback, aber ORB enthält 2. SS



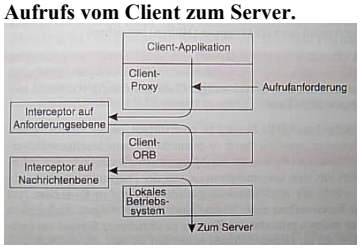
Interoperatiblität

Standardisierung zwischen den verschiedenen ORBs durch **GIOP (General Inter-ORB Protocol)**, setzt auf zuverlässigem,verbindungsorientierten Transportportokoll auf (zB TCP) TCP+GIOP=IIOP (Internet Inter-ORB-Protocol).

9.1.3 Prozesse

Clients

Mechanismus zum Anpassen der Proxies (zB wegen Caching-Strategien): **Interceptor: Mechanismus zjm Abfangen und Anpassen eines Aufrufs vom Client zum Server.**



Portable Objektadapter

Objektadapter (=Wrapper) passen Programme an, dass Clients es als Objekt erkennen.

POA(Portable Object Adapter)=Komponente, die verantwortlich dafür ist, dass der server-seitige Code den Clients als CORBA-Objekte präsentiert wird.

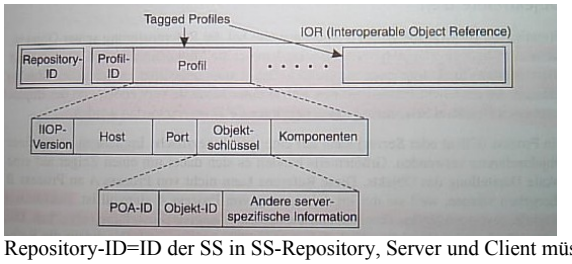
Servant =Teil des Objekts, der die Methoden implemeniert, die die Clients aufrufen können.

Agenten

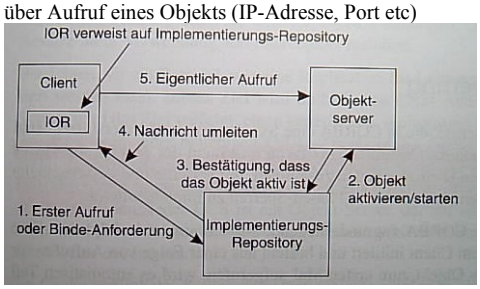
CORBA gibt nur SS vor, die ein **Agentensystem** (=Plattform, die das Erzeugen, Übertragen…von Agenten ermöglicht) implementieren sollte- unterschiedliche Agententypen können in einer einzigen verteilten Applikation verwendet werden. Jedes Agentensystem muß mehrere StandardOps implementieren.

9.1.4 Namensgebung

ORBs müssen (sprachspezifische) Objektreferenzen von Client-Prozessen in eine prozessunabhängige Darstellung verwandeln: **IOR (Interoperable Object Reference).** Aufbau:



Repository-ID=ID der SS in SS-Repository, Server und Client müssen auf dasselbe SS-R. Zugriff haben! Tagged Profiles enthält vollst. Info über Aufruf eines Objekts (IP-Adresse, Port etc)



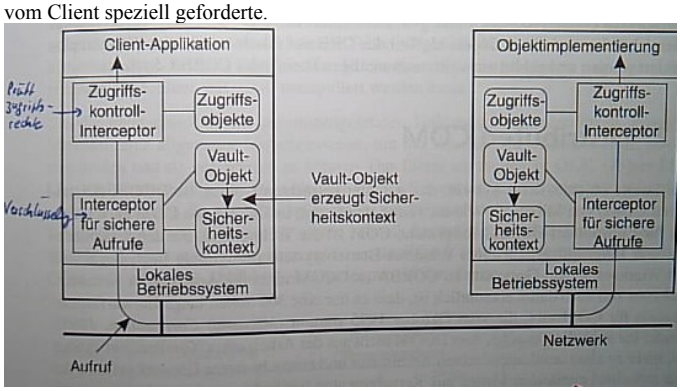
CORBA-Namensdienst
Name=(ID Art)-Paar, Art=Dateinamenserweiterung.
ORB enthält **Namenskontext** (Tabelle mit Zuordnung Name:Objektreferenz)

9.1.5 Synchroniserung

Wichtigste Dienste: Concurrency Control-Dienst (Nebenläufigkeitskontrolle) und Transaction-Dienst (Transaktionen), arbeiten zusammen um verteilte und verschachtelte Transaktionen von 2-Phasen-Sperren zu implementieren

9.1.8 Sicherheit

Client bindet sich zu Objekt->Client-ORB entscheidet welche Sicherheitsdiesnte die Clientseite benötigt=vom Admin festgelegte als auch vom Client speziell geforderte.



Um ORB allgemein zu halten, werden ersetzbare Sicherheitsdienste (=Implementierung der Dienste über Standard-SS) implementiert. Der Zugriffskontoll-Interceptor überprüft die Zugriffsrechte, der Interceptor für sichere Aufrufe kümmert sich um den Schutz der Nachricht (verschlüsselt Nachrichten für Integrität und Vertraulichkeit), indem er einen Sicherheitskontext für Client einrichtet, dieser wird mithilfe eines Sicherheitskontextobjekts dargestellt:
Client-Interceptor sendet Nachricht an Objektserver, so dass der Server einen Sicherheitskontext für nachfolgende Aufrufe erzeugen kann. Danach wird der Client zum Zielobjekt gebunden-eine Sicherheitszuordnung ist eingerichtet.

KE3

3.1 Threads

3.1.1 Threads-Einführung

Zur Ausführung eines Programms erzeugt das BS einen virtuellen Prozessor pro Prozess. Transparente Nebenläufigkeit mehrerer Prozesse hat hohen Preis: Sichern des CPU-Kontextes (Registerwerte etc.) , MMU verändern etc.
Threads sind LWPs (Light weight process): Thread-System verwaltet nur Mindestinfo zur Ermöglichung mehrerer Threads pro CPU:

Thread-Kontext besteht nur aus CPU-Kontext und min. Thread-Verwaltung- mehrere Threads eines Prozesses teilen sich die Betriebsmittel des Prozesses und greifen auf denselben Speicherraum zu.
Konsequenz: Erhöhter Programmieraufwand, da Threads anders als Prozesse nicht gegeneinander geschützt sind.
Thread-Verwendung in nicht verteilten Systemen

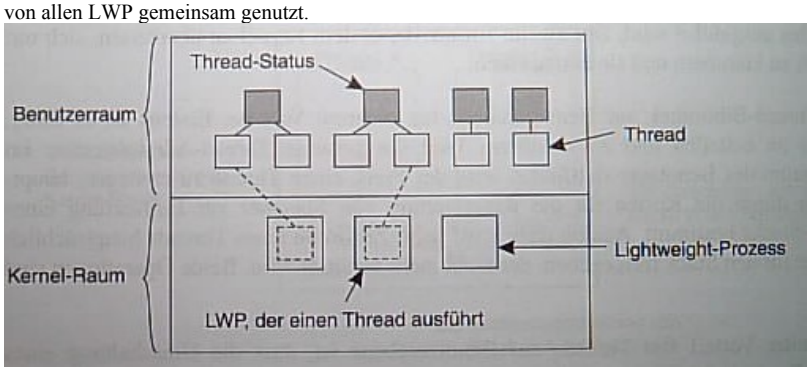
Bsp. Tabellenkalkulation-Berechnung nicht möglich, wenn Programm auf Eingabe wartet, daher üblich: 2 Threads: 1. Interaktion mit Benutzer, 2. Aktualisierung der Kalkulation
Bei Multiprozessoren 1 CPU pro Thread, Daten gemeinsam im RAM.
In Unix werden große Applikationen meist von separaten Prozessen ausgeführt, die mit IPC (Interprocess Communication) kommuniziren, dort sind dann immer Kontextumschaltungen nötig vom Benutzer- in Kernelraum und vice versa (Änderung Speicherabbildung der MMU und Leerung des TLB)=> besser: Thread-Verwendung- Kommuniktion nur unter Verwendung gemeinsamer Daten, Thread-Umschaltung vollständig im Benutzerraum oder Kernel ist über Threads informiert=>Leistungsverbesserung.

Thread-Implementierung

Threads werden iA in Form von Thread-Paketen bereitgestellt->enthalten Ops, mit denen Threads erstellt und zerstört werden+Sychron.variablen (Mutex,Bedingung).
2 Ansätze:

1. Thread-Bibliothek vollst. Im Benutzerraum: Billig,schnell (nur CPU-Register speichern bei Umschaltung), aber blockierender Systemaufruf blockiert gesamten Prozess (alle angehör. Threads).
2. Kernel kümmet sich um Threads: Teuer, jede Thread-Op muß von Kernel ausgeführt werden, und das ist immer ein Systemaufruf, also genauso teuer wie 1.

=>Lösung: **LWP** =Hybridform (aus Threads auf Benutzerebene und Kernelebene), wird im Kontext eines einzigen (heavy weight) Prozesses ausgeführt, mehrere LWP pro Prozess. Jeder LWP hat seinen eigenen Prozess (implizite Zuweisung), Thread-Tabelle wird von allen LWP gemeinsam genutzt.

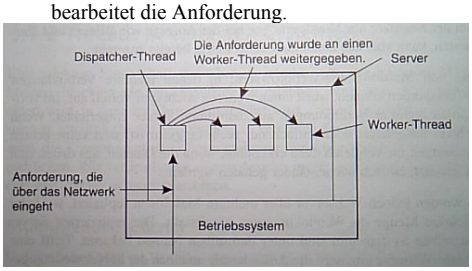


Findet ein LWP einen ausführbaren Thread, wechselt er in dessen Kontext, muß dieser Thread blockieren, ruft der LWP die Routine zur Einplanung des nächsten Threads auf. Die Kontextumschaltung findet vollständig im Benutzerraum statt.
Blockierender Systemaufruf eines Threads: Ausführung (im Kontext des aktuellen LWP) wechselt vom Benutzer- in Kernelraum. BS kann entscheiden, den Kontext zu anderem LWP umzuschalten.
Vorteile des LWP: Billig, Threads zu erstellen,zerstören,synchronisieren; Systemaufruf unterbricht nicht den gesamten Prozess; nicht erforderlich, dass Appl. etwas über LWPs weiß; LWP einfach einsetzbar in Multiprozessoren.
Nachteil: LWP erzeugen und zerstären genauso teuer wie Threads auf Kernel-Ebene.
Anderer Ansatz: **Scheduler-Aktivierungen:** Kernel nimmt Upcall für das Thread-Paket vor, wenn Thread nach Systemaufruf blockiert.
Vorteil: Verwaltung der LWPs eingespart
Nachteil: Nicht elegant, verletzt Struktur geschichteter Systeme=Aufrufe nur zur nächstniedrigeren Ebene.

3.1.2 Threads in verteilten Systemen

Multithreaded-Clients: zB. Browser: lädt Dokus und Bilder parallel
Multithreaded Server:

- Beliebt: **Dispatcher**-Thread-liest Anforderungen und wertet sie aus, **Worker**-Thread wird dann vom Server ausgewählt und bearbeitet die Anforderung.



- **Singlethreaded Prozess**: Keine Parallelität, blockierende Systemaufrufe
- **Automat** (nur ein Thread): blockiert nicht, sondern zeichnet Status der akt. Anforderung in Tabelle auf und setzt Arbeit dann fort.

3.2 Clients

3.2.1Benutzeroberflächen

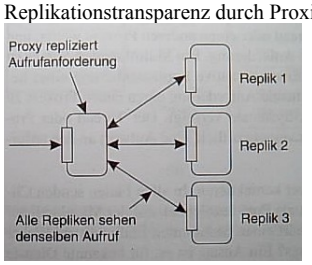
X Window

X-Kernel enthält alle Terminal-spez. Gerätetreiber, stark von Hardware abhängig.

X-Lib: Schnittstelle des X-Kernel, als Bibliothek den Applikationen zur Verfügung gestellt.
X-Kernell und X-Appl. müssen nicht auf derselben Maschine sein.
X-Protokoll: Instanzen vn X-Lib können mit X-Kernel Daten austauschen.
Zusammengesetzte Dokumente
Moderne GBOS leisten mehr als X: Gemeinsame Fensternutzung+Datenaustausch, Drag&Drop,In-Place-Bearbeitung.
Grundlegende Idee: Sammlung von Daten unterschiedlichen Typs integriert in GBO.

3.2.2Clientseitige Software für Verteilungstransparenz

Besteht neben GBO und anderen für die Appl. benötigter SW aus Komponenten, mit denen Verteilungstransparenz erzielt werden soll..
Replikationstransparenz durch Proxies.



Ortstransparenz: Server informiert bei Änderung Clients, Middleware verbirgt das.
Fehlertransparenz: Realisiert über Middleware
Nebenläufigkeit: über spezielle Zwischenserver (besonders Transaktionsmonitore)
Persistenztransparenz: häufig vollständig auf Server.

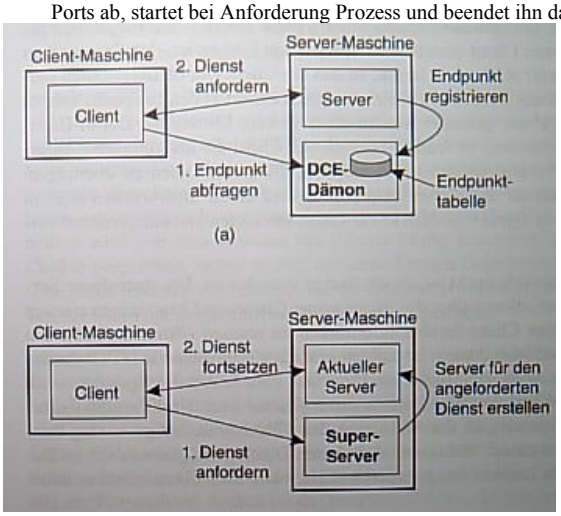
3.3 Server (Prozess)

Aufbauarten:

- iterativ: verarbeitet Anforderungen selbst
- nebenläufig: gibt sie an separaten Thread weiter

Kontaktstelle:Ports-jeder Server hört bestimmten Port ab.
Clientkenntnisse des Ports:

1. Festgelegt (zB http:80)
2. dynamisch: Dämon wie bei DCE auf jedem Server ausführen. Client kontaktiert Dämon über bekannten Port, fordert Port an und kontaktiert den Server.
3. besser: Statt DCE-Dämon **Superserver**: Prozesse warten häufig passiv->Verschwendung! Superserver hört alle bekannten Ports ab, startet bei Anforderung Prozess und beendet ihn danach wieder



Unterbrechen des Servers:

- Beenden der Applikation clientseitig.
- Client und Server so entwickeln, dass sie Out-of-Band Daten senden können: Diese Daten haben höhere Priorität, Übertragung über separaten Port oder denselben (in TCP möglich) –dringende Daten werden zuerst ausgewertet.

Status:

1. Statuslos (kann Status ändern, ohne Clients zu informieren) zB Webserver, vergisst nach Anforderung Client vollständig
2. Statusbehaftet: verwaltet Infos über seine Clients zB Tabelle über (Client-Datei-)Paare; Vorteil: Leistungsverbesserung; Nachteil: Absturz-Tabelle muß wiederhergestellt werden.
3. Cookies: Server zeichnet Clientverhalten auf zB bevorzugte Website.

Objekt-Server

Objekt-Server= Ort, an dem sich Objekte befinden.
Wichtigster Unterschied zu anderen Servern: OS stellt nicht wirklich spezifischen Dienst bereit. Diese werden von den Objekten implementiert-Objekt besteht aus 2 Teilen: Code und Daten (Status).

Alternativen zum Aufruf von Objekten

Objekt-Server muß wissen, welcher Code ausgeführt werden soll, mit welchen Daten arbeiten etc.
Ansätze:

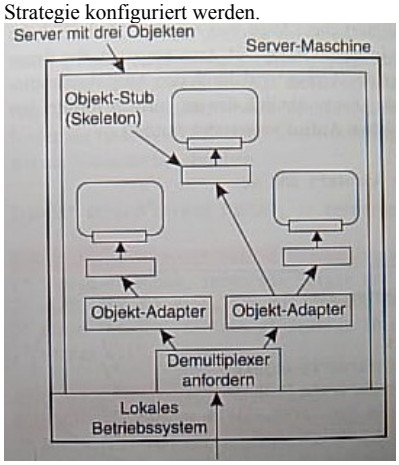
- Voraussetzen, dass alle Ojekte gleich aussehen, nur eine Möglichkeit, Objekte aufzurufen (DCE)->unflexibel
- Unterschiedliche Strategien: zB transisiente Objekte: Erzeugen bei 1.Aufruf und zerstören,wenn kein Client mehr daran gebunden ist.
- Jedem Objekt in einem eigenen Speichersegment platzieren (Objekte nutzen somit weder Code noch Daten gemeinsam) erforderlich, wenn Objektimplementierung Code und Daten nicht voneinander trennt oder aus Sicherheitsgründen.

Alternativ: wenigstens Code gemeinsam nutzen zB Klasse Datenbank, muß für Objekte dann nur noch Status laden

- Threading: a) Ein Steuerthread b) Thread für jedes Objekt-somit geschützt gegen nebenläufigen Zugriff

Objekt-Adapter

Aktivierungsstrategie=Entscheidung darüber, wie ein Objekt aufgerufne wird (Objekt muss vor aufruf in Adressraum des Servers gebracht werden=**aktiviert**).
Objekt-Adapter (Wrapper)=SW, die eine bestimmte Aktivierungsstrategie implemeniert (zB ein Thread pro Objekt oder ein Thread für alle Objekte). Stellen generische Komponenten dar, die den Entwicklern verteilter Objekte helfen sollen, müssen nur für eine Strategie konfiguriert werden.

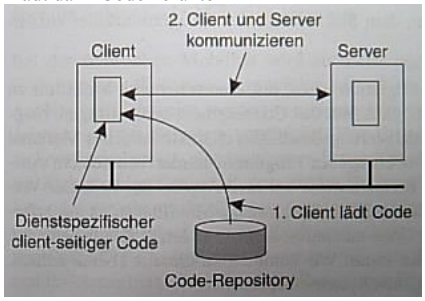


Es können sich mehrere Objektadapter auf einem Server befinden=>mehrere Aktivierungsstrategien.
Objektadapter extrahieren Objektreferenz aus Aufrufanforderung und leiten sie an den Skeleton weiter.
Braucht Prozeduren zum De-/Registrieren der Objekte, Registrierung gibt ID für Objekt zurück.
Skeleton muß invoke(…) implementieren.
Unabhängigkeit von Objekten und Adaptern:Man kann Objekt-Adapter auf der Middlewareschicht platzieren und unabhängig davo Objekte entwickeln, für die man dann nur noch angeben muß, welcher Adapter deren Aufruf steuern soll.

3.4 Code-Migration

Ansätze

Performance:1. Sinnvoll, Dateien in Nähe der Speicherpostitionen zu verarbeiten, 2. Parallelität
Flexibilität:1. Zerlegung von Code („Multitier"), 2. Dynamische Konfiguration: Implementierung erst, wenn Client sich zu Server bindet->lädt dann Code herunter



Modelle für dieCode-Migration

Prozess besteht aus 3 Segmenten:

- Code-Segment (Menge der Anweisungen des Programms)
- Ressourcensegment (Verweis auf externe Ressourcen (Drucker, Dateien…)
- Ausführungssegment (Ausführungsstatus des Prozesses zB priv Daten, Stack, Programmzähler)

Zu unterscheiden:

Schwache Mobilität: Nur Code-Segment wird übertragen (Java-applets)	Vom Sender initiierte Migration (zB Programme hochladen)	Im Zielprozess ausführen (werden im Adressraum des Servers ausgeführt, Nachteil Sicherheit)
		In separatem Prozess ausführen
	VomEmpfänger initiierte Migration (zB Java-Applets, geht von Zielmaschine aus)	Im Zielprozess ausführen
		In separatem Prozess ausführen
Starke Mobilität: Auch Ausführungssegment wird übertragen (D'Agents)	Vom Sender initiierte Migration	Prozess migrieren
		Prozess klonen (wird parallel zum Originalprozess ausgeführt)
	VomEmpfänger initiierte Migration	Prozess migrieren
		Prozess klonen

Migration und lokale Ressourcen

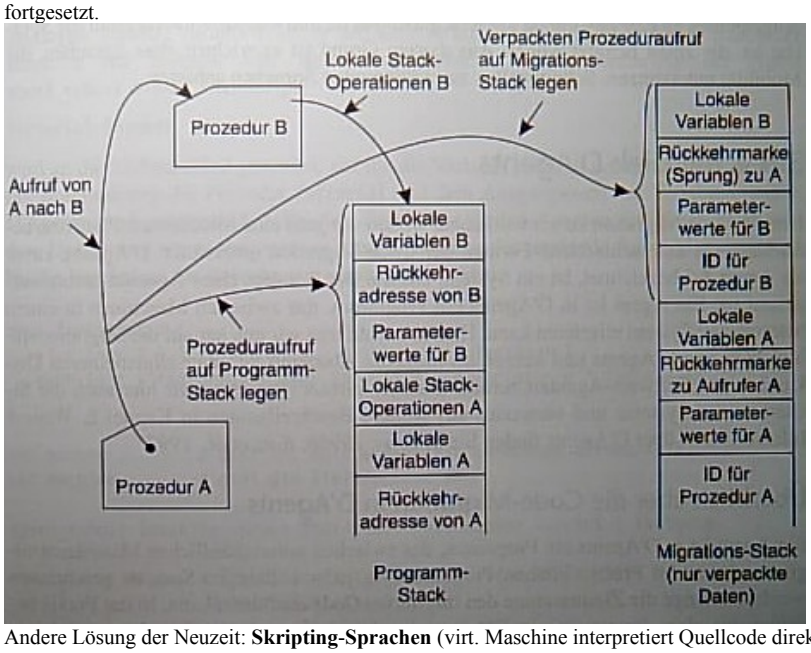
Problem bei Migration des Ressourcensegments: Nicht so einfach, ohne Änderung zu übertragen.
3 Typen von Bindungen von Prozessen zu Ressourcen:

1. **Über ID** (stark): URL, Verweise mit Hilfe einer Internetadresse,lokale Kommunikationsports: genaus die Ressource, nichts anderes
2. **Bindung nach Wert** (mittel): Andere Ressource kann selben Wert bereitstellen zB Java-Bibliotheken, Ort verschieden **.**
3. **Bindung nach Typ** (schwach): Verweise auf lokale Geräte zB Drucker

Bindung von Ressource zu Maschine**:**
Nicht zugeordnete Ressourcen: können einfach verschoben werden zB Daten, die nur einem Programm zugeordnet sind.
Zugeordnete Ressourcen: Verschieben/Copy möglich, aber hohe Kosten zB Datenbanken
Fixe Ressourcen: können nicht verschoben weren zB lokale Geräte, lokale Kommunikationsendpunkte

Migration in heterogenen Ressourcen

Problem bei heterogenen Systemen: Schwache Mobilität: ausreichend, Quellcode zu kompilieren, aber untersschiedliche Codesegmente zu erzeugen- eines für jede Zielplattform.
Bei starker Mobilität ist das Problem die Übertragung des Ausführungssegments (wegen plattformabhängigen Informationen)-nur möglich bei selber Architektur und BS!
Lösung: Beschränkung auf bestimmte Ausführungspunkte (Aufruf einer Subroutine): Programmstack wird auf **Migrations-Stack** kopiert, dieser wird mitübertragen und das Ziel entpackt diesen und legt damit neuen Laufzeitstack an.- die Ausführung wird dann mit der Subroutine fortgesetzt.



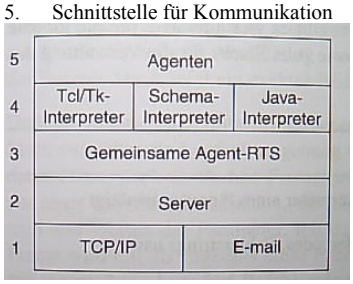
Andere Lösung der Neuzeit: **Skripting-Sprachen** (virt. Maschine interpretiert Quellcode direkt) und **höchst portable** Sprachen (Java,NET)

D'Agents

Überblick über Code-Migration in D'Agents
Agent in D'Agents=Programm, das zwischen unterschiedlichen Maschinen migrieren kann-können in jeder Sprache geschrieben werden, solange Zielmaschine den migrierten Code ausführen kann
=> Programme sind in interpretierten Sprachen geschrieben (TCL[Tool Command Language]): Java,Scheme.
Mobilität auf 3 Arten:
Vom Sender initiierte schwache Mobilität: agent-submit: Name der Zielmaschine und Script werden übergeben, Quellprozess blockiert.

Starke Mobilität durch Prozess-Migration: agent-jump: Code-Ressourcen-,Ausführungssegment werden in Nachricht verpackt und an Ziel gesendet, Quellprozess blockiert.
Starke Mobilität durch Prozess-Cloning: agent-fork: wie jump, nur setzt Quellprozess Arbeit fort
Implementierungsaspekte
D'Agents hat 5 Schichten:

1. Agenten: werden von separatem Prozess ausgeführt. Migriert Agent auf Maschine A, erzeugt Server A Prozeß, der den entsprchenden Interpreter für den Agenten ausführt. Neuem Prozess wird Status des migrierten Agenten übergeben.
2. Interpreter: Je ein Interpreter (bestehend aus Sprachinterpretation,Sicherheitsmodul, SS zur Kernsschicht, Modul für Statusaufzeichnung des Agenten) für jede unterstützte Sprache
3. Gemeinsame Agenten-RTS: Sprachenunabhängiger Kern, um Agenten zu -starten,-beenden,-verschiedene MigrationsOps, -Kommunikation zwischen Agenten
4. Server: Verantwortlich für VErwaltungder Agenten und deren Kommunikation sowie Authentifizierung, weist jedem Agenten eine eindeutige ID zu.
5. Schnittstelle für Kommunikation



Schwierigste Aufgabe bei Implementierung von D'Agents:
Aufzeichnung und Weitergabe des Status- in TCL 4 Tabellen für Definition von Variablen und Scripts, und 2 Stacks für Verwaltung des Ausführungsstatus.
Aufruf eines grundlegenden TCL-Befehls (kein Aufruf einer benutzerdefinierten Prozedur):
Interpreter wertet Befehl aus, erzeugt Datensatz, der auf den Befehlsstack gelegt wird (enthält zB Parameterwerte),Zeiger auf Prozedur…). Kann dann Komponente übergeben werden, die für eigentliche Ausführung des Befehls verantwortlich ist.
Aufruf einer benutzerdefinierten Prozedur:
Laufzeitumgebung von D'Agents verwaltet auch Stack mit Aktivierungsdatensätzen=Aufruf-Frames.
Aufruf-Frame

- ⇨ besteht aus Tabelle mit Variablen, die lokal für die Prozedur sind und Namen und Werten der Parameter
- ⇨ verwaltet alle Referenzen auf den zugehörigen Befehl

agent-jumpp-Aufruf: Status wird in Byte-Folge verpackt und an Ziel übergeben, dort wird neuer Prozess erzeugt, der TCL-Interpreter ausgeführt und Ausführung dort fortgesetzt, wo zuvor unterbrochen.

3.5 Software-Agenten

SW-Agenten in verteilten Systemen

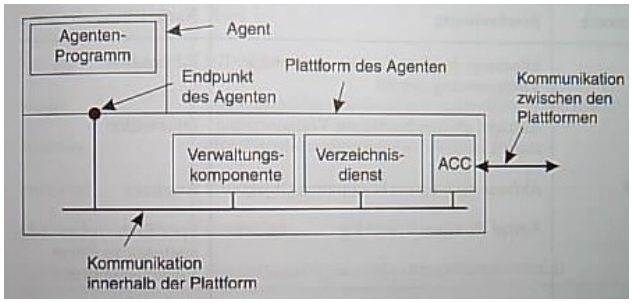
SW-Agent= autonomer Prozess, der in der Lage ist, auf Änderungen in seiner Umgebung zu reagieren und diese zu initiieren, möglicherweise in Zusammenarbeit mit Benutzern oder anderen Agenten.

- ⇨ **Kollaborativer Agent:** bildet Teil eines Multi-Agenten-Systems, bei dem Agenten versuchen, gemeinsam ein Ziel zu erreichen (zB Meeting)
- ⇨ **Mobiler Agent:** kann zwischen verschiedenen Maschinen verschoben werden zB Infosuche im Internet
- ⇨ **Schnittstellen-Agent**: Hilft Benutzer, ein oder mehrere Applikationen zu benutzen, lernfähig zB Käufer, Verkäufer
- ⇨ **Informations-Agent**: verwaltet Infos aus verschiedenen Quellen zB Spam-Filter

Agenten-Technologie

Allgemeines Modell, entwickelt von FIPA (Foundation for Intelligent Physical Agents):
Agenten werden bei einer Agenten-Plattform registriert.
Komponenten der Plattform:

- ⇨ **Verwaltung**: Agenten erzeugen/löschen,Suchen eines aktuellen Endpunkts für best. Agenten (Namensdienst)
- ⇨ **Verzeichnisdienst**: Agenten können ermitteln, was andere Agenten zu bieten haben, basiert auf Attributen (Gelbe Seiten)
- ⇨ **ACC** (Agent Communication Channel): kümmert sich um gesamte Kommunikation zwischen Agenten-Plattformen, kann als Server implementiert werden, der bestimmte Ports abhört. Wegen Interoperabilität zwischen Plattformen: Kommunikation zwischen ACCs nach IIOP (Internet InterORB)-Protokoll. Beispiel für ACC: Server in D'Agents



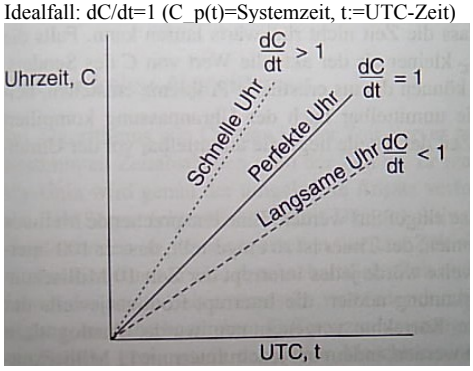
Agenten-Kommunikationssprachen
Kommunikation unter Agenten mit **ACL** (Agent Communication Language):
Zweck der Nachricht (zB INFORM)+Nachrichteninhalt

5.1 Uhr-Synchronisierung

Computer-Time=Quarzkristall, 2 Register zugeordnet: Zähler- und Halteregister.
Jede Schwingung dekrementiert Zähler, Zähler==0?=> Interrupt=Uhr-Tick, Zähler aus Haltereg. neu laden.
Uhr-Assymetrie: Unterschiede in Zeitwerten.
Grundlage für jede moderne zivile Zeitverwaltung=**UTC (Universal Coordinated Time).**
Diskrepanz zwischen **TAI** (International Atomic Time) und Sonnenzeit >=800msec. Zur Synchronisierung Einführung von Schaltsekunden
WWV=Kurwellen-Radiosender, kurzer Impuls am Anfang jeder UTC-Sekunde.

Algorithmen für Uhr-Synchronisierung

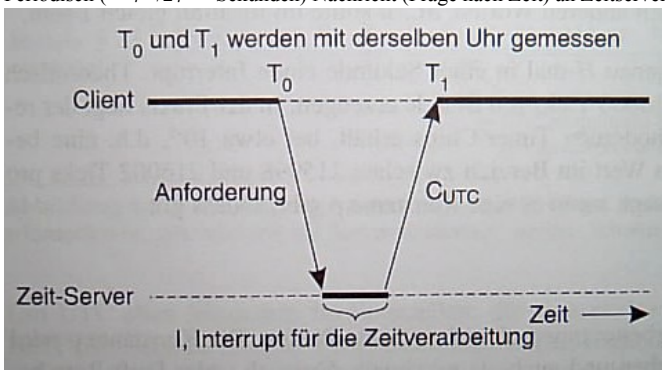
Grundlegendes Systemmodell für Maschinen ohne WWV-Empfänger:
Idealfall: dC/dt=1 (C_p(t)=Systemzeit, t:=UTC-Zeit)



Herstellerkonstante ρ mit 1- $1-\rho \leq \frac{dC}{dT} \leq 1+\rho$ => Uhren können sich bis zu 2Differenz aufweisen.

Garantie der Hersteller, dass 2 Uhren sich nie um mehr als ρ unterscheiden0>Synchronisation alle $\frac{\rho}{2\rho}$ Sekunden.

Algorithmus von Cristian
Gut für Systeme, wo eine Maschine WWV hat und die anderen damit synchronisiert=**Zeit-Server**
Periodisch (>= ρ /2 ρ Sekunden) Nachricht (Frage nach Zeit) an Zeitserver->schnellstmögliche Antwort, die die aktuelle Zeit enthält.



2 Probleme:

1. Falls die Zeit in der Antwort kleiner ist als die akt. Systemzeit: Eine Objektdatei, die nach Zeitanpassung kompiliert wurde, hat dann Zeitstempel vor der Zeit der Quelle, die unmittelbar vor der Uhranpassung modifiziert wurde.

=> Nur schrittweise Änderung der Zeit nötig:Timer erzeugt zB 100 Interrupts/sec=10ms, die jedes Interrupt der Zeit dazuaddiert, bei Verlangsamung werden nur 9msec dazuaddiert, bis Änderung vollzogen wurde, bei Beschleunigung analog

2. Zeitverzögerung bis zur Antwort (von Netzlast abhängig)

⇨ Lösung:

- Sender misst Zeitintervall bis zur Antwort, $\Delta T = \frac{T1 - T0}{2}$ (Hin und Zurück)
- Besser: vom Betrag des gemessenen Zeitintervalls noch die Zeit I für die Verarbeitung des Interrupts abziehen, also $\Delta T = \frac{T1 - T0 - I}{2}$

Christian empfielt, Messreihen vorzunehmen, alle Messungen, in denen die Übertragungszeit einen bestimmten Schwellenwert überschreiten, werden verworfen, vom Rest der Messungen kann entweder ein Mittelwert genommen werden oder aber die schnellste Messung.

Berkeley-Algorithmus

Geeignet für Systeme ohne WWV

Regelmäßige Zeitabfrage an Maschinen durch Zeit-Server (Zeit-Dämon), berechnet durchschnittliche Zeit und teilt die individuelle Differenz den anderen Maschinen mit, indem er ihre gesendete Zeitabweichung miteinberechnet:

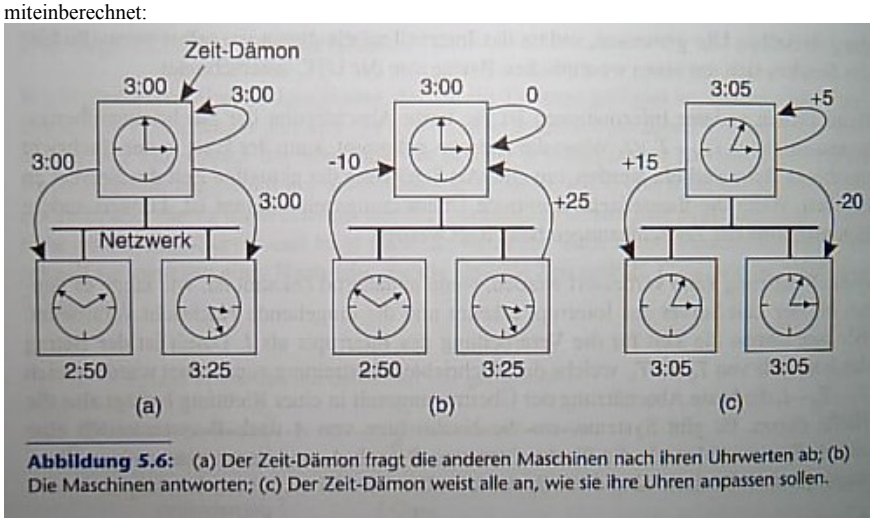


Abbildung 5.6: (a) Der Zeit-Dämon fragt die anderen Maschinen nach ihren Uhrwerten ab; (b) Die Maschinen antworten; (c) Der Zeit-Dämon weist alle an, wie sie ihre Uhren anpassen sollen.

Mittelwert-Algorithmen

Dies ist ein dezantralisierter Algo., unterteilt die Zeit Resynchronisierungsintervalle fester Länge:
Das i-te Intervall läuft von T_0 iR bis T_0 (i+1)R, T_0 = fester Zeitpunkt in der Vergangenheit, R=Systemparameter.
Am Anfang jeder Nachricht erfolgt Broadcast an alle Maschinen.
Dann Start von lokalem Timer, innerhalb dessen alle Broadcasts sammeln.
Danach Algo zur neuen lokalen Zeitberechnung-verschiedene Algos:

- Einfach: nur Mittelwert aller Werte ermitteln
- Variante: die m höchsten und m niedrigsten Werte verwerfen, Rest Mittelwert
- Jede Nachricht korrigieren mit Schätzwert (zB anhand bekannter Topologie des Netzwerks, oder Echomessungen)

Mehrere externe Zeitquellen

Übereinkunft nötig für jede CPU mit UTC-Quelle: Jeder sendet in bestimmten Zeitabständen Broadcast (zB am Anfang jeder UTC-Minute). Probleme durch Verzögerung, Kollosionen (gerade wenn mehrere Maschinen ihre Zeit gleichzeitig senden), Paketverarbeitung während des Broadcasts.

5.2 Logische Uhren

Logische Uhren: Nur interne Konsistenz der Uhren wichtig, unabhängig von realer Zeit-Reihenfolge wichtiger als absolute Zeit

Lamport-Zeitstempel

Lamport-Relation, auch **passiert-vor-Relation** genannt (transitiv):

1. Ereignis a tritt vor Ereignis b auf 0> a->b
2. Ereignis a =Prozess sendet Nachricht, Ereignis b=Prozess empfängt Nachricht => a->b (Nachricht kann nicht vor Sendung empfangen werden)

Für 2 Ereignisse, die keine Nachrichten austauschen gilt weder x->y noch y->x.

Algo von Lamport: Zeigt die empfangene Nachricht eine niedrigere Uhrzeit an als der Absendezeitpunkt, wird die Systemuhr einen Tick höher als die Absendezeit gestellt.

Ergänzung: Uhr muss zwischen 2 Ereignissen mind. 1x ticken.

Evtl erforderlich: Es gibt nie 2 Ereignisse zur selben Zeit0> dezimalpunkt nach Zeit: 40.1, 40.2, …

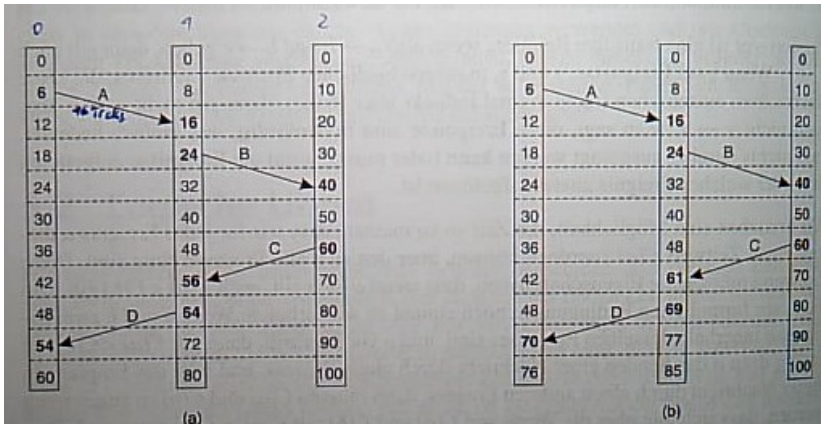


Abb b korrigiert die Uhren!

Bsp. Vollständig sortiertes Multicasting

Beispiel: 2 Kopien einer Kontodatenbank in 2 verschiedenen Städten, hebt ein Kunde in der einen Stadt etwas ab und gleichzeitig werden in der anderen Stadt Zinsen gutgeschrieben, kommt es wegen Kommunikationsverzögerungen zu Inkonsistenzen.
Hier benötigt man einen vollständig sortierten Multicast=eine Multicast-Op, wobei alle Nachrichten in derselben Reihenfolge an alle Empfänger ausgeliefert werden.
Empfängt ein Prozess einen Multicast, wird sie in eine nach Zeitstempeln sortierte Warteschlange gestellt. Empfänger sendet Multicast-Bestätigung, die Zeitstempel der empfangenen Nachricht sind nach Lamport kleiner als die der Bestätigung. Die Nachricht in der Warteschlange wird erst an die Appl. ausgeliefert, wenn sie von jedem Prozess bestätigt wurde. Da zwei Nachrichten nie denselben Lamport-Zeitstempel haben können und Zeitstempel eine konsistente globale Reihenfolge der Ereignisse reflektieren, haben alle Warteschlangen irgendwann dieselbe Kopie. Daher werden alle Nachrichten überall in derselben Reihenfolge ausgeliefert.

Vektor-Zeitstempel:

Problem mit Lamport-Zeitstempel: **Kausalität**- zB geht die Veröffentlichung eines Artikels immer einer Antwort darauf voraus!
=>Vektor-Zeitstempel VT(a) für Ereignis a: Wenn VT(a)<VT(b) geht Ereignis a sicher Ereignis b voraus.
Jeder Prozeß P verwaltet einen Vektor Vi mit folgenden Eigenschaften:

1. V.i[i] ist die Anzahl der Ereignisse in P.i
2. Wenn V.i[j]=k, erkennt P.i, dass auf p.j k Ereignisse aufgetreten sind.

Zu 1. Wird sichergestellt, indem man V.i[i] bei jedem neuen Ereignis in P.i inkrementiert
Zu 2. Vektor den Nachrichten mitgeben.
P.i sendet Nachricht m-> Zeitstempel vt von m teilt P.j mit, wie viele Ereignisse in anderen Prozessen m vorausgegangen sind und von wie vielen m evtl kausal abhängig ist->P.j setzt seinen Vektor V.j[k] auf max{V.j[k],vt[k]} (evtl müssen also noch Nachrichten empfangen werden vor m)-> dann V.j[k]++(=Ereignis des Empfangs einer nächsten Nachricht).
Abwandlung, um kausale Nachrichtenauslieferung zu garantieren, wegen Verzögerung bei der Übertragung
(P.i sendet Nachricht a als Multicast mit Zeitstempel vt(a)=V.i,
P.j sendet Reaktion auf P.i=Nachricht r, Zeitstempel vt(r)=V.j ,
r kann vor a bei P.k eintreffen):

1. vt.(r)[j]= V.k[j]+1 (=r=nächste Nachricht, die P.k von P.j erwartet)
2. vt.(r)[j]<=V.k[i] für alle i<>j (=P.k hat keine Nachricht gesehen ,die nicht auch P.j gesehen hat, als es Nachricht r gesendet hat)

KE4

8.1 Sicherheit und Verschlüsselung

Sicherheitsgefährudngen, Strategien, Mechanismen

Eigenschaften: Verfügbarkeit, Sicherheit, Wartbarkeit, Vertraulichkeit, Integrität
Sicherheitsgefährungen:

1. Lauschangriff
2. Unterbrechung (Dienste, Daten o.ä. nicht mehr nutzbar, zerstört)
3. Modifizierungen (Änderungen von Daten/Diensten)
4. Erzeugung (Daten erzeugen zB zusätzliche Passwort-Datei)

Sicherheitsstrategien

1. Verschlüsselung
2. Authentifizierung
3. Autorisierung (nach Authentitfizierung, zB Berechtigung für best. Datenbänke)
4. Auditing (Protokollierung der Zugriffe)

Bsp. Die Sicherheitsarchitektur von Globus

8 Aussagen zu den Sicherheitsstrategien von Globus

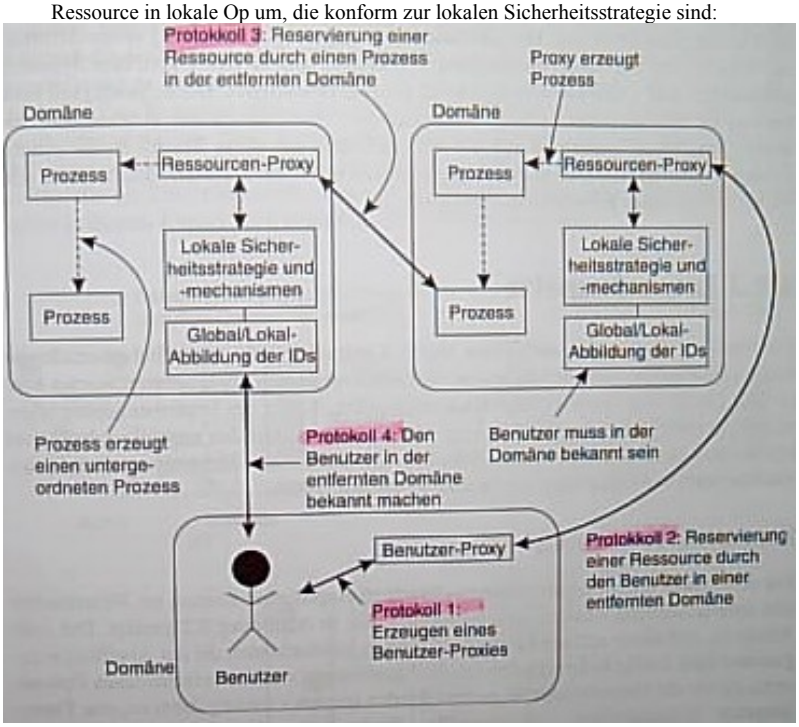
1. Umgebung besteht aus mehreren administrativen Domänen (Globus ändert keine lokalen Sicherheitsentscheidungen)

2. Lokale Ops unterliegen nur der Sicherheitsstrategie der lokalen Domäne (Globus führt dort keine zusätzlichen Maßnahmen ein)
3. Für globale Ops ist es erforderlich, dass der Initiator der Operation in jeder Domäne bekannt ist, die von der Op betroffen ist.
4. Für Ops zwischen Einheiten in unterschiedlichen Domänenist eine wechselseitige Authentifizierung erforderlich.
5. Eine globale Authentifizierung erzeugt eine lokale Authentifizierung
6. Der Zugriff auf Ressourcen wird nur von der lokalen Sicherheit gesteuert (nach Authentifizierung)
7. Benutzer können Rechte an Prozessen delegieren (zB.Agenten)
8. Eine Gruppe von Prozessen innerhalb derselben Donäne kann Berechtigungen gemeinsam nutzen

=> Globus braucht hauptsächlich Mechanismen für domänenübergreifende Authentifizierung u. Bekanntmachung eines Benutzers in entfernten Domänen

=> 2 Arten von Vertretern

1. Benutzerproxy: Prozess, der berechtigung erhält, begrenzte Zeit für einen Benutzer zu agieren
2. Ressourcen-Proxy: Prozess, der innerhalb einer bestimmten Domäne ausgeführt wird; wandelt globale Ops für eine Ressource in lokale Op um, die konform zur lokalen Sicherheitsstrategie sind:



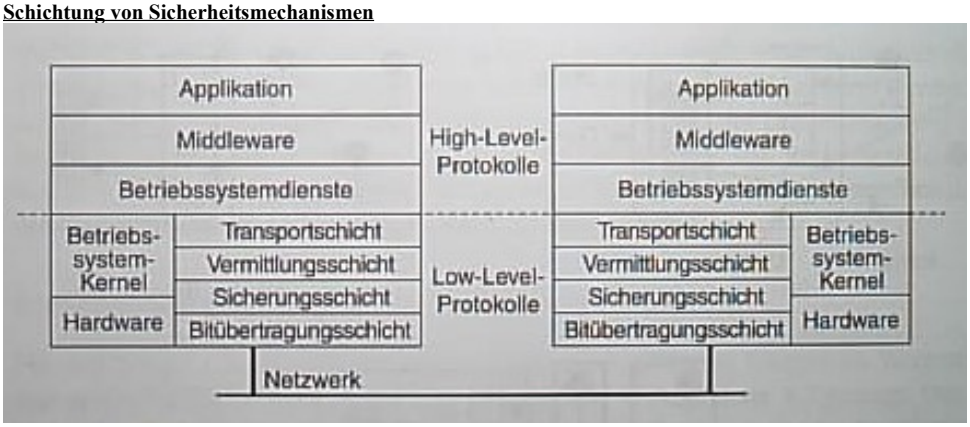
Entwurfsaspekte

Konzentrierte Kontrolle

Ansätze:

1. Daten direkt schützen gg falsche Ops
2. Daten gegen Zugrif schützen (Methodenfreigabe zB SS)
3. Daten schützen durch Überprüfung des Aufrufers

Schichtung von Sicherheitsmechanismen



Unterschied Vertrauen-Sicherheit: Sicher entweder ja oder nein, aber aus Clientbetrachtung ist das eine Vertrauensfrage!

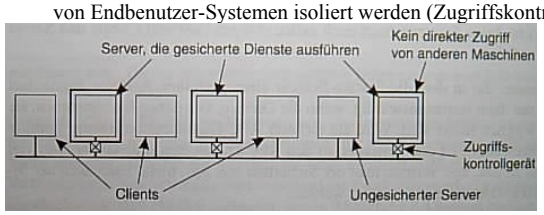
Verteilung von Sicherheitsmechanismen

TCB (Trusted Computing Base)= Menge aller Sicherheitsmechanismen in einem verteilten Computersystem, die benötigt wird, um eine Sicherheitsstrategie zu erzwingen. Je kleiner TCB, desto besser!
Middleware-basierteverteilte systeme bedingen Vertrauen zu den existierenden lokalen BS, von denen sie abhängig sind.
Ansonsten: Teilfunktionalität der BS in das verteilte System einfügen.
Anderer Ansatz: Sicherheitsdienste abtrennen-auf unterschiedlichen Maschinen verteilen (zB Dateiserver von Clients isolieren dh auf Maschine mit vertrauenswürdigem BS platzieren, Clients auf nicht vertrauenswürdigen BS)

➔ Abtrennung reduziert TCB auf kleine Zahl von Maschinen und SW.

Schützt man diese Maschinen , weitere Steigerung möglich mit: RISSC (REduced Interfaces for Secure System Components): Verhindert, dass der Client direkt auf kritische Dienste zugreift.

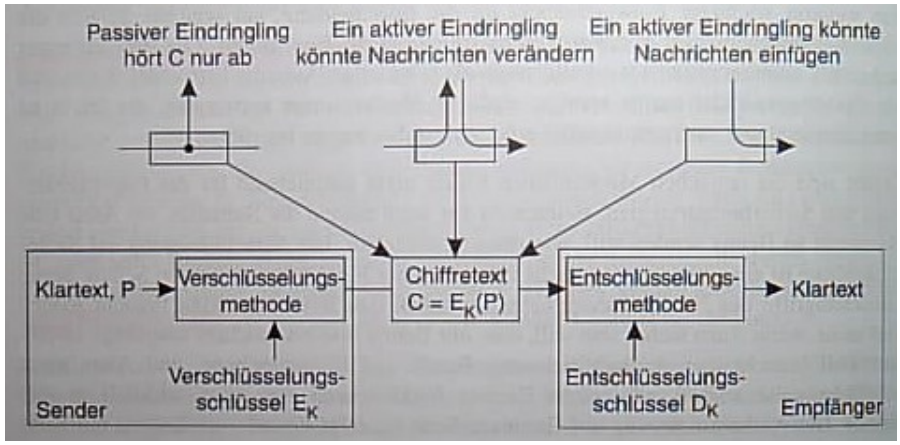
➔ Sicherheitskritische Server auf separater Maschine, die unter Verwendung von sicheren Low-Level-Netzwerkschnittstellen von Endbenutzer-Systemen isoliert werden (Zugriffskontrollgerät)



Einfachheit

Auf welchen Schichten sollten Sicherheitssysteme platziert werden?
Verschlüsselung auf Sicherheitsebene einfach, doch damit nur das Ziel die Nachrichten empfängt, ist Authentifizierung auf Benutzerebene nötig, dazu Konzept mit krypografischen Schlüsseln (wegen Vertrauen)+Mechanismus wie zB Zertifikate.
Aber evtl Applikation schon komplex-> wichtig, dass die Mechanismen auch einfach sind, dies steigert Vertrauen (zB Zahlungssysteme).

Kryptrographie –Verschlüsselung

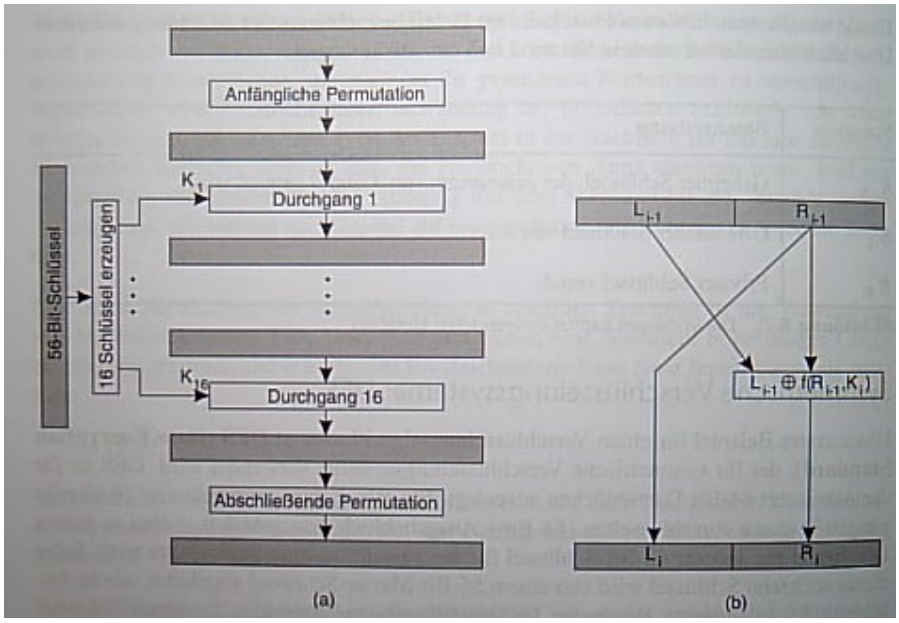


P=Klartext, K=Schlüssel, Ek=Verschlüsselung, Dk=Entschlüsselung.
Angriffsarten: Lauschangriff, Veränderung der Nachricht, Einfügen einer Nachricht.
Symmetrische Verschlüsselungssysteme: Verschlüsselung +Entschlüsselung mit selbem key
Assymetrische Verschlüsselungssysteme (Systeme mit öffentl. Schlüssel): Separate Schlüssel : privat und öffentlich
Hash-Funktionen: (=Einwegefunktionen): h=H(m)
Eigenschaften:

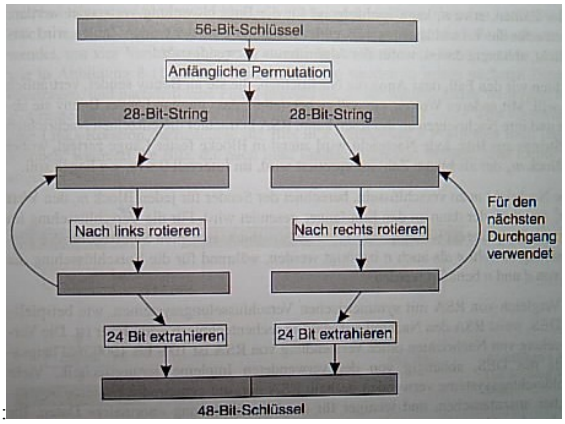
- **Einwege-Funktion:** Rechentechn. Unmöglich, Eingabe m zu ermitteln, wenn h bekannt ist
- **Schwacher Kollosionswiderstand**: rechentechn. Unmöglich, m' zu finden mit m=m', so dass H(m)=H(m') für h=H(m)
- **Starker Kollosionswiderstand**(gei kryptogr. Hash-Funktionen): Wenn nur H bekannt ist, rechentechn. Unmöglich, m' zu finden mit m=m'=>H(m)=H(m')

Symetrische Verschlüsselungssysteme: DES

DES=Data Encryption Standard
64-Bit-Block wird in 16 Durchgängen in verschlüsselten 64-bit-Ausgabeblock umgewandelt.
48 bit-Schlüssel in jedem Durchgang abgeleitet von 56 bit-Masterkey.
Algo schwierig zu knacken, mit Brute-Force möglich. Sicher durch 3-fache Anwendung mit Verschlüeeln-Entschlüsseln-Verschlüsseln-Modus.



Einzelne Permutationen:



Verschlüsselungssysteme mit öffentlichem Schlüssel: RSA

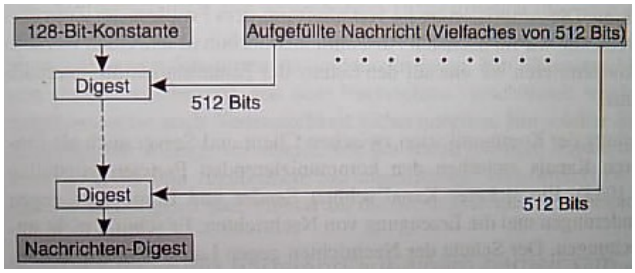
RSA: Rivest,Shamir,Adleman, beruht auf großen Primzahlen

1. Auswählen von 2 sehr großen Primzahlen p und q
2. Berechnen von n=p*q und z=(p-1)(q-1)
3. Auswählen von d, die relativ prim zu z ist
4. Berechnen der Zahl e, so das gilt: e*d=1 mod z

d zB für Entschlüsselung, **e** für Verschlüsselung, **n** muss bekannt sein für Entschlüsselung: max. Blocklänge

Hash-funktionen: MD5

Umwandeln der Nachricht in 512 bit Blöcke: 448 bit +64 bit Integer (Länge der ursprünglichen Nachricht).
Anfangsdigest konstant mit 4*32 bit Variablen p,q,r,s=128 bit.
Aufteilung des 512 bit –Blockes in 16*32 bit->16 Iterationen mit Funktione, die jeweils p,q,roder s aus den anderen 3 Variablen berechnet und in jeder Iteration eine Konstante aufaddiert.
4 Funktionen verfügbar0> 4*16 =64 Durchgänge pro 512-Block.
Nun neuer 128-bit-Digest für nächsten Block.
Übertragen wird der Nachrichten-Digest=End-Digest:



8.2 Sichere Kanäle

2 Aspekte:

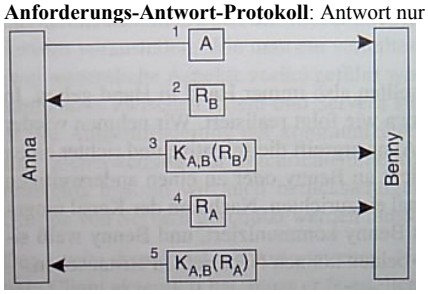
1. Authentifizierung + Sicherstellung der Nachrichtenintegrität (Nachrichtenveränderung während Übertragung) u.Vertraulichkeit
2. Autorisierung

Authentifizierung

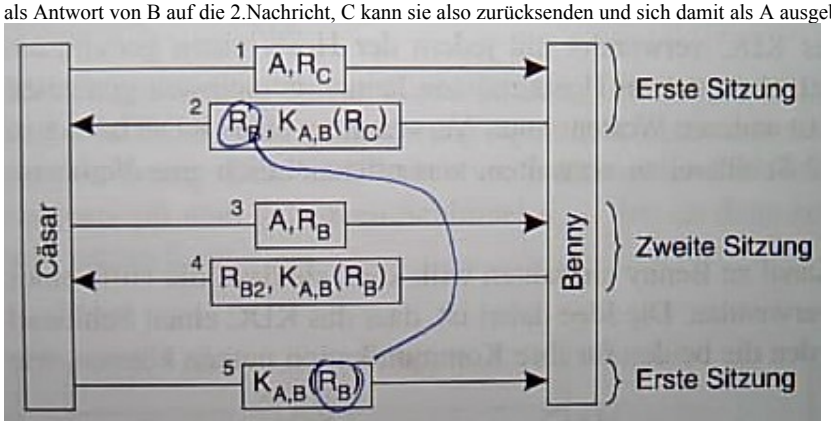
Authentifizierung und Nachrichtenintegrität kommen nicht ohne einander aus.
Für Integrität nach Authentifizierung: Verschlüsselung mit Sitzungsschlüsseln (gemeinsam genutzter schlüssel, um Nachrichten zu verschlüsseln- wird nach Sitzung zerstört)

Authentifizierung basierend auf einem gemeinsam genutzten Schlüssel

Gemeinsam genutzter Schlüssel K_a,b
Anforderungs-Antwort-Protokoll: Antwort nur korrekt, wenn der andere den Schlüssel kennt.

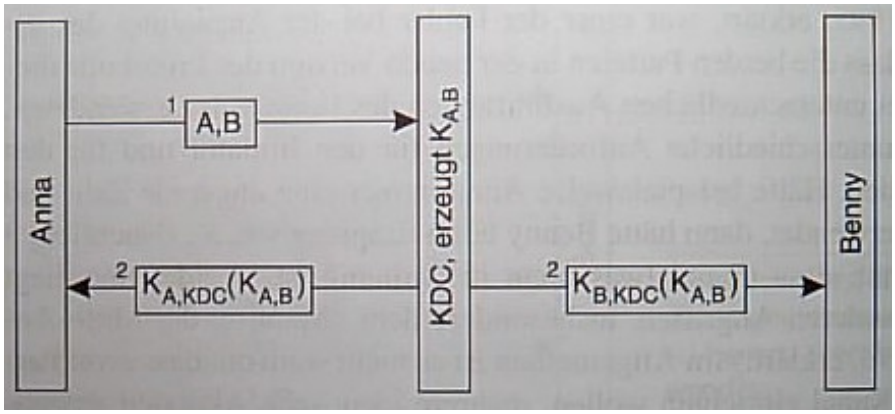


Angriff möglich mit Reflexions-Angriff: Angreifer C richtet 2.Sitzung ein, indem er unter Vorgabe, er sei A, 2 Anforderungen an B sendet: R_C und R_B. Dabei ist R_B die Nachricht, die B geantwortet hat, und die C verschlüsseln soll. Dieses verschlüsselte R_B kommt aber dann als Antwort von B auf die 2.Nachricht, C kann sie also zurücksenden und sich damit als A ausgeben:

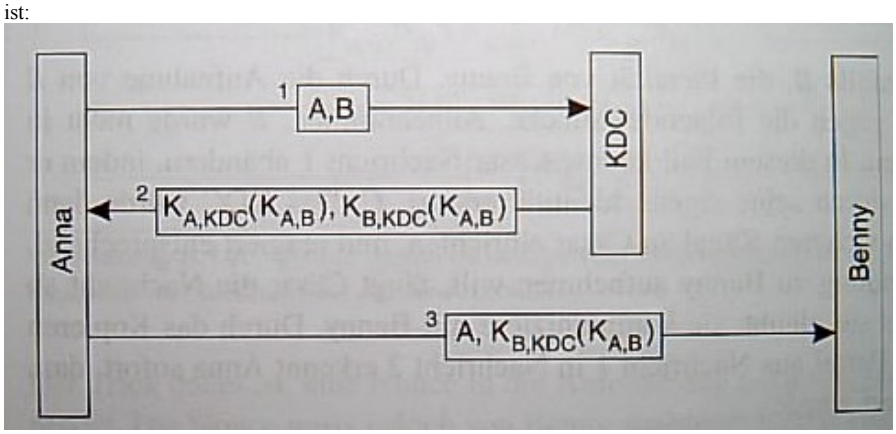


Authentifizierung unter Verwendung eines Schlüsselzentrums

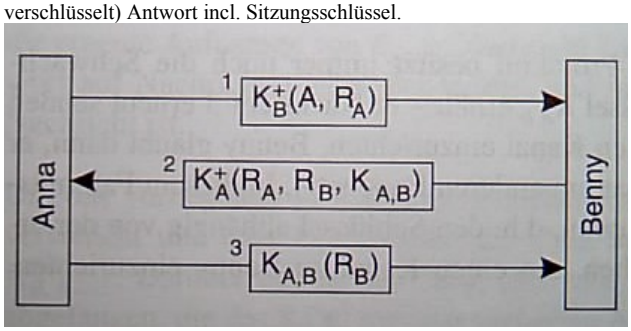
Normal: Bei N Hosts muss System N(N-1)/2 Schlüssel verwalten.
Alternative: **KDC** (Key Distribution Center): Nur noch N Schlüssel nötig.
KDC gibt schlüssel an A und B für Kommunikation aus(=Ticket)
Jeder Host besitzt einen Schlüssel für das KDC, dieser liefert verschlüsselt den gemeinsamen Schlüssel:



Variante: **Needham-Schroeder-Authentifizierungsprotokoll**:
Zusätzlich zur Verbindungsanfrage wird ein **Nonce** mitgesandt:=Zufallszahl zum Verknüpfen von Nachrichten.
Sehr sicher: Nonce wird vom Partner B verschlüsselt gesandt: K_B;KDC(R_B-1), dieses von A an den KDC geschickt, wird dort in das Ticket mit aufgenommen und von A wieder an B geschickt. Damit weiss B, dass die ursprüngliche Anforderung mit dem Schlüssel verknüpft ist:



Authentifizierung mit der Verschlüselung mit öffentlichem Schlüssel
Öffentliche Verschlüsselung A->B mit K_B+(A,R_A). Nur B kann dies mit seinem privaten Schlüssel entschlüsseln, sendet (öffentlich verschlüsselt) Antwort incl. Sitzungsschlüssel.

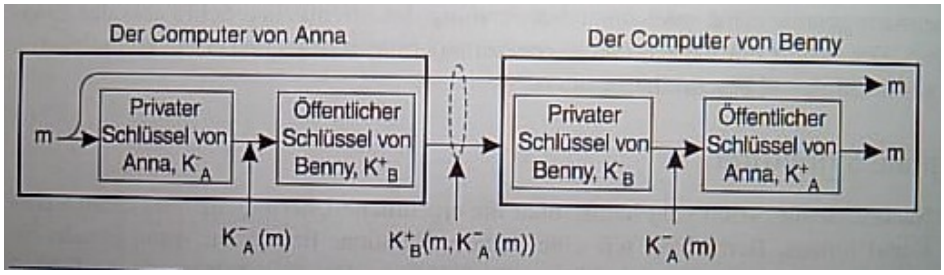


Nachrichtenintegrität und Vertraulichkeit

Nachrichtenintegrität: Nachricht wird vor Veränderung geschützt
Vertraulichkeit: Durch Verschlüsselung
Digitale Signaturen
Üblich: Verwendung von öffentlichen Schlüsseln.
Probleme:

- A kann vorgeben, Schlüssel sei ihm vor Sendung gestohlen worden
- Bei Privatschlüsseländerung werden alte Nachrichten wertlos
- Kostspielig (Rechenzeit)

Billiger und eleganter: Verwendung eines Nachrichten-Digest
Mit Hash-Funktion Nachrichten-Digest berechnen, dann privat verschlüsseln und senden, bei B entschlüsseln: Digest berechnen und Übereinstimmung prüfen:

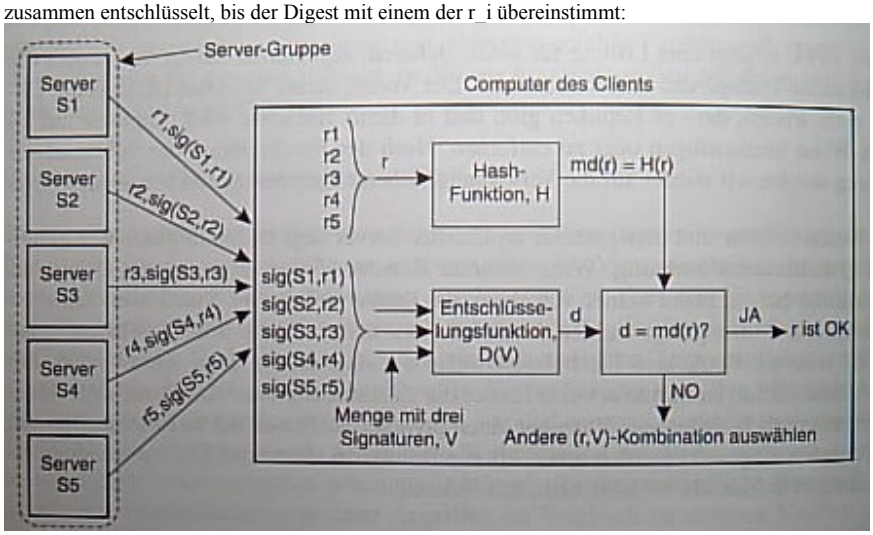


Sichere Gruppenkommunikation

Alle Gruppenmitglieder selber Schlüssel-> empfindlich gegen Angriffe
Pro Paar Mitglieder ein Schlüssel->n(n-1)/2 Schlüssel nötig
Jedes Mitglied hat ein Paar (öffentl./privat)->besser, nicht vertrauenswürdiges Mitglied kann leicht entfernt werden

Sicher replizierte Server

Problem: Angriff durch Übernahme eines oder mehrerer der Server
Lösung: Sammeln der Serverantworten und Authentifizierung, bei Mehrheit zwischen Antworten Korrektheit der Antwort. Problem hier: Replikationstransparenz
Lösung: Jeder Server sendet Nachricht (r_i,sig(S_i,r_i)) , sig(S_i,r_i)=k_i(md(r_i))=privat verschlüselter Nachrichtendigest.
Client:
Die r_i werden über Hash-Funktion zu md(r) berechnet.
Müssen mind. 3 Server vertrauenswürdig sein, so werden 3 verschiedene Signaturen sig(S_i,r_i) über eine Entschlüsselungsfunktion zusammen entschlüsselt, bis der Digest mit einem der r_i übereinstimmt:



Verbesserung:
(m,n)-**Schwellenwertschema**, verbessert die Replikationstransparenz: Nachricht wird in n Abschnitte zerlegt (n=Anzahl der Server) =**Schatten**- m Schatten werden benötigt um die Nachricht wiederherzustellen.
Server senden r_i + sig(S_i,r_i) an alle Server, hat ein Server mind. M Nachrichten erhalten, prüft er die Gültigkeit wie oben und falls positiv, sendet er r und die Menge der Signaturen an den Client, der überprüft, ob md(r)=D(V) .

8.3 Zugriffskontrolle

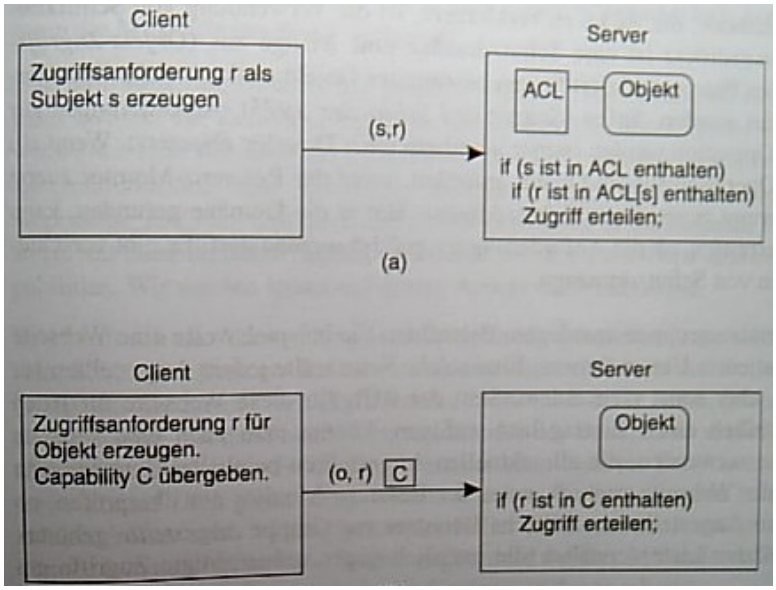
Zugriffskontrolle=Überprüfung der Zugriffsrechte
Autorisierung=Erteilung der Zugriffsrechte

Allg. Aspekte

Schutz Objekt vor Subjekt über Referenz-Monitor: Zeichnet auf, welches Subjekt was tun darf und entscheidet, welche Ops es ausführen darf.

Zugriffskontrollen-Matrix

Subjekt als Zeile, Objekt als Spalte. M[s,o] bezeichnet, welche Ops das Subjekt ausführen darf.
Effizient:
ACL (Access Control List): Matrix ist spaltenweise über die Objekte verteilt, leere Einträge werden weggelassen.
Capabilities: Matrix ist reihenweise auf die Subjekte verteilt, leere Einträge werden weggelassen



Schutzdomänen

Dient zum Verkleinern der Listen.

Schutzdomäne=(Objekt-,Zugriffsrechte-)Paar, jedes Paar spezifiziert für ein Objekt genau, welche Op ausgeführt werden darf.

Subjekt fordert Op für Objekt an->Referenzmonitor sucht zuerst die dieser Anforderung zugeordnete Schutzdomäne->dann Überprüfung, ob Anforderung ausgeführt werden darf.

zB Benutzergruppen (Angestellte dürfen auf Website). Statt ACL einen Eintrag für jeden Angestellten hinzuzufügen-> separate Gruppe Angestellte mit allen Angestellten anlegen->Referenzmonitor muss dann nur prüfen, ob Benutzer = Angestellter.

Noch besser: hierarchische Gruppen. Mitgliedschaft in Untergruppen impliziert Zugang zu Obergruppen. Nachteil: kostspielig

=> Jedem Subjekt ein Zertifikat geben, mit Info welcher Gruppe er angehört.

Schutzdomäne als Rollen: Benutzer kann mehrere Rollen haben, unterschiedliche Rechte je nach Rolle

Objekte hierarchisch gruppieren (SSvererbung)

Firewalls

Firewall=spezieller Referenzmonitor, der externe Zugriffe auf Teile eines verteilten Systems kontrolliert.

2 Typen:

1. Paketfilternder Gateway: Router, der entscheidet, ob Paket durchgelassen wird. Untersucht header.
2. Gateway auf Applikationsebene: untersucht Inhalt zB Spam-Filter. Speziell: Proxy-Gateway-gibt nur solche Nachrichten weiter, die bestimmte Kriterien erfüllen.Verwirft zB ausführbaren Code

Sicherer mobiler Code

Agenten müssen vor böswilligem Host geschützt werden und vice versa

Einen Agenten schützen

Kein Schutz vor Änderungen möglich, wohl aber können Änderungen erkannt werden:

=> Ajanta-System-hat 3 Mechanismen:

1. schreibgeschützter Status: besteht aus mehreren vom Eigentümer signierten Datenelementen. Bei Erzeugung wird Nachrichtendigest erzeugt und privat verschlüsselt. Erkennen einer Beschreibung durch Vergleich signierter Nachrichtendigest zu ursprünglichem Status.
2. Sichere Protokolle: können nur ergänzt werden. Anfang: Protokoll leer, mit zugeordneter Prüfsumme C_init = K+(N) , N=geheime nur Eigentümer bekannte Nonce.

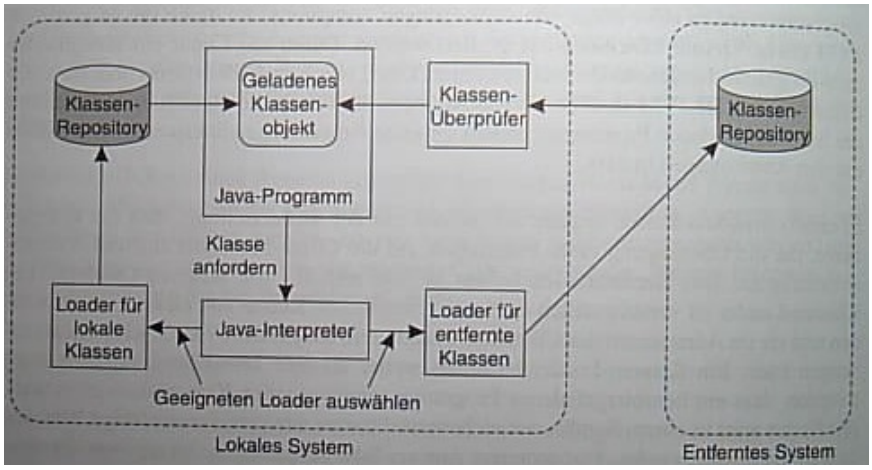
   Anfügen von Daten: Anfügen und Verschlüsseln an das Protokoll: C_neu=K+(C_alt,sig(S,X),S)

   Prüfen: Für jede Iteration K-(C)=C_next. Wird so lange gemacht, bis Prüfsumme=Anfangssumme.
3. Selektive Offenlegung des Status: Bereitstellung Array von Datenelementen, jedes öffentlich verschlüsselt für je einen Server; gesamtes Array wird zur Integritätswahrung vom Eigentümer signiert

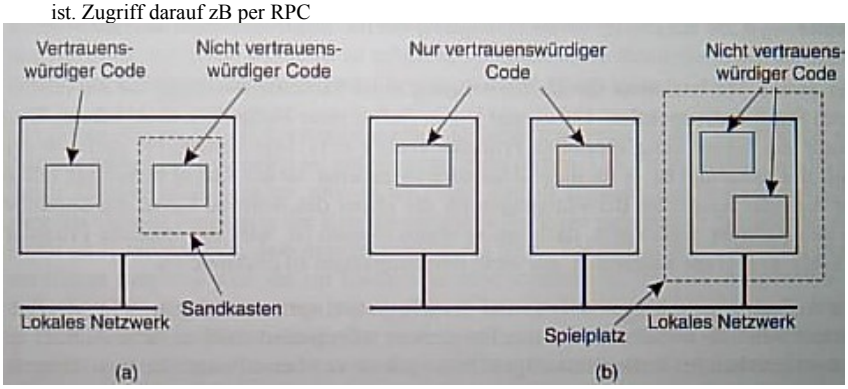
Das Ziel schützen

- Durch **Sandkasten**: Technik, wobei ein heruntergeladenes Programm so ausgeführt wird, dass jede seiner Anweisungen vollständig kontrolliert werden kann. Bei verbotenen Anweisungen oder Zugriffe auf nicht freigegebene Speicher->Beenden.

Implementierung schwierig, einfacher mit interpretiertem Code (Java: Classloader laden Programm herunter, installiert im Adressraum des Clienten. Bytecode-Überprüfung prüft, ob es den Sicherheitsregeln des Sandkastens entspricht. Danach Objektinstantiierung. Sicherheits-Manager übernimmt verschiedene Überprüfungen zur Laufzeit, verweigert zB Zugriff auf lokale Dateien)

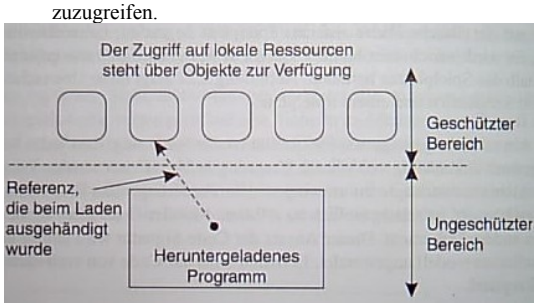


- Durch **Spielplatz**:=separate, speziell ausgelegte Maschine, die exclusiv für Ausführung von mobilem Code reserviert ist. Zugriff darauf zB per RPC



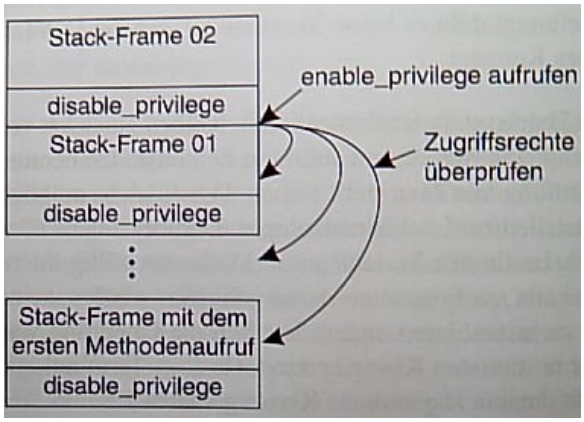
Flexibler: Sicherheitsstrategie- 3 Mechanismen:

1. **Verwendung von Objekt-Referenzen als Capabilities**: Ohne Referenz keine Möglichkeit, auf Dateien zuzugreifen.



2. **Stack-Introspektion**: jedem Aufruf einer Methode einer lokalen Ressource geht voraus: enable_privileg und endet: disable_privileg

Einfügen dieser Aufrufe in SS aufwändig, besser wenn Java-Interpreter das übernimmt (zB bei Applets der Fall). Bei jedem Aufruf einer lokalen Ressource ruft Interpreter enable_Privileg auf und prüft auf Erlaubnis. Falls ja, wird gleich disable-privileg auf Stack gelegt.



3. **Verwaltung des Namensraumes**: Programme müssen Dateien einbinden, bevor sie Zugriff auf Ressourcen erhalten-dafür muss ein Name für die zur Laufzeit geladene Klasse an den Interpreter gegeben werden.=> derselbe Name kann in unterschiedlichen Klassen aufgelöst werden,je nach Autorisierung(wird von classloader übernommen.)

8.4 Sicherheitsmanagement

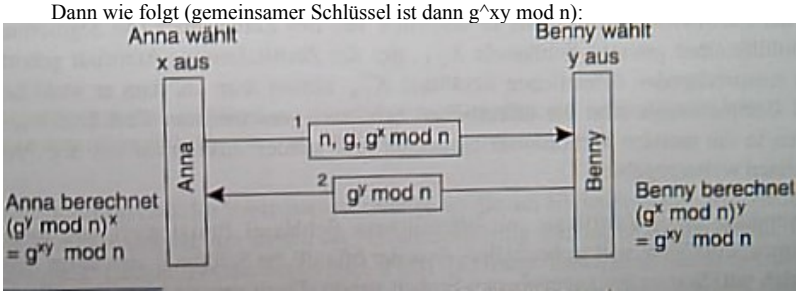
Schlüsselverwaltung

Einrichtung von Schlüsseln

a) A holt öffentlichen Schlüssel von B für Kommunikation und umgekehrt
b) Ähnlich der Erzeugung und Verteilung eines Sitzungsschlüssels

a und b setzen sicheren Kanal voraus, also bereits eine Form der Erzeugung und Verteilung von Schlüsseln.

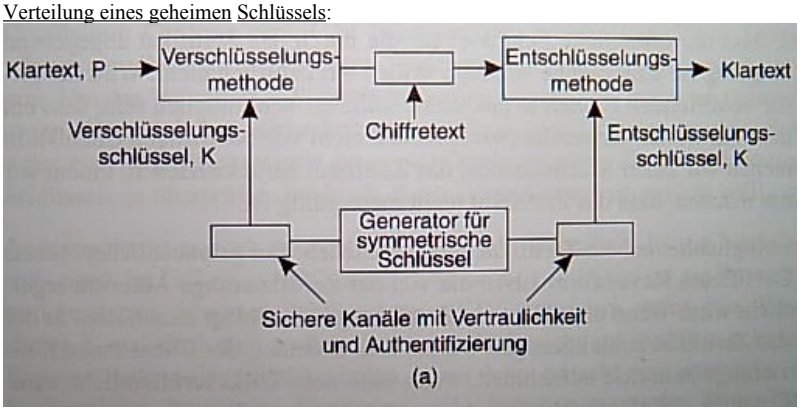
c) Einrichtung eines gemeinsam genutzten Schlüssels über unsicheren Kanal: **Diffie-Hellman-Schlüsselaustausch:** Einigung auf 2 große Zahlen, n und g, dies kann über unsicheren Kanal geschehen. Dann wie folgt (gemeinsamer Schlüssel ist dann g^xy mod n):



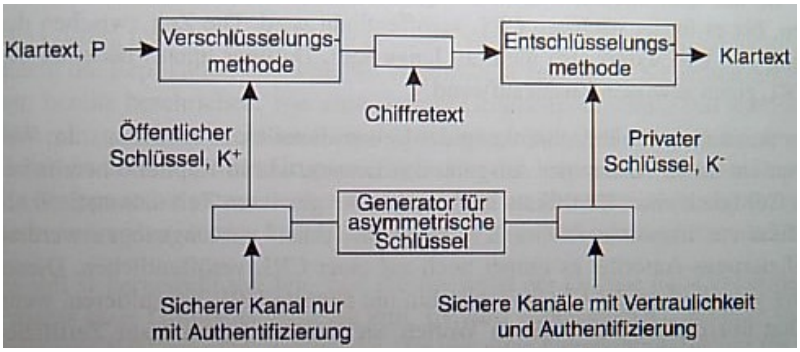
Schlüsselverteilung

a) Post, Telefon
b) Öffentlicher Schlüssel: Authentifizierung (kommt wirklich vom Absender), privater Schlüssel: Authentifizierung und Vertraulichkeit

Verteilung eines geheimen Schlüssels:



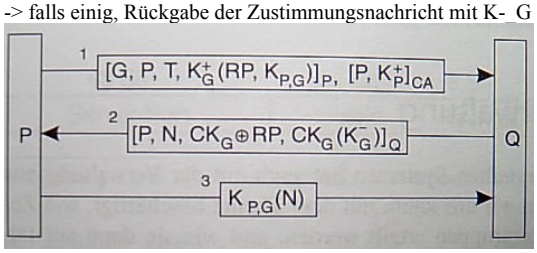
Verteilung eines öffentlichen Schlüssels:



Authentifizierte Verteilung öffentlicher Schlüssel: mit **Zertifikaten für öffentliche Schlüssel**= öffentlicher Schlüssel+ID-String(Benutzer, Host,..). Beide durch Zertifizierungsautorität signiert, ebenfalls auf Zertifikat abgelegt. Öffentlicher Schlüssel K+_CA meist in Browsern eingebaut, mit diesen wird das Zertifikat auf (öffentlicher Schlüssel-, ID-)Paar geprüft.
Hierarchische Vertrauensmodelle: Falls der Client dem Zertifikat nicht traut, kann er evtl auf höherer Ebene Authetifizierung anfordern (zB **PEM**-Privacy Enhanced Mail)
Lebensdauer von Zertifikaten
Normal 1 Jahr, Gültigkeit veröffentlicht in CRL (Certifikate Revocation List)

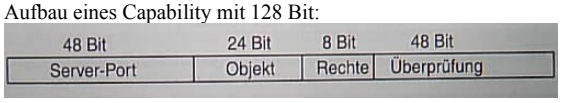
Sichere Gruppenverwaltung

Bei KDCs und CAs höchste Verfügbarkeit nötig-> Replikation nötig, Sicherheit durch gemeinsamen Schlüssel.
Vewaltung replizierter Server:
Voraussetzungen: Gemeinsamer Schlüssel CK_G für Gruppennachrichten, Paar(K+,K-) für Kommunikation mit Nichtmitgliedern.
P will G beitreten:
-> (Join Request), besteht aus: G,P,lokaler Zeit von P,erzeugte Antwortmöglichkeit RP,erzeugten geheimen Schlüssel K_P,G, RP und K_P,G werden mit K+_CA verschlüsselt, JR wird signiert und zusammen mit Zertifikat gesendet, das den öffentlichen Schlüssel von P enthält.
-> Q authentifiziert P, kontaktiert andere Gruppenmitglieder wegen Aufnahme von P
-> falls einig, Rückgabe der Zustimmungsnachricht mit K- G

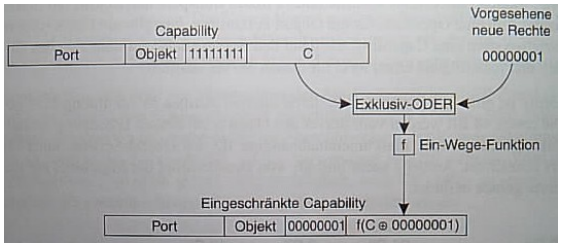


Autorisierungsverwaltung

Netzwerkbetriebssysteme: Auf jeder Maschine ein Account pro Benutzer
Einfacher: 1 Account auf zentralem Server
Capabilities und Attribut-Zertifikate
Besser: Capabilities (fälschungssichere Datenstruktur für spezifische Ressource, die genau angibt, welche Zugriffsrechte der Eigentümer im Hinblick auf diese Ressource besitzt.)
Implementierung in BS Amoeba:
Objektbasiert (Objekt auf Server, Clientzugriff durch Proxies)
Client ruft Op für Objekt auf->Client übergibt seinem lokalen BS Capabilities-> dieses sucht Server mit Objekt und führt RPC dort aus.
Aufbau eines Capability mit 128 Bit:



Suche des Servers mittels Broadcasting.
Bei Objekterzeugung wählt der Server zufälliges Überprüfungsfeld aus->Speicherung in 1.Capability,2. internen Tabellen.
Alle Rechte-Felder sind auf 1 gesetzt-> Rückgabe dieses **Eigentümer-Capability** an Client.
Client will Rechte einschrönken->Rückgabe des Capability+Bit-Maske für neue Rechte an Server-> dort XOR-Verknüpfung des Überprüfungsfeldes mit neuen Rechten, Schicken durch 1-Wege-Funktion und Erzeugung neuer Capablility-> Rückgabe an Client, dieser kann es weitergeben.
Bei Objektaufruf erkennt Server anhand des Rechte-Feldes, dass es sich nicht um die Eigentümer-Capability handelt-> Server lädt interne Zufallszahl, verknüpft diese mit Rechte-feld+Ein-Wege-Funktion und vergleicht dies mit dem Überprüfungsfeld
=>fälschungssicher!



Moderne verteilte Systeme: **Attribut-Zertifikat**: Listen bestimmte (Attribut-,Wert-)Paare für bestimmte Einheit auf, insbesondere Zugriffsrechte, Ausgabe von sog. **Attribut-Zertifizierungs-Autoritäten.**

Delegierung

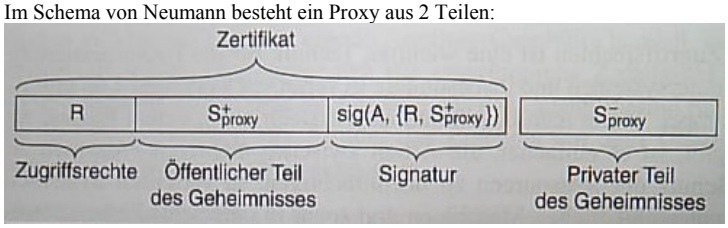
Von Zugriffsrechten, temporär.

Delegierung zB über Proxy=Token, für Erlaubnis derselben Rechte wie Subjekt, das Token erteilt hat- Proxy-Erzeugung nur mit max. den Rechten des Erzeugers.

a. Delegierung einfach, wenn Proxy-Erzeuger alle kennt.Bei Weitergabe von B nach C muss C A um Zertifikat bitten

b. Zertifikaterzeugung mit Blanko-> muss gegen unerlaubtes Kopieren geschützt werden.

Im Schema von Neumann besteht ein Proxy aus 2 Teilen:



Schutz vor Veränderung: mithilfe Signatur

Schutz vor Diebstahl: S+_proxy=komplizierte Frage, S-_proxy=Antwort.

Reicht B Proxy an C weiter, stellt C die Frage an B-> ist Antwort richtig, weiss C, dass Rechte von A kommen.

Vorteil: A muss nicht konsultiert werden!

8.4 Kerberos

Basiert auf Needham-Schroder-Identifizierungsprotokoll.

=Sicherheitssystem, das Clients hilft, einen sicheren Kanal zum Server einzurichten.

2 Komponenten:

1**. Authentifizierungsserver (AS)**: Authentifiziert Benutzer und stellt Schlüssel für TGS bereit

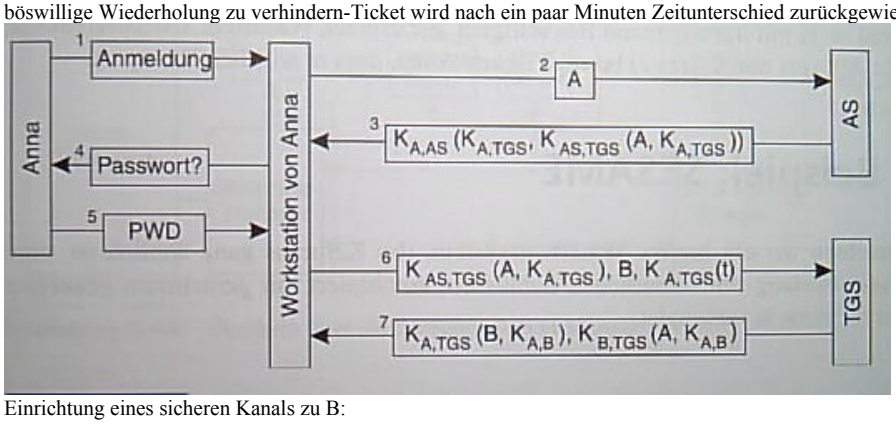
2. **Ticket Granting Service (TGS)**: Stellt spezielle Nachrichten + Tickets aus, um Server die Echtheit des Clients zu bescheinigen.

Will A Verbindung mit B einrichten, sendet sie ihren Namen im Klartext an AS.

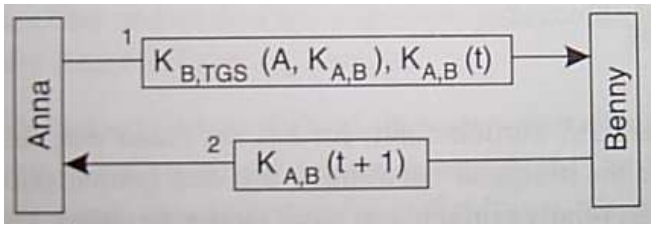
Server gibt Sitzungsschlüssel K_A,TGS zurück sowie Ticket für TGS (enthält ID von A und gemeinsamern Schlüssel für A-TGS-Kommunikation), alles verschlüsselt mit geheimem Schlüssel zw. A und AS (wegen Vertraulichkeit).Vorteil: Schlüssel muss nicht gespeichert werden

Danach Aufforderung an A zu Passworteingabe, damit wird der gemeinsame Schlüssel K_A,AS erzeugt.

Damit kann die Antwort von AS entschlüsselt werden und mit der Bitte um Schlüssel für B an TGS geschickt werden (inkl Zeitstempel, um böswillige Wiederholung zu verhindern-Ticket wird nach ein paar Minuten Zeitunterschied zurückgewiesen)



Einrichtung eines sicheren Kanals zu B:



KE5

10.1 Sun Network File System (NFS)- meist Unix

NFS-Überblick

Grundlegendes Konzept: Jeder Dateiserver stellt standardisierte Sicht seines lokalen Dateisystems bereit.
Grundsätzliche Architekturen

1. **Entferntes Zugriffsmodell (NFS):** Transparenter Zugriff auf Dateisystem, von entferntem Server verwaltet. Clients kennen die reale Position der Dateien nicht, erhalten nur die Schnittstelle des Dateisystems, der Server ist für Implementierungen der Ops verantwortlich, Dateien bleiben auf Server
2. **Upload/Download-Modell (CODA):** Client lädt Datei herunter, greift lokal darauf zu, lädt sie nach Gebrauch wieder hoch.

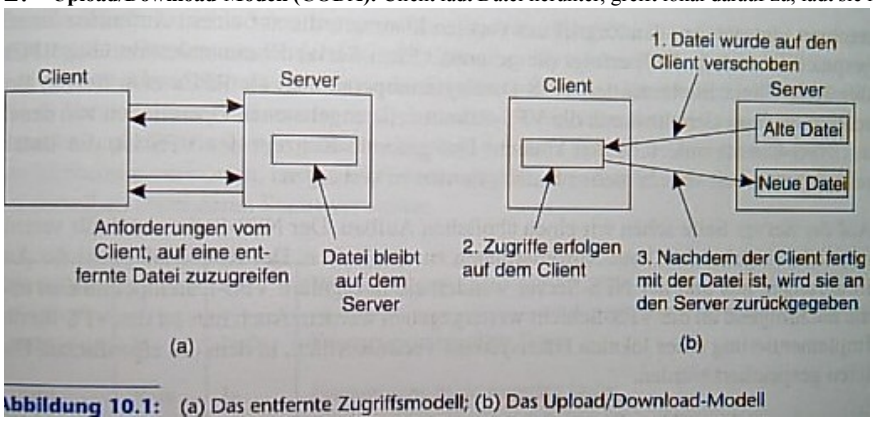
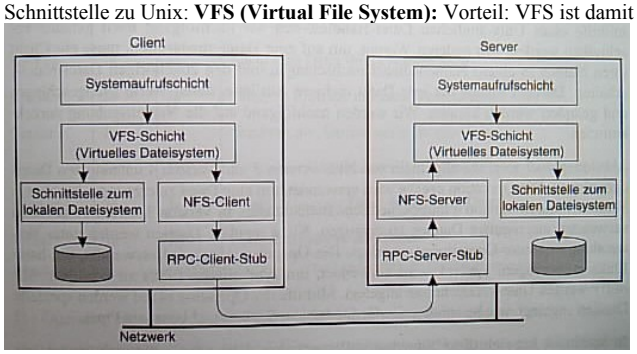


Abbildung 10.1: (a) Das entfernte Zugriffsmodell; (b) Das Upload/Download-Modell

Schnittstelle zu Unix: **VFS (Virtual File System):** Vorteil: VFS ist damit von lokalen Dateisystemen unabhängig:



Dateisystem-Modell
Ähnlich Unix-basierten Modellen: Dateien =nicht interpretierte Byte-Folgen
DateiOps oft andere Bedeutung zwischen NFS3 und 4

Kommunikation

NFS-Protokoll basierend auf RPC-Schicht, die Unterschiede zwischen verschiedenen BS verbirgt.
Kommunikation Client-Server gemäß **ONC RPC**-Protokoll (Open Network Computing)
Bis V4 ein Aufruf pro Op (zB LOOKUP, dann READ), ab V4 Unterstützung von zusammengesetzten Prozeduren :

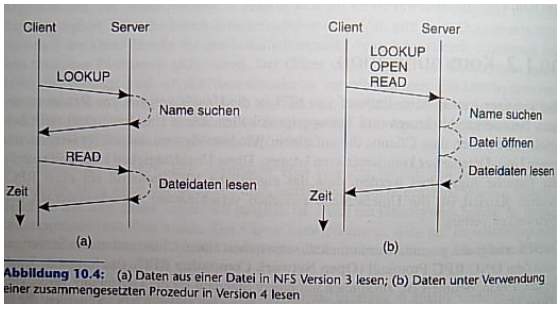


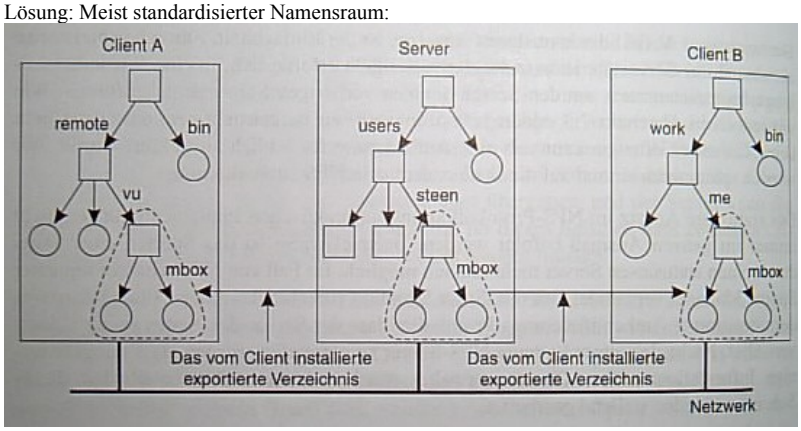
Abbildung 10.4: (a) Daten aus einer Datei in NFS Version 3 lesen; (b) Daten unter Verwendung einer zusammengesetzten Prozedur in Version 4 lesen

Prozesse

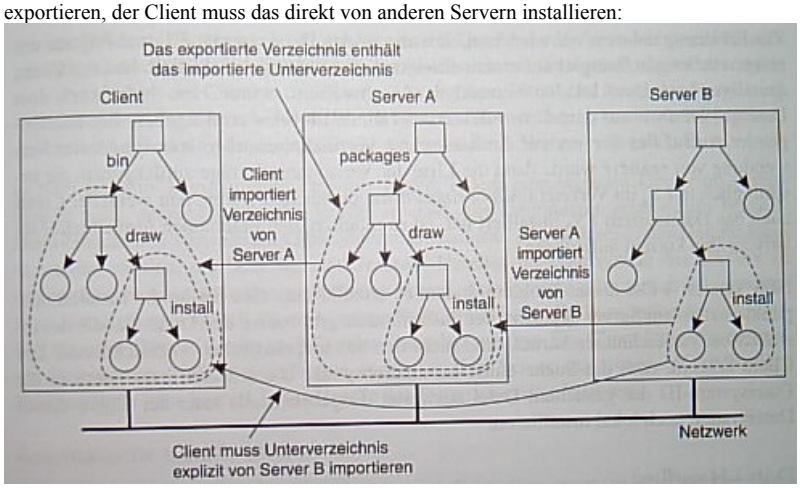
Ab V4: Server statusbehaftet, vor allem wegen Weitverkehrsnetzen-> hier effizientes Cache-Konsistenzprotokoll erforderlich, weil Clients Cache brauchen. Server verwaltet hier bestimmte Infos über Dateien seiner Clients (zB. **Lease**= zeitbegrenzte Lese- und Schreibberechtigung)

Namensgebung

Idee: Transparenter Zugriff auf entferntes Dateisystem (von Server verwaltet) für Clients.
Transparenz durch Ermöglichung, (Teile eines) entfernten Dateisystems im eigenen lokalen Dateisystem zu installieren.
Server exportiert ein Verzeichnis=Server stellt Verzeichnis+Einträge Clients zum Installieren zur Verfügung.
Problem: kein einheitlicher Namensraum auf Clients
Lösung: Meist standardisierter Namensraum:



NFS-Server kann Verzeichnisse installieren, die von anderen Servern exportiert werden->darf aber dann diese nicht an seine Clients exportieren, der Client muss das direkt von anderen Servern installieren:



Namesauflösung in V3 iterativ (nur 1 Datei gleichzeitig nachzuschlagen), in V4 rekursiv (vollständiger Pfadname kann übergeben werden).
NFS3 gibt beim Lesen eines installierten Verzeichnisses den ursprünglichen Inhalt zurück. NFS4 erlaubt Überschreitung der Installationspunkte auf einem Server.

Datei-Handles

Datei-Handle = Referenz auf Datei in einem Dateisystem. Wird erzeugt bei Erzeugung der Datei,ist undurchsichtig (Client erfährt nichts über Inhalt).
Idealfall: Datei-Handle ist echte ID für eine Datei relativ zu einem Dateisystem.

Vorteil:

- Bei Datei-Op muß nicht jedes Mal ein Name nachgeschlagen werden

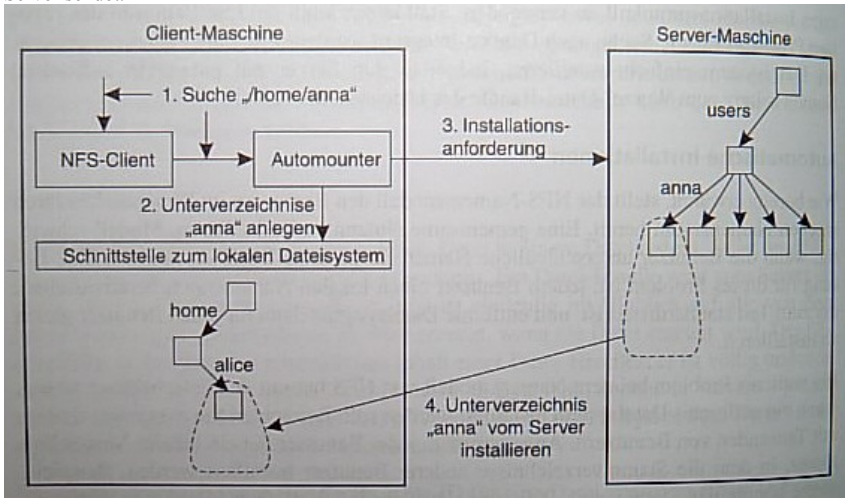
- Client kann unabhängig von ihrem (aktuellen) Inhalt auf die Datei zugreifen.

Nach der Installation wird dem Client ein **Wurzel-Datei-Handle** zurückgegeben, den er nachfolgend als Ausgangspunkt für die Suche nach Namen verwenden kann.

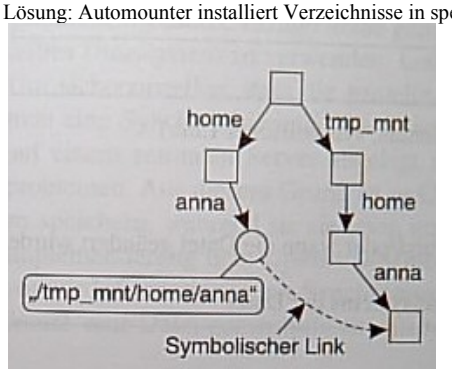
Automatische Installation

Problem: Eigener Namensraum pro Benutzer->ein anderer Name für dieselbe Datei.
Lösung: Bereitstellung von (standardisiertem) lokalen Namensraum pro Benutzer, entfernte Dateisysteme für jeden Benutzer gleich installieren.

Problem: Wann installieren? Bsp: Anmeldung Anna, Stammverzeichnis /home/anna steht ihr zur Verfügung, aber Dateien sind auf entferntem Server=>autom. Installation bei Anmeldung. Hat sie Zugriff auf /home/benny, soll dann Bennys Stammverzeichnis auch automatisch installiert werden? Viel Kommunikation erforderlich!
Lösung: Realisierung durch **Automounter** (=separater Prozeß), transparente Installation.
Zugriff auf /home->Unix-Kernel gibt lookup-Op an NFS-Client weiter, dieser an den Automounter, der die Installationsanforderung an den Server sendet.



Problem: wegen Transparenz muss Automounter an allen Ops beteiligt sein, und muss wissen, ob Daten zur Verfügung stehen etc.
Lösung: Automounter installiert Verzeichnisse in speziellen Unterverzeichnissen, und legt symbolischen Link darauf:



Dateiattribute

Unterscheidung: Zwingende Attribute und empfohlene Attribute. Array von (Attribut-,Wert-)Paaren.
Zwingend zB TYPE,SIZE,CHANGE,FSID

Synchronisierung

Die Semantik gemeinsam genutzter Daten

Unix-Semantik	Jede DateiOp ist unmittelbar für alle Prozesse sichtbar. Zeitreihenfolge, gibt immer neuesten Wert zurück. Lese- und SchreibOps gehen direkt an den Server (nur einer!). Ineffizient, Besserung durch Caching
Sitzungssemantik	Änderungen werden erst dann für andere Prozesse sichtbar, wenn die Datei geschlossen wird. Lokale Caches, schliesst Client die Datei, wird Kopie an Server gesendet. Entspricht upload/download-Modell. Bei gleichzeitigem Hochladen: Zufallsauswahl
Nicht änderbare Dateien	Alle Dateien schreibgeschützt, können aber durch neue Dateien ersetzt werden. Gleichzeitiges Ersetzen: Zufallsauswahl Ersetzen, während B liest: 1. weiterhin lesen oder 2. Leseversuche scheitern
Transaktionen	Alle Änderungen finden atomar statt

Dateisperren in NFS

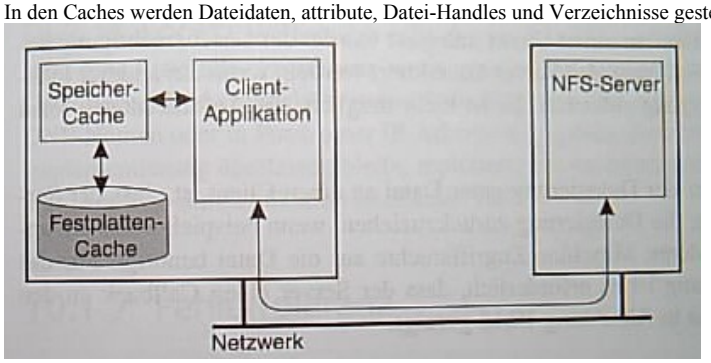
Unter NFS3 mit separatem Protokoll, ab V4 integriert.

4 Ops: Lock, lockt (fest), locku (unlock),renew (lease erneuern)
Sperren werden für bestimmte Zeit erteilt, sonst bei Absturz des Clients Blockaden.
Möglichkeit der **genauen Reservierung**: Benötigt Zugriffstyp in Kombination mit Verweiterungszugriffstyp für andere Clients:

	NONE	READ	WRITE	BOTH
READ	OK	NOT	OK	NOT
WRITE	OK	NOT	NOT	NOT
BOTH	OK	NOT	NOT	NOT

Caching und Replikation

Client-Caching
Allgemeines Modell, das von NFS vorausgesetzt wird: Jeder Client kann Speichercache haben, der Zugriff auf Festplattencache hat-
In den Caches werden Dateidaten, attribute, Datei-Handles und Verzeichnisse gestellt:



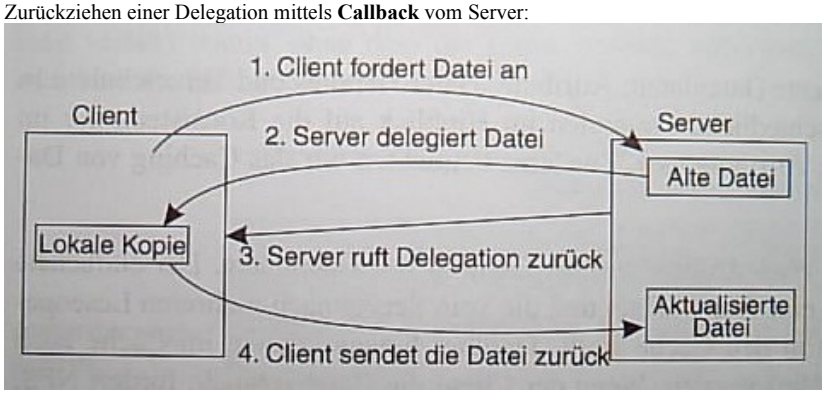
2 Ansätze für Caching von Dateidaten:

1. Client öffnet Datei und stellt Daten nach vollständigem Laden in Cache

Auch Schreibops in Cache möglich-nach Schliessen der Datei muss diese dann an Server zurückgeschrieben werden.
Öffnet Client eine Datei im Cache, die er schon mal geschlossen hatte, müssen alle Daten ungültig gemacht werden.

2. Server kann auch einige seiner Rechte an Client delegieren:

Offene Delegierung :
Server delegiert open/close-Ops an Client, Server muss dann nicht mehr kontaktiert werden und weist Zugriffe anderer Clienten ab.
Wurde eine Datei nur mit Leseberechtigung delegiert, darf dieser Client anderen lokalen Clienten aber keine Schreibberechtigung erteilen, hier muss der Server kontaktiert werden.
Zurückziehen einer Delegation mittels **Callback** vom Server:



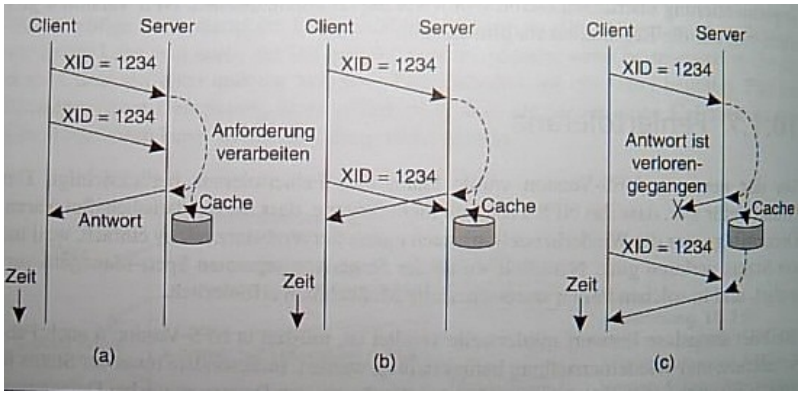
Veränderungen an Attributwerten sollten unmittelbar an Server geschrieben werden=write through.
Leases werden verwendet,um inkonsistenzen abzuschwächen.
Datei-Replikation: Nur mit ganzen Dateisystemen möglich (=Datei, Attribute, Verzeichnisse, Datenblöcke). Das Attribut FS_LOCATIONS enthält Liste mit Positionen, wo das zugehörige Dateisystem der Datei sich befinden kann (DNS-Name, IP-Adresse).

10.1.7 Fehlertoleranz

Nötig wegen Statusbehaftung, davor unnötig.
Status in 2 Fällen zu berücksichtigen:

1. Sperre von Dateien
2. Delegierung

RPC-Fehler
Größtes Problem: Doppelte Anforderungen
Lösung: **Cache für doppelte Anforderungen:** Jede Anforderung trägt **eindeutige Transaktions-ID (XID)** im Header.
Erfolgt doppelte Anfrage vor oder kurz nach einer Serverantwort, ignoriert Server diese.
Erfolgt doppelte Anfrage eine Zeitspanne nach Serverantwort: Server folgert, dass Antwort verlorenging->erneute Antwort



Dateisperren bei Vorliegen von Fehlern

Lease: Client muss stets renew-Anforderungen stellen, um Sperre für Dateien zu erneuern.
Erreicht Client Server nicht rechtzeitig, hat er noch eine **Gütezeit-** innerhalt dieser akzeptiert Server nur renew-Ops, keine neuen Sperren.
Probleme mit Leases:

- Synchronisierte Uhren nötig
- Zuverlässigkeit für renew-Nachricht: kann verloren gehen oder verzögert werden und läuft dann ab.

Offene Delegierung bei Vorliegen von Fehlern

Probleme:

- Clientabsturz (delegiert): Client ist zum Teil für Wiederherstellung der Dateien verantwortlich.
- Serverabsturz: Bei Wiederherstellung erzwingt Server von jedem Clienten der renew-Anforderungen stellt, seine Daten auf Server zurückzuschreiben-> Delegation wird zurückgenommen

Sicherheit

Hauptsächlich zwischen Client und Server gemeint.
Erforderlich: Sichere RPCs und Dateizugriffskontrolle:

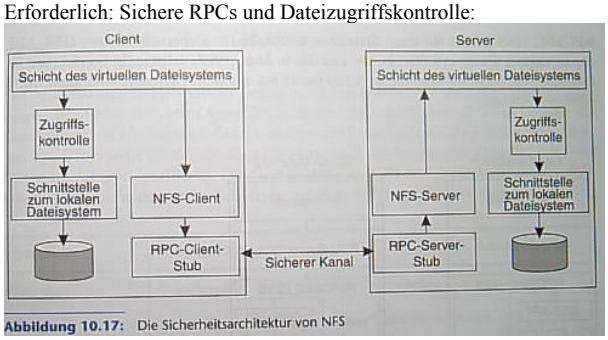


Abbildung 10.17: Die Sicherheitsarchitektur von NFS

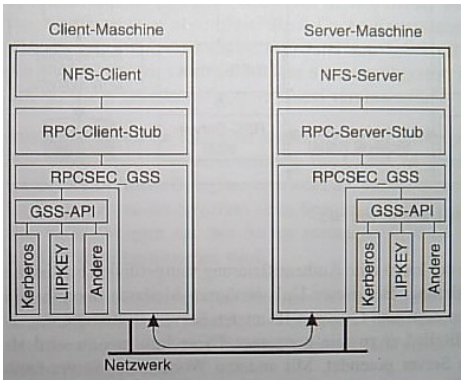
Sichere RPCs

Bis V4: Sicherer Kanal=sicherer RPCs=Authentifizierung
3 Möglichkeiten:

1. System-Authentifizierung: Nicht signierter Klartext mit User-ID an Server. Server setzt konkrete Anmeldung und Vertrauen in die Clientmaschine voraus.
2. Diffie-Hellman-Schlüsselaustausch
3. Kerberos

Ab V4: Sicherheitserweiterung durch Unterstützung von RPCSEC_GSS:

- Unzählige Sicherheitsmechanismen für Einrichtung sicherer Kanäle
- Bereitstellung von Ankerpunkten für verschiedene Authentifizierungssysteme
- Unterstützt Nachrichtenintegrität und Vertraulichkeit
- Basiert auf Standardschnittstelle GSS_API zB können implementiert werden: Kerberos, LIPKEY (=System öffentlicher Schlüssel, welcher ermöglicht, dass Clients unter Verwendung von Passwort und öffentlicher Schlüsssel auf Server identifiziert werden können.)



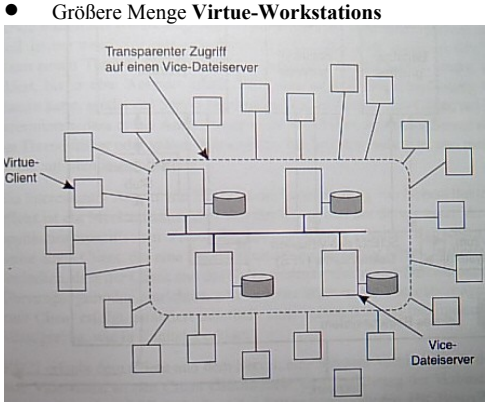
Zugriffskontrolle
Dateiattribut ACL stellt Liste mit Zugriffskontrolleinträgen dar, jeder für bestimmten Benutzer (-gruppe) spezifiziert (zB Read-data=Berechtigung, Datei zu lesen)

10.2 Dateisystem Coda

Überblick

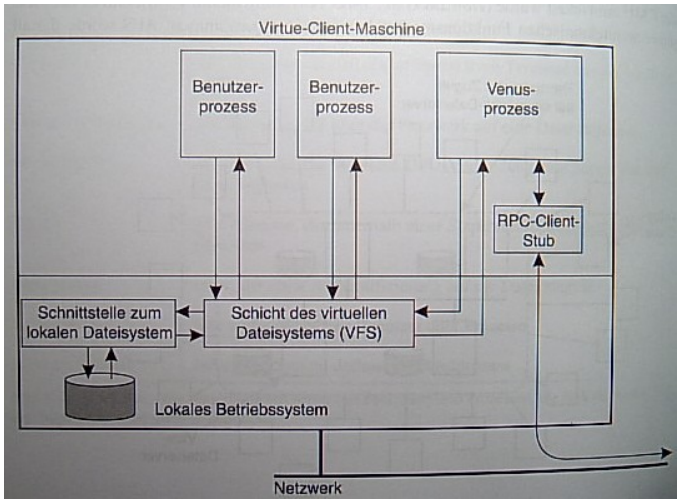
Hauptunterschied zu NFS: höchste Verfügbarkeit, Client kann auch ohne Server Betrieb fortsetzen
Ziel: hohes Maß an Namens- und Positionstransparenz, so dass Eindruck eines rein lokalen Systems entsteht
Ableitung von AFS (Andrew File System).
Zwei Gruppen:

- Zentral administrierte **Vice-Server**
- Größere Menge **Virtue-Workstations**



Clients:
In jeder Virtue-Workstation ist ein Prozess auf benutzerebene:=**Venus**
1. dafür verantwortlich, Zugriff auf die Datei bereitzustellen, die von den Vice-Dateiservern verwaltet wird.
2. dafür verantwortlich, Client zu ermöglichen, Betrieb fortzusetzen, selbst wenn Zugriff auf die Dateiserver nicht möglich ist.

VFS-Schicht fängt Aufrufe von Client-Applikationen auf.
Venus kommuniziert mit Dateiservern über RPC-Client-Stub auf Benutzerebene in UDP, Höchstens-einmal-Semantik.



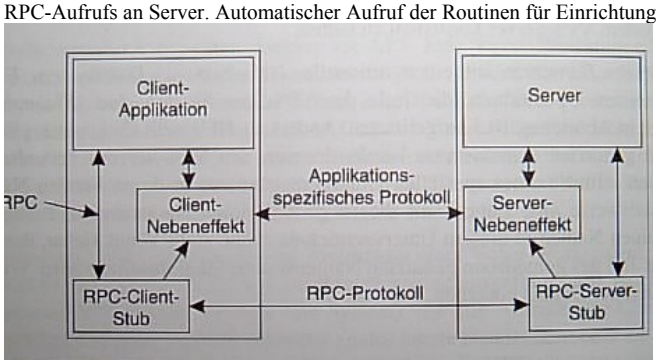
Server:
3 Prozesse:
1. Dateiserver
2. Authentifizierungsserver
3. Aktualisierungsprozesse (zur Konsistenzerhaltung)
Weitere Merkmale: Gemeinsam genutzter Namensraum, von Vice-Servern verwaltet

Kommunikation

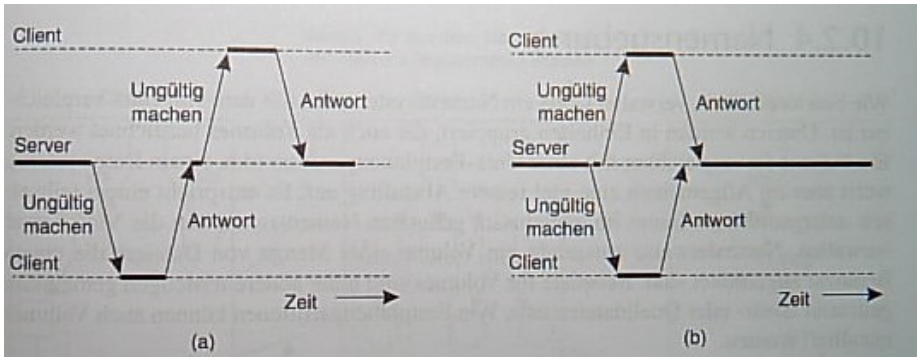
Kommunikation zwischen Prozessen mit **RPC2-System**
Unterstützt zuverlässige RPCs über (unzuverlässiges) UDP:
Aufruf entfernter Prozedur-RPC2-Client-Code startet neuen Thread->Aufrufanforderung an Server->Blockade, bis Antwort.
Server sendet regelmäßig Nachrichten, dass Anforderung noch verarbeitet wird (Client erkennt, dass Server nicht abgestürzt)

Unterstützung von **Nebeneffekten**= Mechanismus, über den der Client und der Server unter Verwendung eines applikationsspezifischen Protokolls kommunizieren können. (Bsp.Client öffnet Datei auf Videoserver- muß isochronen Übertragungsmodus einrichten)

RPC2 erlaubt Client, separate Verbindung für Übertragung von Viedodaten einzurichten. Einrichtung der Verbindung als Nebeneffekt eines RPC-Aufrufs an Server. Automatischer Aufruf der Routinen für Einrichtung und Dateiübertragung.



RPC2 unterstützt Multicasting=Parallele RPCs mithilfe des **Multi-RPC-Systems**.
-transparent für den Aufgerufenen und Aufrufer (nicht unterscheidbar von normalen RPCs)
-Aufrufer ruft mehrere Ein-Wege-RPCs parallel auf und blockiert, bis alle Antworten von den nicht ausgefallenen Empfängern empfangen werden
-Alternativ: Einrichtung einer Multicast-Gruppe, Senden eines RPC an alle Mitglieder mit IP-Multicast
-Einsatz: Nach Änderung einer Datei: Ungültigmachen der anderen Client-Kopien:



Prozesse

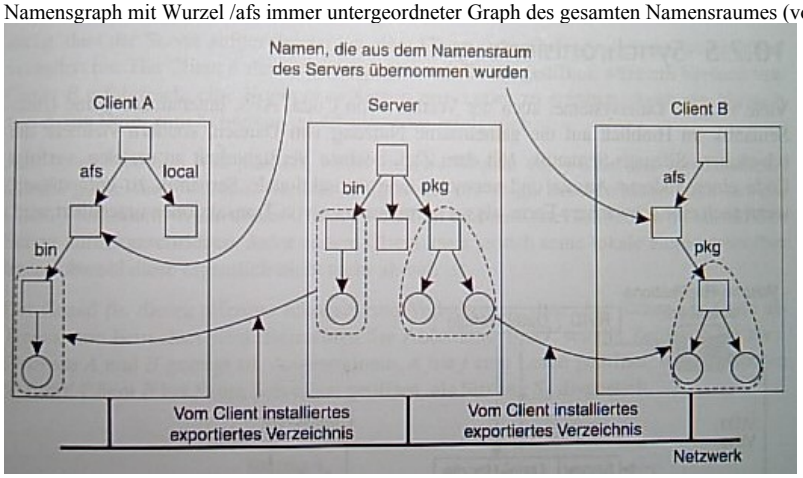
Clients: Venusprozesse
Server: Vice-Prozesse
Beide intern als Menge nebenläufiger Threads
Separater Thread für IIO-Ops-> Vorteil: Blockiert nicht

Namesgebung

Dateiengruppierung in Einheiten=**Volumes** (Menge von Dateien, die einem Benutzer zugeordnet sind)
Volumes

- bilden grundlegende Einheit, mit der der gesamte Namensraum aufgebaut wird
- bilden Einheit für die serverseitige Replikation

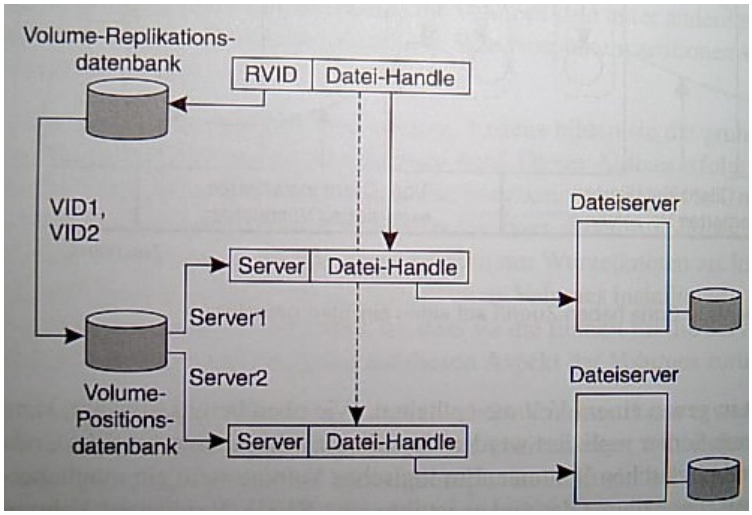
Namensraum –Aufbau durch Installation von Volumes an Installationspunkten
Installationspunkt in Coda=Blattknoten eines Volumes, der sich auf den Wurzelknoten eines anderen Volumes bezieht.
Client kann nur die Wurzel eines Volumes installieren.
Namensgraph mit Wurzel /afs immer untergeordneter Graph des gesamten Namensraumes (von Vice-Servern verwaltet):



Datei-IDs
Unterscheidung:

1. Logische Volumes= evtl repliziertes physisches Volume, hat **RVID** (Replicated Volume ID)=positions- und replikationsunabhängige Volume-ID. Einer RVID können mehrere Repliken zugeordnet werden.
2. Physische Volumes: hat **VID** (Volume-ID), identifiziertspez. Replik auf positionsunabhängige Weise.

Datei-ID besteht aus 2 Teilen: 32 Byte RVID | 64 Bit Datei-Handle (identifiziert Datei innerhalb eines Volumes eindeutig)
Datei-Suche: Client übergibt RVID einer Datei an Volume-Replikationsdatenbank->diese gibt Liste der dieser RVID-zugeordneten VIDs zurück-> damit kann Client Server suchen mithilfe der Volume-Positionsdatenbank->gibt die Position der VID zurück.

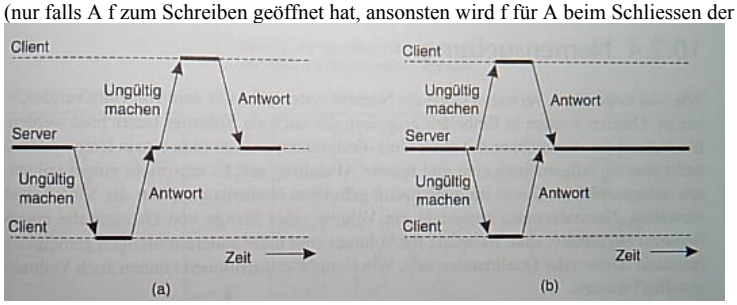


Synchronisierung

Unterstützung transaktionaler Semantik wegen höchster Vergügbarkeit.

Gemeinsam genutzte Dateien in Coda

Client öffnet Datei f->ganze Kopie von f wird auf Client übertragen->Server zeichnet dies auf und blockiert Schreibzugriffe anderer Clients (nur falls A f zum Schreiben geöffnet hat, ansonsten wird f für A beim Schliessen der Datei von B ungültig gemacht):



Transaktionale Semantik

Partition =Teil des Netzwerkes, der vom Rest isoliert ist, besteht aus mehreren Clients/Servern.
Konflikte haben 2 Ops, wenn sie beide mit denselben Daten arbeiten und mind. 1 eine SchreibOp ist.
Codaansatz: Eine Sitzung=1 Transaktion
Unterschiedliche Sitzungstypen werden automatisch von dem von einer Applikation vorgenommenen Systemaufruf abgeleitet.
Vorteil bei Applikationen, die die VFS-Schnittstelle verwenden: Für jeden Sitzungsteil ist im Voraus bekannt, welche Daten gelesen oder verändert werden.
zB Sitzungstyp store:zugeordnet ist eine Liste mit Metadaten, was verändert werden darf (zB Dateilänge,-inhalt, aber nicht Datei-ID).
Damit weiß ein Venus-Prozess, welche Daten zu Beginn vom Server geladen werden müssen und kann Sperren anfordern.
Konfliktlösung über mehrere Partitionen über **Versionsnummern**
Startet Client Sitzung, werden alle relevanten Daten +Versionsnummer übertragen.
Erfolgt Netzwerkpartitionierung (=Client und Server werden getrennt), kann Client fortgesetzt werden. Besteht Verbindung wieder, wird f probeweise an Server übertragen-Server akzeptiert diese aber nur, wenn diese Aktualisierung zur nächsten Version von f führen würde.
Ansonsten wird Aktualisierung rückgängig gemacht, Client muss seine Version speichern und manuell abgleichen.

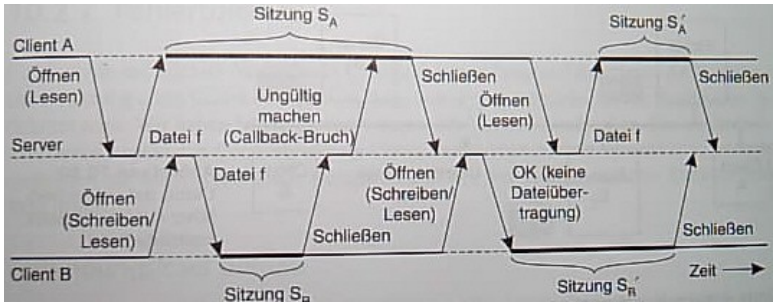
Caching und Replikation

Client-Caching

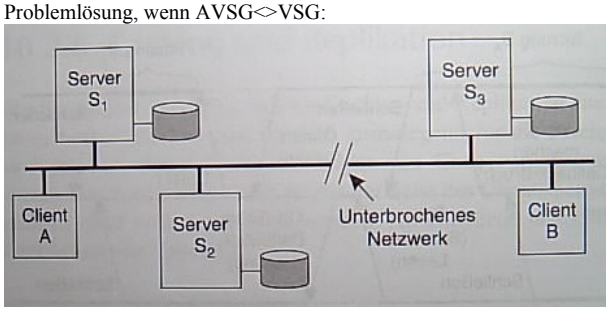
Wichtig aus 2 Gründen:

1. für Skalierbarkeit
2. Fehlertoleranz

=>darum werden immer ganze Dateien in Cache gestellt!
Server zeichnet **Callback-Versprechen** für Client auf: Aktualisiert ein Client seine Kopie zum 1.Mal, benachrichtigt er den Server ->dieser sendet **Callback-Bruch** =Ungültigkeitsnachricht an die anderen Clients.
Wenn Client eine Datei öffnet, die sich in seinem Cache befindet, muss er überprüfen, ob das Callback-Versprechen noch gilt:



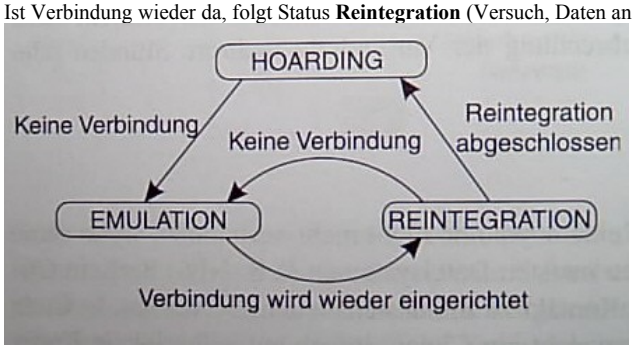
VSG (Volume Storage Group)= Menge der Server, die eine Kopie des Volumens besitzen.
AVSG (Accessible VSG)=kontaktierbares VSG
Konsistenzwahrung eines replizierten Volumes mit **ROWA (Read one, Write all)**: Client liest nur von einem Server der AVSG- schliesst er die Datei, wird sie parallel mittels MultiRPC zu allen Mitgliedern der AVSG übertragen.
Problemlösung, wenn AVSG<>VSG:



Server S_i in einer VSG verwaltet **Coda-Versionsvektor** CVV_i(f)- Gilt CVV_i(f)[j]=k, weiß Server S_i, dass S_j mindestens Version k der Datei f gesehen hat.
CVV_i(f) anfangs [1,1,1] für 3 Server. Später ist zB CVV_1(f)= CVV_2(f)=[2,2,1], während CVV_3(f)=[1,1,2].
Wird die Partitionierung wieder korrigiert, vergleichen die Server ihre Versionen->Konflikt !
Konfliktlösung auf applikationsabhängige Weise bzw Benutzer muss mit einbezogen werden.

Fehlertoleranz

In Coda kann Client weiterarbeiten, auch ohne Verbindung!
Verbindungsloser Betrieb
NFS: Client kann nicht witerarbeiten
Coda: Client nimmt lokale Kopie der Datei.
Beobachtung: Selten schreiben 2 Clients dieselbe Datei.
Client füllt vorsorglich Cache mit Dateien=**Hoarding.**
Ist AVSG=0, geht Client in Status **Emulation.**
Ist Verbindung wieder da, folgt Status **Reintegration** (Versuch, Daten an Server zu schreiben und permanent zu machen).



Für jede Workstation verwaltet Coda eine **Hoard-Datenbank**.Client kann dort Pfadnamen wichtiger Dateien eintragen- diese haben Prioritäten, benutzerdefiniert kombiniert mit Zugriffsdaten . Dateien werden dann nach dieser Priorität in Cache geladen.
Cache-Ausgleich wird alle 10min neu berechnet=**Hoard Walk**
Wiederherstellbarer virtueller Speicher-RVM
RVM=Mechanismus auf Benutzerebene, mit der wichtige Datenstrukturen im Hauptspeicher gehalten werden und sichergestellt wird, dass sie nach Absturz wiederhergestellt werden können.

=>Abbildung als Bytefolge im Hauptspeicher, alle Ops werden in separatem Write-ahead-log in stabilem Speicher aufgezeichnet.
Sinnvoll, batteriegepufferten Teil des RAM zu verwenden.

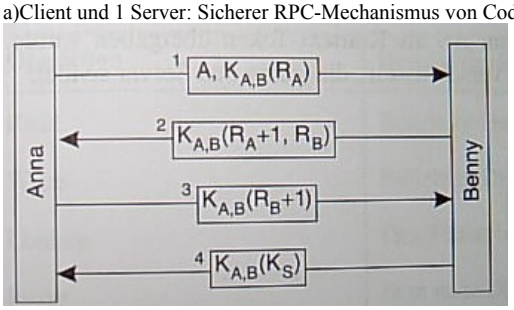
Sicherheit

2 Teile:
1. Einrichtung sicherer Kanäle

2. Zugriffskontrolle auf Dateien

Sichere Kanäle

a)Client und 1 Server: Sicherer RPC-Mechanismus von Coda:



b)Client und mehrere Server: Client erhält Authentifizierungs-Token von AS

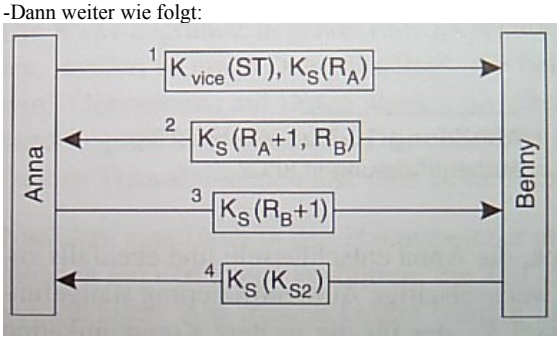
Authentifizierungstoken=ähnlich Kerberos-Ticket. Enthält ID des Prozesses, der es erhalten hat, Token-ID, Sitzungsschlüssel, Zeitstempel mit Gültigkeitsbeginn und –ablauf.

Anmeldung A:

-Wechselseitige Authentifizierung mit AS über Passwort

-AS gibt 2 Tokens zurück: 1. Klartexttoken, das A identifiziert, 2. Geheimes Token=krypt. Versiegelung des Klartexttokens mit geheimem Schlüssel K_vice, der von allen Vice-Servern gemeinsam genutzt wird.

-Dann weiter wie folgt:



Besteht keine Verbindung bei Authentifizierung, wird diese auf später verschoben, vor Reintegrations-Status.

Zugriffskontrolle

Wie AFS: Zugriffskontrolllisten.

Zugriffskontrolllisten werden nur Verzeichnissen zugeordnet (einfach, skalierbar)

Keine Berechtigung für Ausführung, da Vice-Server nicht erkennen kann, ob Client die heruntergeladene Datei ausführt oder nicht.

Auch negative Berechtigungen möglich, gegen böswillige Benutzer.

Vergleich verteilter Dateisyssteme

	NFS	Coda
Philosophie	Versucht, System bereitzustellen, das Clients transparenten Zugriff auf ein Dateisystem ermöglicht, das auf einem entfernten Server gespeichert ist. Erinnert an objektbasierte Systeme. NFS4: Verbergen von Latenzzeiten durch Cachen der ganzen Datei=lokale Ausführung	Ziel: höchste Verfügbarkeit (Netzwerkprobleme) Erbt Entwurfsziele von AFS, am wichtigsten Erzielung von Skalierbarkeit
Zugriffsmodell	Entfernt	Up/Download
Kommunikation	RPC	
Clientprozesse	NFS3: Schlank, Ops werden auf Server ausgeführt, NFS4: Fett, durch Caching ganzer Dateien auch lokale Ausführung von Ops	Venus-Prozess erledigt viel Arbeit: Caching ganzer Dateien, emuliert Server-Funktionen
Server-Prozesse	Datei auf einem einzigen, nicht repliziertem Server	Server in Gruppen angeordnet, jede Gruppe kann replizierte Datei beherbergen
Namensgebung	Jeder Benutzer erhält eigenen, privaten Namensraum, teilweise Standardisierung wegen Schwierigkeit, ansonsten Dateien abhängig von Namen gemeinsam zu nutzen	Gemeinsam genutzter Namensraum, Ergänzung mit privatem, lokalen Namensraum möglich
Datei-ID-Gültigkeitsbereich	Nicht systemübergreifend, eindeutig auf Dateiserver	Global durch (VID,Handle)
Synchronisierung	Sitzungssemantik: nur Aktualisierung des letzten schliessenden Prozesses wird vom Server berücksichtigt	Transaktionale Semantik: nur Sitzungen erlaubt, die serialisiert werden können.
Caching und Replikation	Clientseitiges Caching, minimale Unterstützung für Replikaiton durch Attribut FS_LOCATIONS	ROWA-Protokoll für Konsistenzsicherung
Fehlertoleranz	Zuverlässige Kommunikation	Replikation und Caching für höchste Verfügbarkeit.
Sicherheit	Sichere Kanäle durch RPCSEC_GSS. Zugriffskontrolle durch Liste mit Ops	Kanäle basierend auf Needham-Schroeder: Zugriffskontrolle durch Verzeichnis-Op

KE6

5.5

5.5.1 Zentralisierter Algorithmus

Vorgehensweise wie Einprozessorsystem: Ein Prozess ist Koordinator, von diesem müssen Prozesse Berechtigungen für den kritischen Bereich anfordern.
Koordinator akzeptiert: sendet OK-Antwort
Koordinator verweigert: sendet keine Antwort, stellt Anforderung in Warteschlange.
Nachteil: Prozess kann nicht zwischen Verweigerung und Absturz des Koord. unterscheiden

5.5.2 Ein verteilter Algorithmus

Algo von Ricart und Agrawala: Sortierung nötig, zB durch Zeitstempel.
Prozess will in kritischen Bereich eintreten->sendet Nachricht an alle Prozesse einschließlich sich selbst (zuverlässige Kommunikation nötig), 3 Fälle:
1. Empfänger will nicht eintreten->sendet OK
2. Empfänger ist bereits in kritischem Bereich->antwortet nicht, stellt Anforderung in Warteschlange.
3. Empfänger will auch in kritischen Bereich-> vergleicht Zeitstempel mit seiner Anforderung: Eingehende niedriger->sendet OK, Eigene niedriger->Anforderung in Warteschlange, keine Antwort.
Prozess darf erst eintreten, wenn er OKs von allen Empfängern hat!
Nachteil: Bei Absturz eines Prozesses auch keine Antwort, Lösung, indem immer OK oder Verweigerung gesendet wird.

5.5.3 Ein Token-Ring-Algorithmus

In Bus-Netzwerk wird softwareseitig ein Ring aufgebaut, jeder Prozess muß nächsten kennen.
Prozess 0 erhält das Token, dieses kreist, bis ein Prozess in kritischen Bereich eintreten will (kein Eintritt in 2.kritischen Bereich für dassselbe Token!).
Kein Starvation (Verhungern) möglich.
Probleme:
Verlust des Tokens-nicht einfach festzustellen, da es auch lange Nutzung bedeuten kann.
Prozessabsturz-Prozess muss Empfang bestätigen, kommt keine Antwort, wird Token an übernächste Position weitergereicht.

5.5.4 Vergleich der 3 Algos

Algorithmus	Nachrichten pro Ein-/Austritt	Verzögerung vor dem Eintritt	Probleme
Zentralisiert	3	2	Koordinator-absturz
Verteilt	2(n-1)	2(n-1)	Absturz eines belieb. Prozesses
Token Ring	1 bis unendlich	0 bis n-1	Verlorrenes Token, Prozessabsturz

5.6 Verteilte Transaktionen

5.6.1 Das Transaktionsmodell

Transaktionen führen eine Reihe von Ops nach dem Alles-oder-Nichts-Prinzip aus.
4 Eigenschaften (ACID):

1. **Atomar**: Sie wird entweder ganz oder gar nicht ausgeführt, andere Prozesse sehen Zwischenzustände nicht
2. **Konsistent**: Invarianten müssen vor und nach der Transaktion eingehalten werden, dazwischen können sie verletzt werden
3. **Isoliert** (serialisiert): Gleichzeitige Ausführung von mehreren Transaktionen erscheint anderen Prozessen wie sequentielle Ausführung.
4. **Dauerhaft**: Ergebnisse werden permanent, wenn Transaktion festgeschrieben

5.6.2 Klassifizierung von Transaktionen

Einige Einschränkungen flacher Transaktionen

Flache Transaktionen genügen ACID-Eigenschaften.
Nachteil von atomar: Partielle Ergebnisse werden nicht festgeschrieben, zB Buchen von 3 Flügen, ist einer nicht verfügbar werden die anderen auch nicht reserviert.

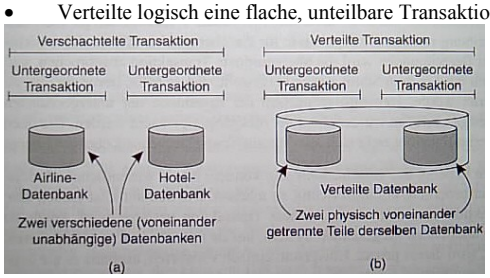
Verschachtelte Transaktionen

Sind aus mehreren untergeordneten Transaktionen aufgebaut, diese können auf unterschiedlichen Maschinen ausgeführt werden.
Problem: Werden untergeordnete Transaktionen festgeschrieben und die übergeordnete Transaktion abgebrochen, müssen diese wieder rückgängig gemacht werden->administrativer Aufwand.
Jede Transaktion erhält private Kopie aller Daten und verwirft/schreibt diese fest.

Verteilte Transaktionen

arbeiten mit Daten, die über mehrere Maschinen verteilt sind.
Problem: Man braucht separat verteilte Algorithmen zum Sperren der Daten und Festschreiben der Transaktionen.
Unterschied zu verschachtelter Transaktion:

- Verschachtelte ist logisch in eine Hierarchie von Transaktionen zerlegt,
- Verteilte logisch eine flache, unteilbare Transaktion, die auf verteilten Daten arbeitet.



5.6.3 Implementierung

Privater Arbeitsbereich

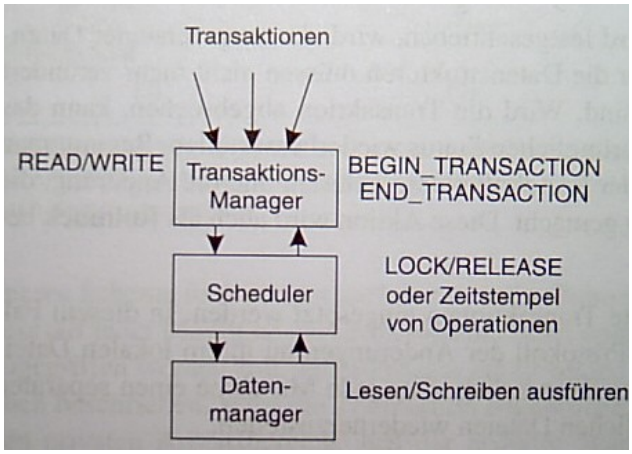
Lesen: Prozess braucht Daten nicht kopieren, Zeiger auf den Arbeitsbereich des übergeordneten Prozesses langt.
Schreiben: Muß kopiert werden, besser aber: Nur Index der Datei (Block auf der Festplatte) wird kopiert. Modifizieren: der jeweilige Block wird kopiert, alle anderen belassen, bei Festschreiben wird aktualisiert. Neue Blöcke heißen **Schatten-Blöcke.**
Abb S 318

Writeahead-Protokoll

Dateien werden tatsächlich verändert, aber vor dem Schreiben wird der Datensatz in ein Protokoll geschrieben- wird die Transaktion abgebrochen, werden von hinten her alle Änderungen rückgängig gemacht=**Rollback**

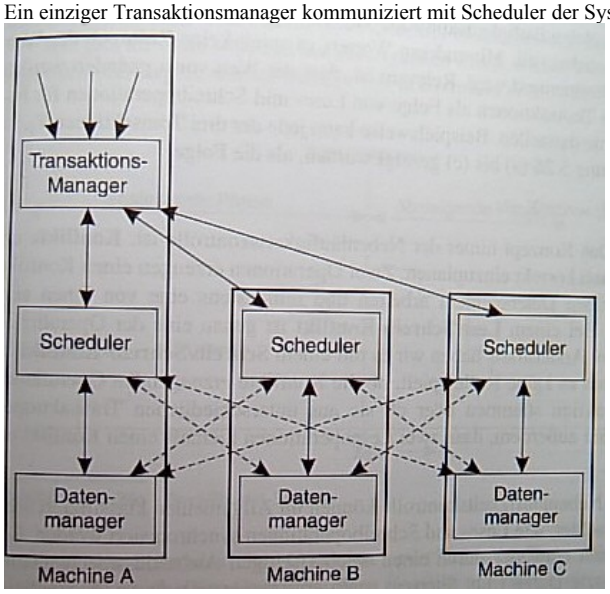
5.6.4 Nebenläufigkeitssteuerung

Ziel der Nebenläufigkeitskontrolle: Ausführung mehrerer Transaktionen derart, dass Reihenfolge der Zugriffe im Ergebnis genauso wie sequentielle Ausführung ist.
3 geschichtete Manager:



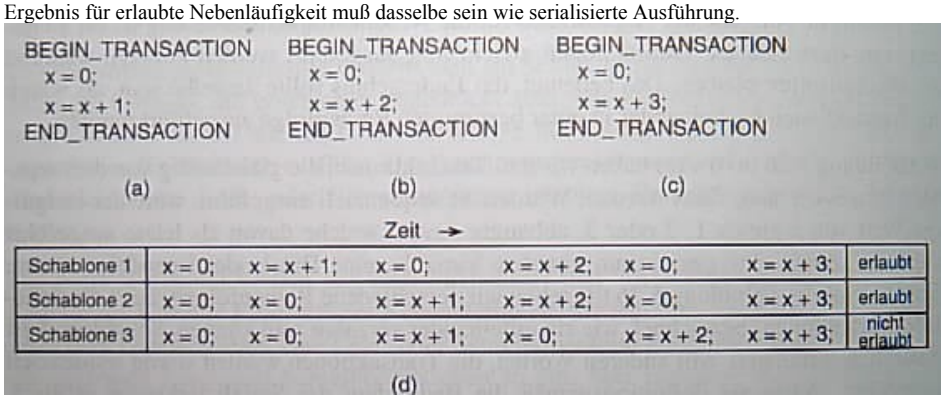
- Datenmanager liest und schreibt Dateien, weiß nichts über Transaktionen
- Scheduler übernimmt Hauptverantwortung für korrekte Kontrolle der Nebenläufigkeit. Legt fest wann welche Transaktione dem Datenmanager eine Lese- oder Schreibop übergeben darf.
- Transaktionsmanager verantwortlich für Atomarität von Transaktionen.

Ein einziger Transaktionsmanager kommuniziert mit Scheduler der Systeme:



Serialisierbarkeit

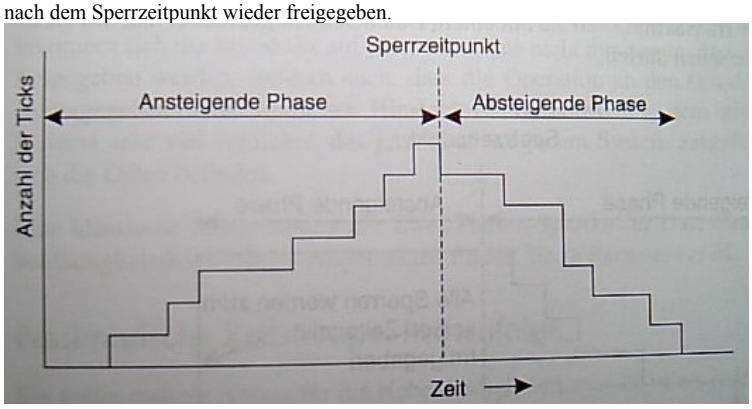
Verschiedene Schablonen=schedules für die Reihenfolge der nebenläufigen Transaktionen.
Ergebnis für erlaubte Nebenläufigkeit muß dasselbe sein wie serialisierte Ausführung.



Konzept hinter Nebenläufigkeit ist, Konflikte erzeugende Operationen korrekt einzuplanen (2 Ops erzeugen Konflikt, wenn mindestens eine von ihnen eine Schreibop ist- Lese/Schreib- und Schreib-Schreib-Konflikt.
Algorithmenklassifizierung für Nebenläufigkeitskontrolle:

Zwei-Phasen-Sperren

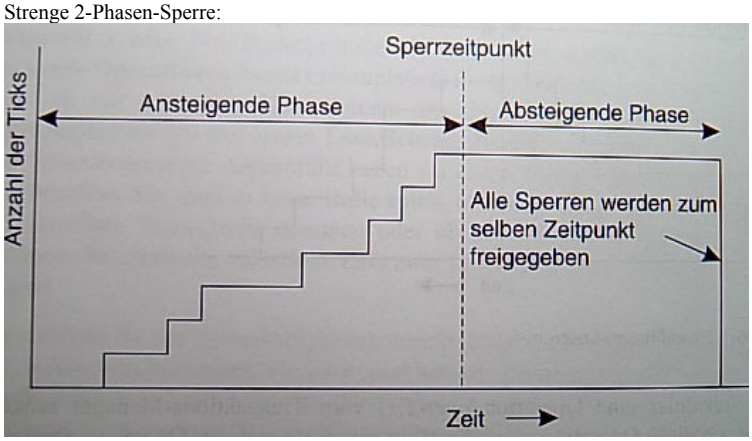
Aufgabe des Schedulers: Sperren derart erteilen/freigeben, dass nur gültige Schablonen entstehen. Zuerst werden alle Sperren angefordert, nach dem Sperrzeitpunkt wieder freigegeben.



Regeln:

1. Erzeugt Op1 Konflikt mit anderer Op, für die bereits Sperre erteilt ist: Op1 (und damit Transaktion 1) wird verzögert; ansonsten Op1 weiterreichen an Datenmanager
2. Datenelement x wird erst freigegeben, wenn Datenmanager bestätigt, dass Op durchgeführt, für das x gesperrt wurde.
3. Nach Freigabe einer Sperre für Transaktion T erteilt Scheduler keine neue Sperre mehr für T

Dadurch sind alle Schablonen, die durch eine Verzahnung gebildet werden, serialisierbar!
Strenge 2-Phasen-Sperre:



Vorteile:

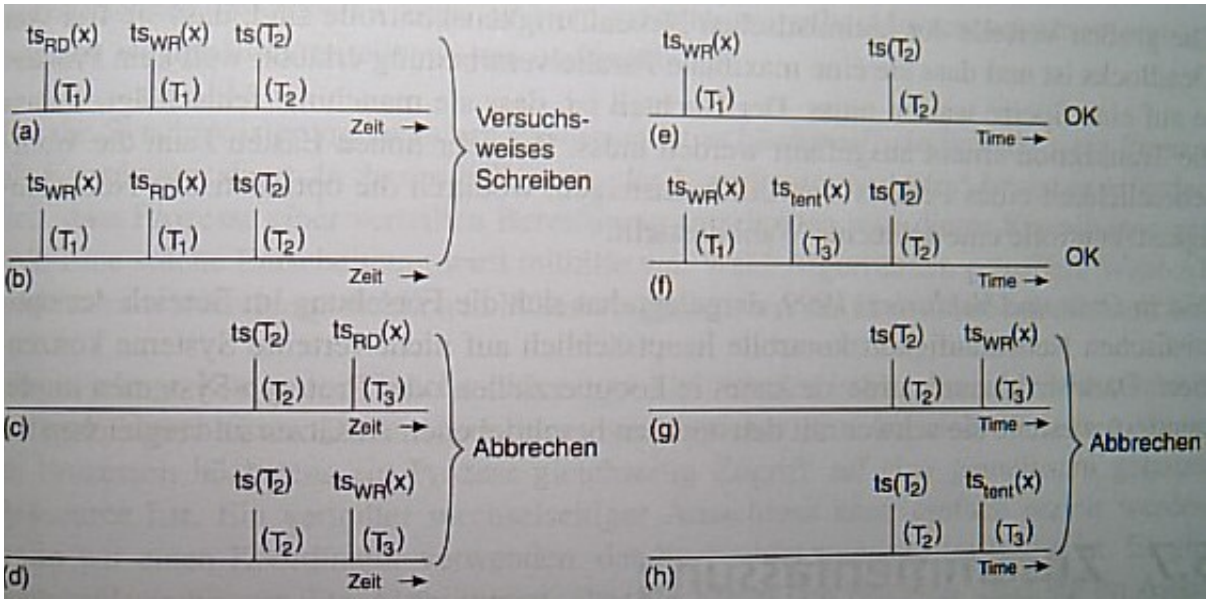
1. Transaktion liest immer einen Wert, der von einer festgeschriebenen Transaktion geschrieben wurde-> Transaktionen müssen nie abgebrochen werden
2. Sperranforderungen/Freigaben werden vom System verarbeitet, unbemerkt von der Transaktion.

=>kaskadenförmige Abbrüche fallen weg (Rückgängigmachen einer festgeschriebenen Transaktion).
Deadlocks sind aber möglich!
Replizierte Datenelemente: Kommunikation mit zentralem Sperrmanager

- Primäres 2PL: Jedem Datenelement wird eine primäre Kopie zugeordnet, der Sperrmanager auf der Maschine dieser primären Kopie ist für sperren/freigeben verantwortlich.
- Verteiltes 2PL: Scheduler auf jeder Maschine kümmert sich ums Sperren/Freigeben. Und um Weitergabe der Ops an den lokalen Datenmanager.

Pessimistische Zeitstempel-Reihenfolge

Jeder Transaktion wird (eindeutiger) Lamport-Zeitstempel zugewiesen.
Jedes Datenelement hat einen Lesezeitstempel (Wert gleich dem Stempel der letzten lesende Transaktion) und Schreibzeitstempel (Wert gleich dem Stempel der letzten schreibenden Transaktion).
Konflikt: Datenmanager verarbeitet zuerst die Op mit dem kleinsten Zeitstempel.
Lese-/SchreibOp: Ist Zeitstempel der Transaktion kleiner als der Lese-/Schreibzeitstempel des Elements, heißt das, dass nach dem Start von T eine andere Schreib-/LeseOp für das Element ausgeführt wurde=> Abbruch!



Unterschied zu Sperre: Transaktion bricht ab, Sperre lässt warten.
-> Frei von Deadlocks!

Optimistische Zeitstempel-Reihenfolge

Beobachtet gelesene und geschriebene Datenelemente.
Beim Festschreiben werden alle anderen Transaktionen auf geänderte Elemente geprüft, wenn ja, wird die Transaktion abgebrochen.
Am besten geeignet für Implementierung auf privaten Arbeitsbereichen.
Frei von Deadlocks, maximale Parallelverarbeitung, aber auch Gefahr von erhöhten Abbrüchen, je höher die Lasten.