Verteilte Dateisysteme

Dateien sind auch in verteilten Systemen der Ablageort von Informationen. In diesem Teil wird auf die besonderen Probleme bei der Speicherung von Daten in einem verteilten System eingegangen.

1.Grundlagen....4

2.Architektur verteilter Dateisysteme....7

  2.1.Sichtweisen auf verteilte Dateisysteme....9

3.Semantik des Dateisharing....13

  3.1.Einzelkopie Semantik....14

  3.2.Sitzungssemantik....16

  3.3.Unveränderlichkeits-Semantik....17

  3.4.Transaktions-Semantik....18

4.Implementierungsaspekte....19

  4.1.Architekturtypen....20

4.1.1.Zugriffsmodelle........21
4.1.2.Zustandslose und zustandsbehaftete Dateiserver........24
4.2.Verzeichnisdienst........26
4.3.Caching und Konsistenz........30
4.3.1.Algorithmen zur Cachekohärenz........34
4.4.Replikation........36
5.Network File System 41
5.1.NFS Architektur........41
5.2.Remote Mounting........44
5.3.NFS Protokoll........45
5.4.Implementierungsaspekte........46
5.5.Bewertung........48
5.6.Beispiel: Unix-Labor........50
5.7.Arbeitsweise von NFS........51
5.8.Ein NFS-Volume mounten........53
5.9.Die NFS-Dämonen........56

5.10.Die exports-Datei..56
5.11.Der Linux-Auto-Mounter...58

1. Grundlagen

Dateien sind Abstraktionsform für die dauerhafte Speicherung von Daten, Informationen und Programmen. Sie bestehen meist aus einer linearen Aneinanderreihung von Bytes. Für den Zugriff verwendet man einen Dateizeiger, der zu einem Zeitpunkt auf ein bestimmtes Byte verweist.

Ein **Dateisystem** ist auf **Einzelplatzrechnern** die Schnittstelle zur Festplatte.

Ein verteiltes Dateisystem ermöglicht es zusätzlich:

- entfernte Dateien zuzugreifen
- plattenlose Geräte zu unterstützen.

Verteilte Dateisysteme sind als **Client/Server**-System aufgebaut:

- das Dateisystem bietet einem Klienten einen Dateidienst (engl.: file service)
- und wird von einem oder mehreren Dateiservern (engl.: file server) implementiert.

Die **Anforderungen** an ein verteiltes Dateisystem lassen sich wie folgt zusammenfassen:

- Zugriffstransparenz
  Der Zugriff auf entfernte Dateien erfolgt für Klienten mit den **gleichen Operationen**, wie der Zugriff auf lokale Dateien.

- Ortstransparenz
  Es spielt keine Rolle, an welchem **physikalischen Ort** eine Datei gespeichert ist.

- Nebenläufigkeitstransparenz
  Greifen mehrere **Klienten gleichzeitig** auf eine Datei zu, soll kein Klient von den Zugriffen der anderen etwas mitbekommen.
  Aus Konsistenzgründen sind nebenläufige, schreibende Zugriffe problematisch. Oft wird dies durch einfache Sperrverfahren verhindert, wodurch dann bei überlappenden Zugriffen keine Nebenläufigkeitstransparenz mehr gilt.

- Fehlertransparenz
  Für Klienten, wie für Server, soll ein Fehlverhalten des jeweils anderen nicht zu Inkonsistenzen und Verklemmungen führen.

- Lasttransparenz
  Die **Anfragelast** auf Serverseite soll sich nicht auf dessen **Antwortzeiten** auswirken.

- Hardware- und Betriebssystem-Transparenz
  Das verteilte Dateisystem soll die **Heterogenität** unterschiedlicher Hardware und Betriebssysteme verbergen.

Die Programmierschnittstelle des Dateisystems muss dazu vereinheitlicht über verschiedenen Betriebsystemen implementiert sein.

- Replikationstransparenz
  Der Klient darf immer **nur eine logische Datei** sehen, egal wie häufig und wo diese repliziert vorgehalten wird.

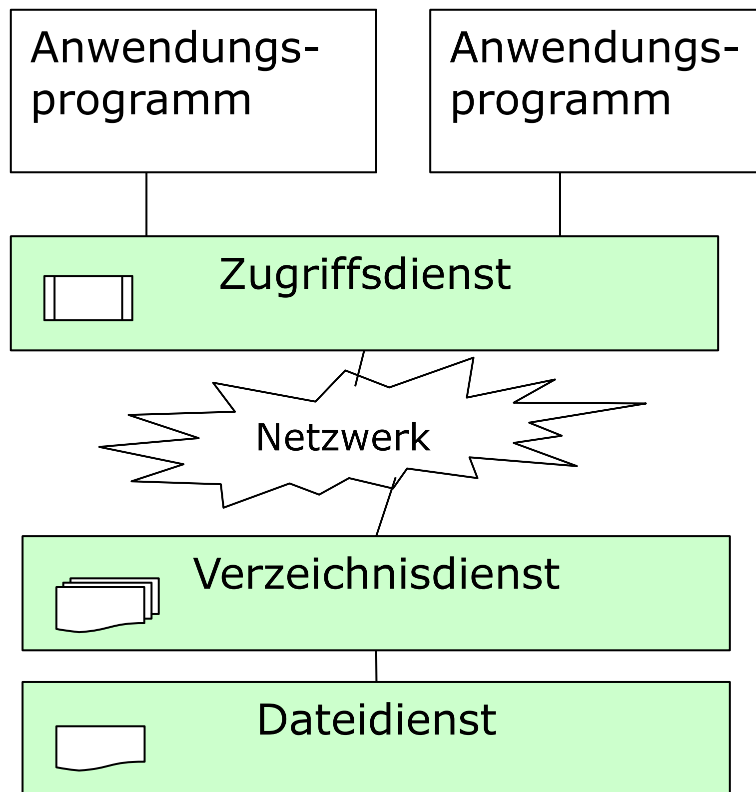
- Migrationstransparenz
  Verändert eine Datei ihren **Speicherungsort**, dann ist dies für den Klienten nicht sichtbar.
  Dateien werden eventuell dynamisch „in der Nähe" von den Klienten gehalten, die häufig darauf zugreifen.

- Partitionierungstransparenz
  Das Dateisystem kann **partitioniert** werden, d.h. bestimmte **Dateiserver** sind nicht mehr zugreifbar für bestimmte Klienten. Trotzdem stehen für alle Klienten noch alle benötigten Dateien zur Verfügung.
  Eine Vereinigung bisher getrennter Partitionen erzeugt dabei möglichst keine Inkonsistenzen.
  Partitionierungstransparenz ist speziell für die Anbindung mobiler Rechner interessant.

Nicht alle diese Anforderungen werden jedoch von den verfügbaren Systemen erfüllt.

2. Architektur verteilter Dateisysteme

Ein verteiltes Dateisystem besteht konzeptionell aus drei **Komponenten**, so wie in der folgenden Abbildung verdeutlicht:



Der **Dateidienst**

- regelt die die **elementaren Operationen** auf Dateien und deren Inhalt.
  (Lesen, Schreiben, Zeitstempel, Allokierung von Plattenblöcken, Plattenein- und ausgabe und Pufferung)

Der **Verzeichnisdienst** ist zuständig

- für die Abbildung von **Dateinamen** auf die binären Dateizeiger,
- das **Verwalten** der Dateien im meist hierarchischen Namensraum,
- den Test und das Ändern der **Zugriffsrechte**,
- die Unterstützung und Umrechnung von symbolischen **Verweisen**.

Der **Zugriffsdienst**

- ist für das **Anwendungsprogramm** die **Schnittstelle**, über die Aufrufe an den Verzeichnis- und Dateidienst erfolgen.
- Die lokalen Aufrufe des lokalen Betriebs- oder Dateisystems werden darauf abgebildet.

- Es kann in unterschiedlich komplexen Ausprägungen vorliegen:
  vom einfachen RPC-Stub, bis zu einem eigenen Dateisystem mit Cacheeinträgen aus dem verteilten Dateisystem.

2.1. Sichtweisen auf verteilte Dateisysteme

Die **Sichtweise** auf verteilte Dateien wird u.a. bestimmt durch die Art des Zugriffs auf Rechner und lokale Daten, d.h. die **Syntax** der **Dateinamen** des verteilten Dateisystems:

1. **Rechnername + Pfadname** bzw. Servername + Pfadname

   - Jeder Rechner oder Dateiserver hat seine eigene, von anderen Rechnern unabhängige Verzeichnisstruktur.

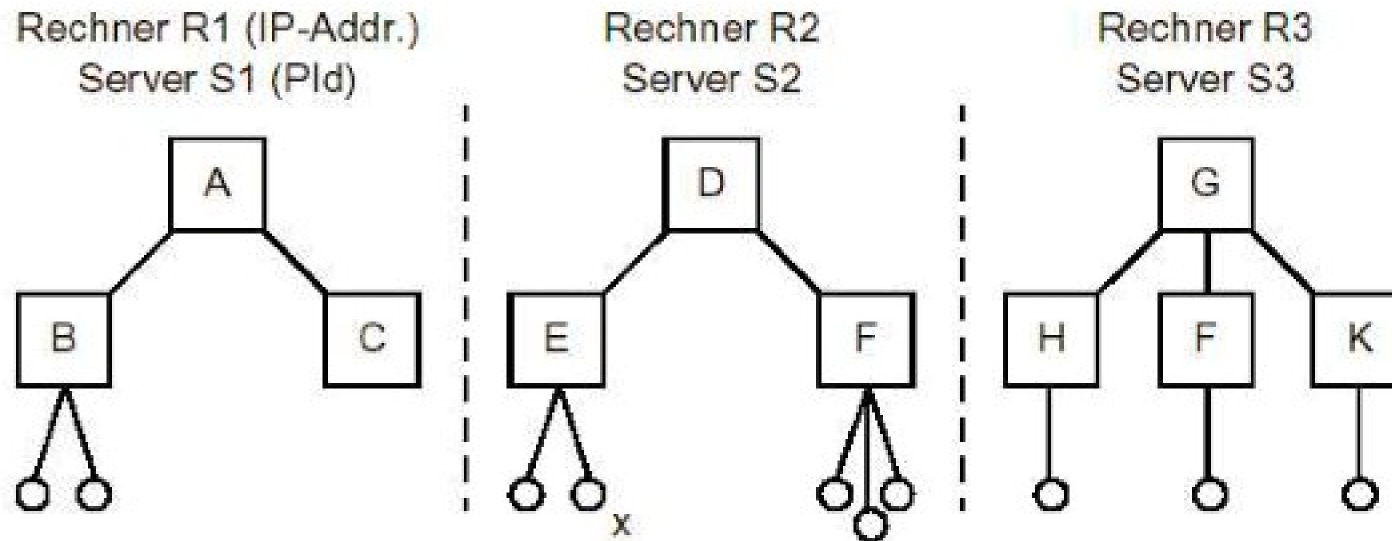
   - ein Dateiname wird durch Konkatenation aus Rechner- oder Servernamen und dem Pfad im Dateiverzeichnis gebildet.

   - Die **Namengebung** ist dadurch **orts**- oder **serverabhängig**.

   - Rechnernamen sind beispielsweise die IP-Adresse

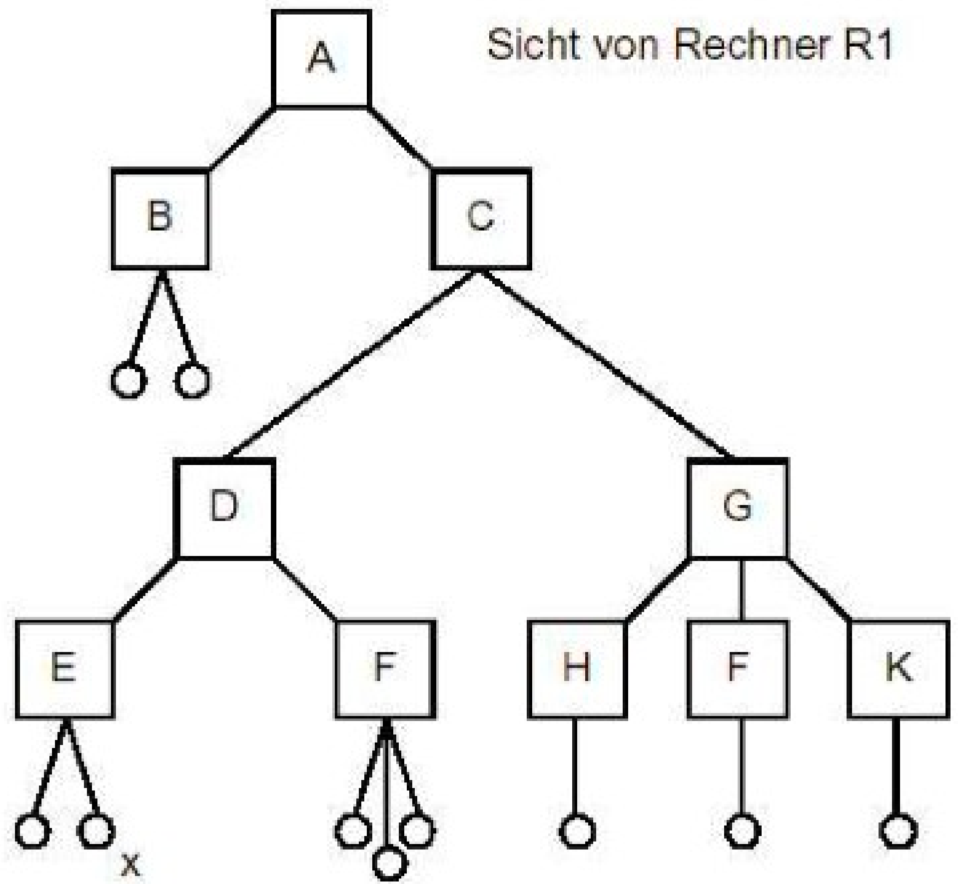
   - Servernamen werden z.B. gebildet aus Portnummer und IP-Adresse.

Beispiel (Autonomer Dateiserver):



2. Remote Mounting

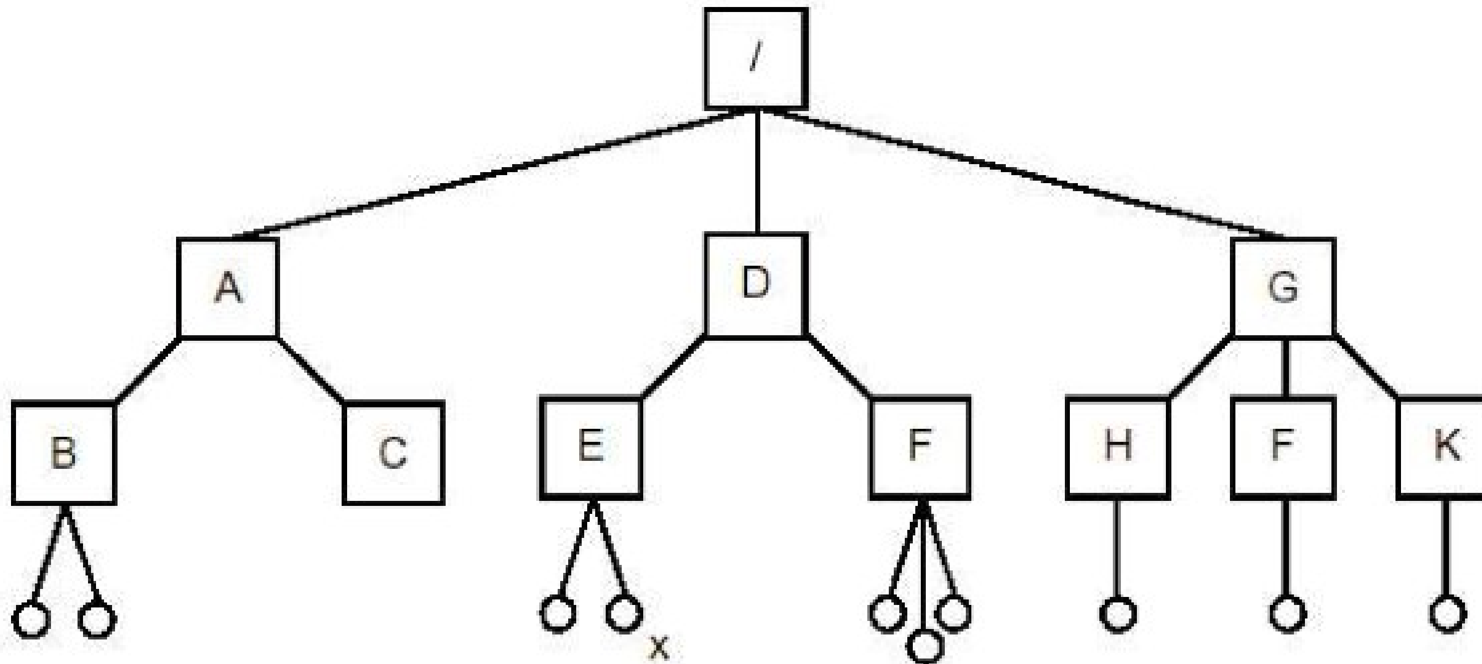
- Eine Verzeichnisstruktur eines entfernten Rechners wird logisch in die lokale Struktur eingebunden.
- Da diese Einbindung für jeden Rechner individuell erfolgen kann, ist **keine Namenstransparenz** gegeben.



Datei x ist also aus Sicht R1 bestimmt durch /A/C/D/E/x

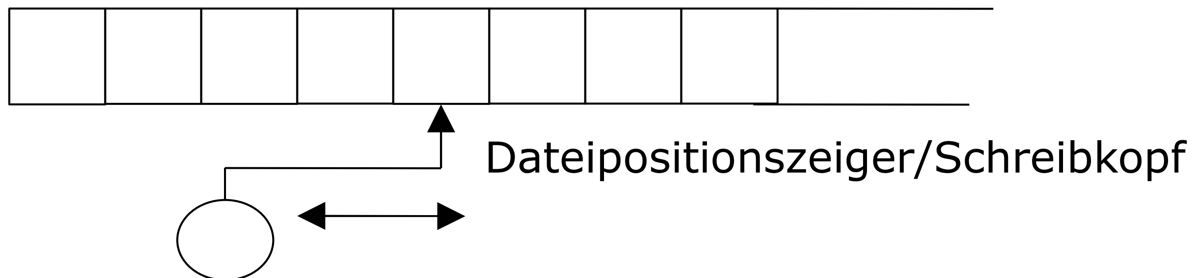
3. Uniformer Namensraum durch **Superroot**

- Über alle lokalen Dateisysteme wird eine übergeordnete Wurzel gesetzt.
- Dadurch erreicht man sowohl eine **Orts**-, als auch **Namenstransparenz**.



3. Semantik des Dateisharing

Greifen mehrere **Prozesse** auf eine **Datei** gemeinsam zu, entstehen **Nebenläufigkeitsprobleme**.



Auf einem **Einzelrechner** ist es durch die zentrale Auslegung des Systems leicht möglich Lese- und **Schreibzugriffe stets sequentiell** geordnet zu halten z.B. durch einen einzelnen Dateipositionszeiger.

Dies kann auf ein verteiltes Dateisystem übertragen werden, wenn auch hier eine zentrale Koordinierung durchgeführt wird.

In den meisten verteilten Dateisystemen wird aber wegen **Performance** mit **Caching** oder **Replikation** gearbeitet, dann funktioniert diese Methode nicht.

Eine (**schlechte**) Lösung der Sharing-Probleme ist es, **generelles Sperren von Dateien** durchzuführen:

> Greift ein Client auf eine Datei zu, so sperrt er sie für den Zugriff durch Andere.

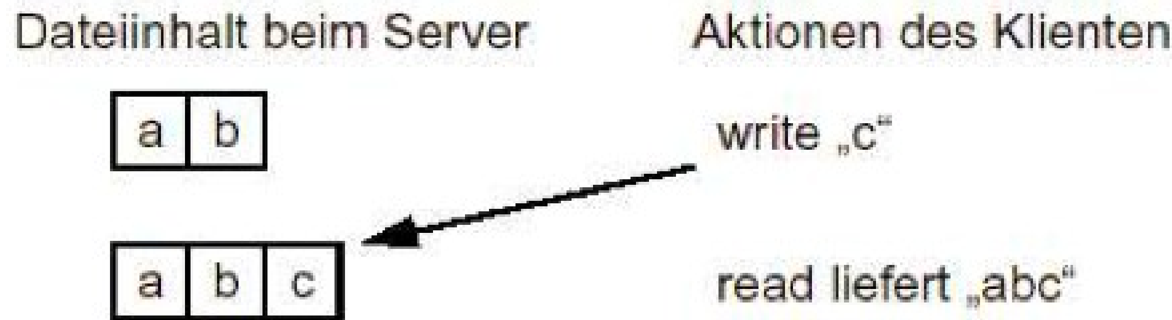
Nachteil: Diese Methode **verhindert** jegliche **Parallelität** des Zugriffs.

Andere Ansätze werden nun diskutiert.

3.1. Einzelkopie Semantik

In Unix wird folgende Methode implementiert, deshalb heißt die auch Unix-Semantik:

> Jeder **Lesezugriff** auf eine Datei erhält immer das **Ergebnis** des **zeitlich davor liegenden letzten Schreibzugriffs**.
>
> Dies gilt auch, wenn ein Client nach dem Öffnen der Datei mehrfach zugreift, also für **jedes** read(...).



Implementiert man **einen zentralen Dateiserver**, ist die Einzelkopie Semantik **leicht** zu **realisieren**.

Wenn die **Klienten** Dateien **cachen**, dann muss das Caching so implementiert werden, dass **Veränderungen** einer Cachekopie **sofort** bei allen anderen Kopien sichtbar werden (write-through).

Um die **Netzunzulänglichkeiten** auszugleichen, wird dazu z.B. eine **virtuelle globale Zeit** benötigt, durch die alle Schreib- und Lesezugriffe in eine kausale Ordnung gebracht werden.

Bewertung:

> Die Einzelkopie-Semantik für verteilte Dateisysteme ist durch ihre zentrale Implementierung bzw. durch den Aufwand zur Konsistenzwahrung (write through) sehr **ineffizient**.

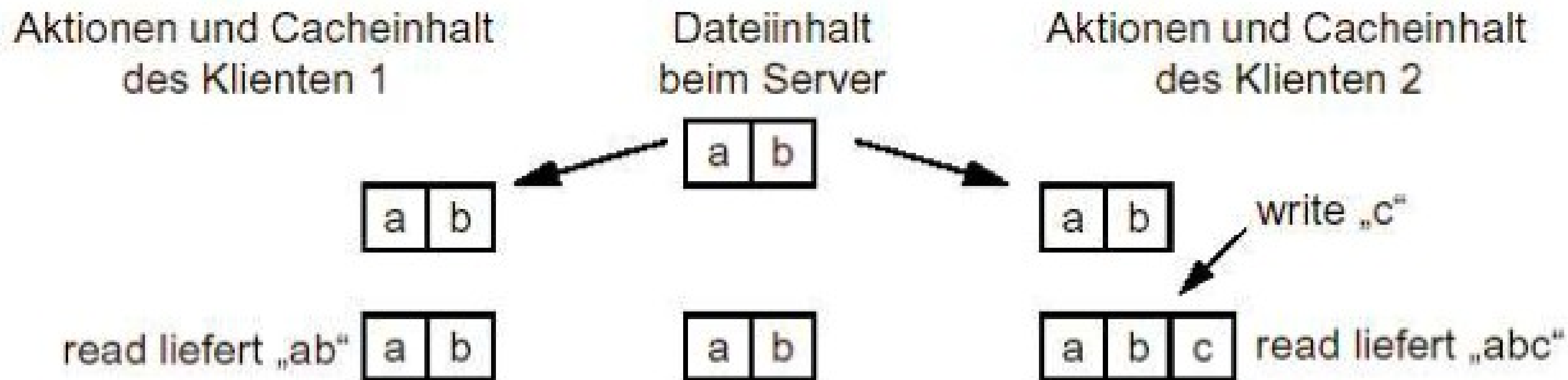
3.2.Sitzungssemantik

Beim **Öffnen** einer Datei erhält der **Klient** eine **eigene Kopie**. Mit dieser Kopie arbeitet er bis zum Schließen der Datei. Dann wird die Datei als Ganzes zum Dateiserver zurück geschrieben.

Dadurch sind **Dateiänderungen** für andere Klienten erst **nach** dem **Schließen** sichtbar.

Probleme treten auf, wenn eine Datei bei mehreren Klienten gleichzeitig geändert wird. Dies kann durch Versionsführung "gemildert" werden.

Das nachfolgende Schaubild verdeutlicht die Situation bei 2 Sitzungen:



Die Idee der **Versionsführung** ist:

- Mit dem letzten Schließen werden alle vorherigen Versionen beim Server überschrieben.
- Die Klienten müssen zwischen den verschiedenen Versionen einen Abstimmungsprozess durchführen.
- Unterschiedliche Versionen werden von dem Server geeignet gemischt.
- Dies kann nur dann sinnvoll erfolgen, wenn der Anwendungskontext bekannt ist (z.B. Gruppeneditoren, vgl. Groove).
- Die Versionen werden getrennt voneinander weitergeführt, etwa durch spezielle Namengebung für die Versionen.

3.3. Unveränderlichkeits-Semantik

Bei diesem Ansatz sind **Dateien nicht veränderbar** (engl.: immutable file semantics).

Dateien können **immer nur neu erstellt** oder gelesen werden. **Änderungen** erzwingen stets ein **Neuanlegen** der Datei.

Greifen mehrere Klienten (lokal) schreibend auf eine gerade mehrfach gelesene Datei zu, entstehen mehrere Versionen. D.h. es ergibt sich eine ähnliche Versionsproblematik wie bei der Sitzungs-Semantik.

3.4.Transaktions-Semantik

Alle **Zugriffe** auf eine Datei werden als **Transaktion** aufgefasst.

Lese- und Schreiboperationen erfolgen in einer Transaktionsklammerung:

```
BeginTransaction

read() bzw write()

EndTransaction
```

Dadurch bleiben die Dateien stets konsistent.

Bewertung:

- Der Verwaltungsaufwand für transaktionsbezogene Dateizugriffe ist sehr hoch und damit das verteilte Dateisystem auch nicht sehr effizient.
- Falls Dateien als Datenbankersatz verwendet werden sollen, ist diese Semantik zwingend.

4. Implementierungsaspekte

Die Nutzerprofile eines verteilten Dateisystems bestimmen die Designentscheidungen.

Ein **verteiltes Dateisystem** für die gemeinsame Nutzung von Dateien in einer **Workstation-Umgebung** etwa ist charakterisiert durch:

- Die Mehrheit aller Dateien ist **klein**
  - Die meisten Dateien sind kleiner als 10 Kilobyte.
  - Die Dateien passen komplett in ein Netzwerk-Paket.
  - D.h. es lohnt sich, die gesamte Datei auf einmal vom Server zum Klienten zu übertragen und nicht Schreib- und Leseanforderungen einzeln über das Netz abzuwickeln.
- Dateien werden **häufiger gelesen** als modifiziert
  - Die Effizienz lesender Zugriffe lässt sich gut mit Caching steigern.
- Viele Dateien haben eine **kurze Lebensdauer**
  - Der Rücktransfer von Dateien im Klientencache zum Server ist für kurzlebige Dateien unnötig, z.B. für temporäre Dateien, die bei einem Compilerlauf erzeugt wurden.

- Datei-**Sharing** ist **selten**
  - Der zusätzliche Overhead für die Konsistenzerhaltung des Caching auf Klientenseite kann zugunsten der kürzeren Antwortzeiten aufgegeben werden.
  - Der durchschnittliche Prozess benutzt wenige (und kleine) Dateien
  - Das Caching kann in den Arbeitsspeicher des Klienten verlagert werden.

Es gibt **Dateiklassen** mit spezifischer Charakteristik, die jeweils **unterschiedliche Mechanismen** zum Verteilen erfordern:

- Dateien, die **ausführbare Programme** speichern, werden nur gelesen, jedoch von vielen Klienten. Sie können also stark **repliziert** werden.

- **Temporäre Dateien** sind sehr kurzlebig, müssen also gar **nicht** im verteilten Dateisystem eingetragen werden.

- Es gibt private Dateien, wie **Mailboxen**, die häufig verändert, aber nicht gemeinsam benutzt werden. Sie können also **nahe beim Klienten** gehalten werden.

4.1. Architekturtypen

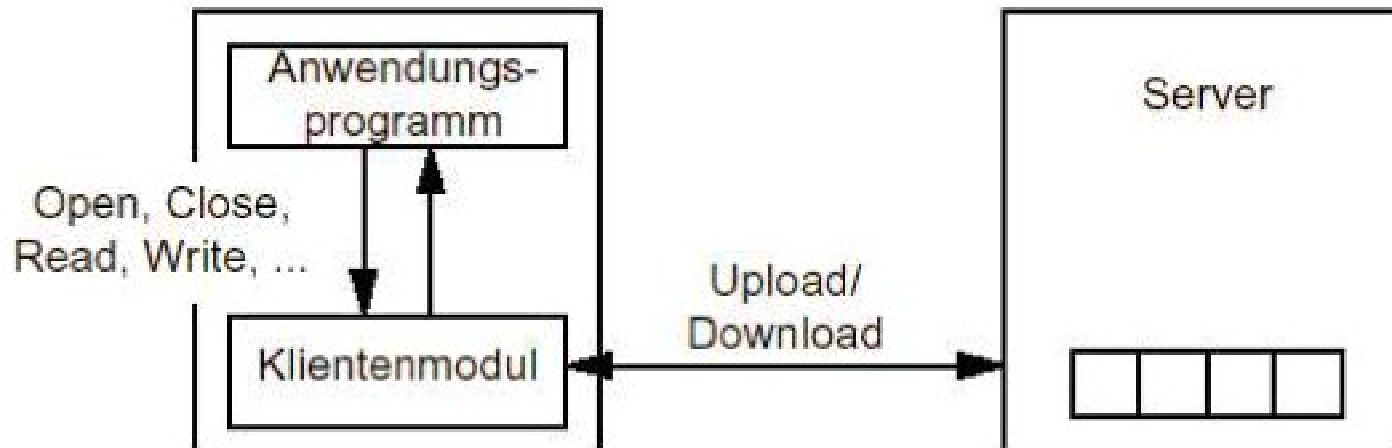
Verteilte Dateisysteme lassen sich aus architektonischer Sicht **einteilen** nach

- der Art des Zugriff

- dem Mechanismus, wie der Zustand einer Datei verwaltet wird.

4.1.1.Zugriffsmodelle

Das Zugriffsmodell legt fest, in welcher Art die Dateioperationen und -zugriffe (auch auf Verzeichnisse) zwischen Klient und Server abgewickelt werden.



Upload/Download-Modell

Das Upload/Download Modell lässt sich wie folgt charakterisiert:

- Dieses Modell entspricht einem Sitzungsmodell.

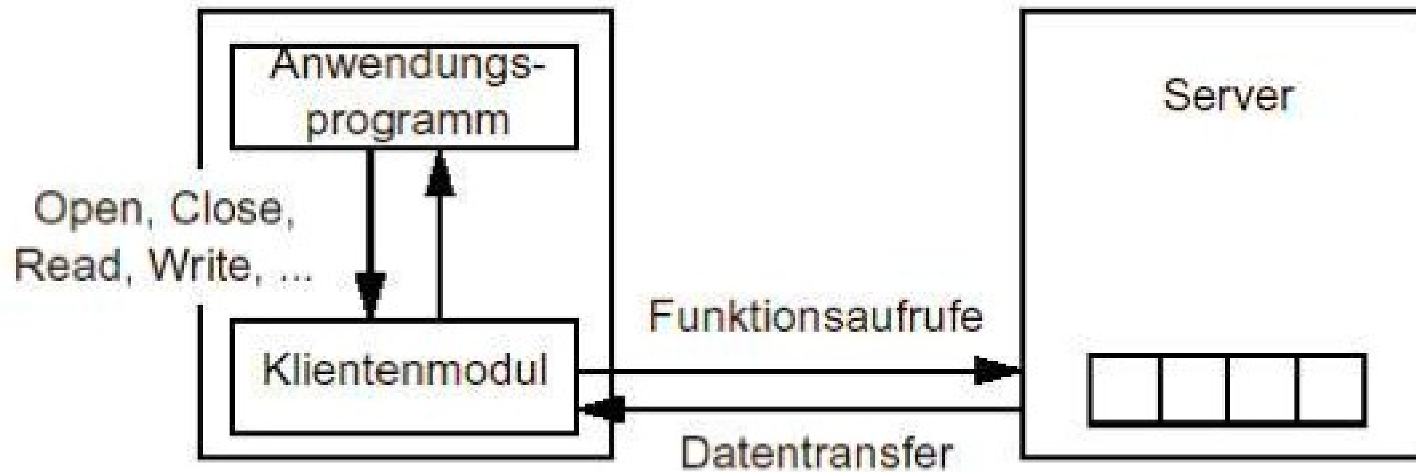
- Beim Öffnen einer Datei durch einen Klienten lädt der Server die Datei vollständig zum Klienten (`download`), der sie dann lokal bearbeiten kann.

- Benötigt der Klient die Datei nicht mehr, schließt er sie und sie wird zum Server zurück übertragen (`upload`).

Bewertung des Upload/Download-Modell:

- Der **Server** bietet eine sehr **einfache Schnittstelle** zum Klientententenmodul.
- Intern kennt der Server Zugriffe, um eine Datei sequentiell vom physikalische Speichermedium zu lesen bzw. darauf zu schreiben.
- Im **Klientenmodul** müssen alle **Dateioperationen implementiert** sein, auch ein Verzeichnismanagement.
- Der **Klient** benötigt ausreichend **Platz** zum lokalen Zwischenspeichern aller benötigten Dateien.
- Die **Netzwerkverbindung** wird nur beim **Öffnen** und **Schließen** von Dateien belastet, dann aber mit einem großen Volumen.

Remote-Access-Modell



Das Remote-Access Model list wie folgt charakterisiert:

- Alle **Zugriffe** eines Klienten auf eine Datei werden einzeln auf dem entfernten **Server** ausgeführt.

Bewertung des Remote-Access-Modell:

- Die meiste **Funktionalität** liegt auf der **Serverseite**.
- Das **Klientenmodul** beschränkt sich meist auf ein reines **Kommunikationsmodul**.
- Die Schnittstelle zwischen Klient und Server benötigt viel Funktionalität.

- Das **Netzwerk** wird **fortwährend belastet**, allerdings mit jeweils kleinerem Volumen.

4.1.2.Zustandslose und zustandsbehaftete Dateiserver

Durch den Zugriff eines Klienten auf eine Datei ergeben sich **dynamische Zustandsinformationen**:

- z.B. **welche Dateien** sind durch **welche Klienten geöffnet**,
- an welcher Stelle befindet sich der **Positionszeiger** einer Datei,
- welche **Sperren** sind gesetzt.

Solche Zustandsinformationen können auf Serverseite oder vom Klienten verwaltet werden.

Verwaltet der Server die Zustandsinformation, so nennt man ihn zustandsbehafteter Server, ansonsten zustandsloser Server.

Vorteile eines **zustandslosen Dateiservers**

- Benötigt **keinen Speicher** für Klienteninformation.
- Operationen zum Öffnen oder Schließen von Dateien sind unnötig, da sich diese Information nicht gemerkt werden kann.
- Das System kann **leichter fehlertolerant** realisiert werden, da sich ohne Austausch von Zustandsinformationen Replikations-und Backupserver einfacher realisieren lassen.
- Beim **Absturz eines Klienten** entstehen für den **Server keine Probleme**, da keine verwaiste Information vorhanden ist.

Vorteile eines **zustandsbehafteten** Dateiservers

- **Kürzere Nachrichten** genügen, um Zugriffe auf Dateien durchzuführen, da nicht mehr in jedem Zugriff alle benötigten Klienteninformationen übertragen werden müssen.
- **Schreib**- und **Lesezugriffe** sind **effizienter**, da z.B. die Positionszeiger einer im Zugriff befindlichen Datei bereits am richtigen Platz stehen.

- Es ist **Precaching** (durch Precaching holt der Server bereits Daten der zugegriffenen Datei von seiner Platte, die der Klient noch gar nicht angefordert hat) möglich, da alle Dateien bekannt sind, die sich im Zugriff befinden.

- Durch Kenntnis des Benutzungsprofils ist intelligentes Precaching realisierbar. Z.B. benötigt ein Compiler immer seine Bibliotheksdateien.

- Dateisperren können vom Server unterstützt werden.

4.2.Verzeichnisdienst

Hauptaufgabe des Verzeichnisdienstes ist das **Auflösen von Dateinamen**: Der Dateiname ist auf einen Zeiger, der auf den physikalischen Speicherungsort der Datei auf der Festplatte zeigt, abzubilden.

Um einen **Dateinamen aufzulösen**, muss man den zugehörigen **Pfadnamen**, der auch über mehrere Rechner verteilt sein kann, **durchlaufen**.

Unterschieden werden prinzipiell zwei **Vorgehensweisen**:

- Iterativ

- Rekursiv

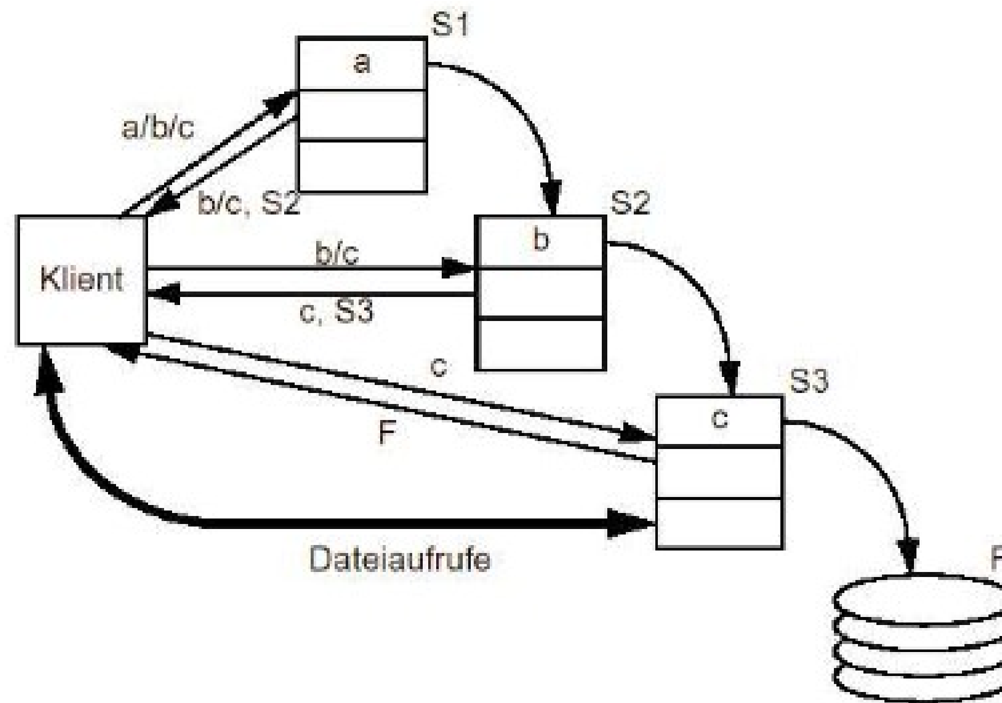
Iteratives Durchlaufen:

- Der Klient fragt beim Server, für den der erste Teil des Pfades passt, nach der Datei.
- Kann dieser den Zeiger auf die eigentliche Datei nicht liefern, weil er nicht den gesamten benötigten Verzeichnisbaum bei sich hält, gibt er dem Klienten den Pfad und Server zurück, bei dem der Klient weiter nachfragen kann.
- Dieses Vorgehen wiederholt der Klient, bis er den Zeiger auf die Datei erhält.
- Dann kann der Klient direkt über den verwaltenden Server auf die Datei zugreifen.

Der **Navigationsalgorithmus** ist vollständig auf **Klientenseite** implementiert.

Als Kommunikationsmodell lassen sich entfernte Prozeduraufrufe (**RPC**) einsetzen.

Veranschaulichung des iterativen Verfahrens:



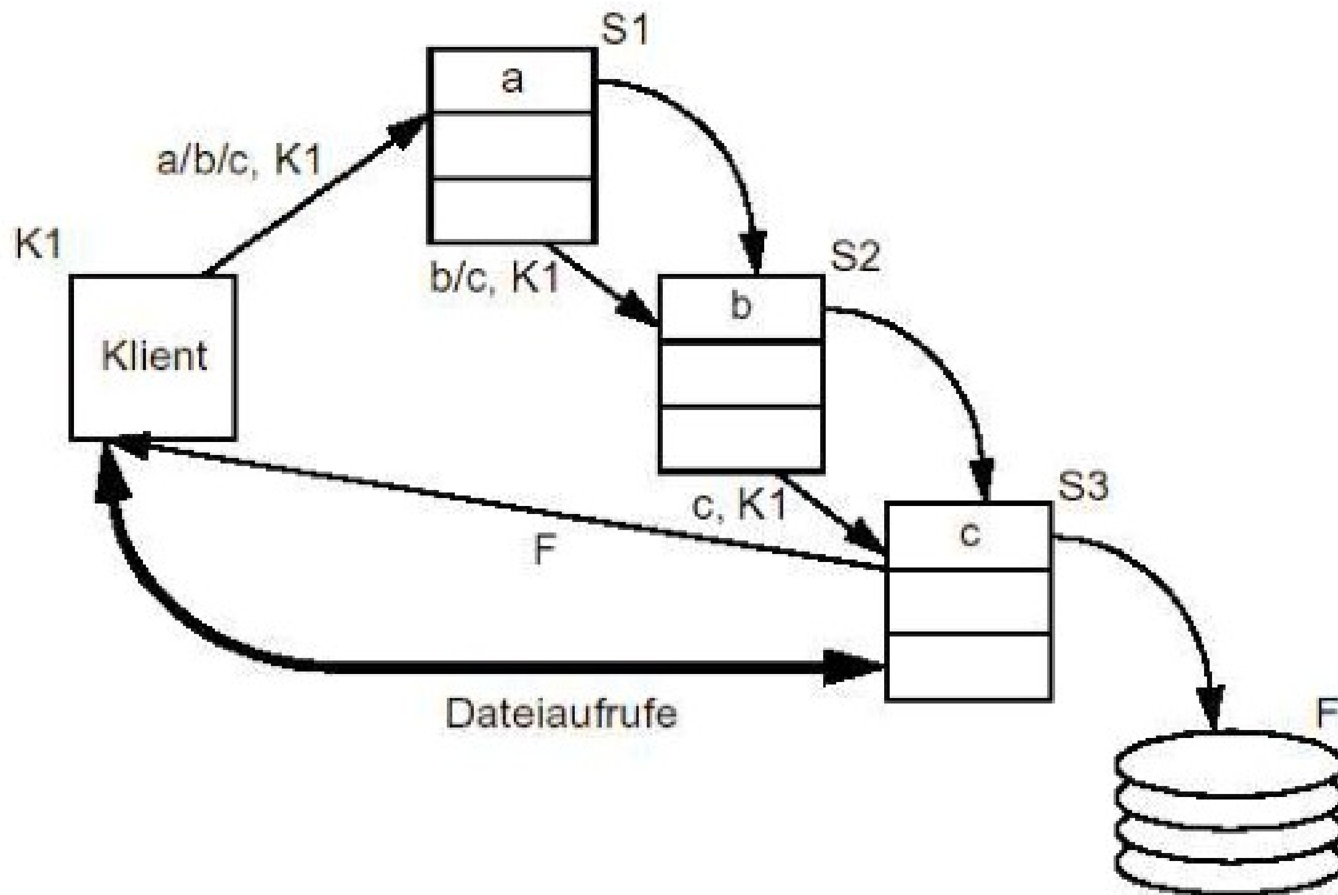
Rekursives Durchlaufen:

- Der Klient stellt an den Server die Anfrage bezüglich einer Datei.
- Kann der Server die Anfrage nicht beantworten, so gibt er sie mit der Information, welcher Klient die Anfrage gestellt hat, an den nächsten Server weiter.
- Ist der gesamte Pfadname aufgelöst, schickt der letzte Server den gewünschten Dateizeiger zum Klienten zurück.

- Die Dateizugriffe können dann zwischen Klient und Server erfolgen.

Wegen der Asymmetrie des Verfahrens kann die Kommunikation in diesem Fall nicht mehr über RPCs stattfinden. Es wird ein **asynchrones Kommunikationsmodell** benötigt.

Veranschaulichung des rekursiven Verfahrens:



4.3.Caching und Konsistenz

Caching bezeichnet das **Zwischenspeichern von Dateien**, um nicht immer langsame Plattenzugriffe (über ein Netz) durchführen zu müssen.

Je nach der **Art des Zwischenspeicherns** findet man folgende Varianten.

Caching im **Arbeitsspeicher des Servers**

- Befindet sich eine Datei bereits im Arbeitsspeicher des Servers, so wird bei erneutem Zugriff auf diese Datei der langsame Plattenzugriff eingespart.
- Der Zugriff über das Netz durch die Klienten bleibt jedoch erhalten.
- Da sich die Datei nach wie vor auf der Serverseite befindet, ist die Erhaltung der Konsistenz einfach zu gewährleisten.
- Das Caching verläuft gegenüber dem Klienten transparent.

Caching auf der **Klientenplatte**

- Der Klient legt benutzte Dateien als Cachekopie auf seiner lokalen Platte ab. Dies erspart bei erneutem Zugriff den Weg über das Netz.

- Die Erhaltung der Konsistenz von mehreren Cachekopien auf Klientenseite ist je nach Konsistenzmodell aufwendig.

Caching im **Arbeitsspeicher des Klienten**

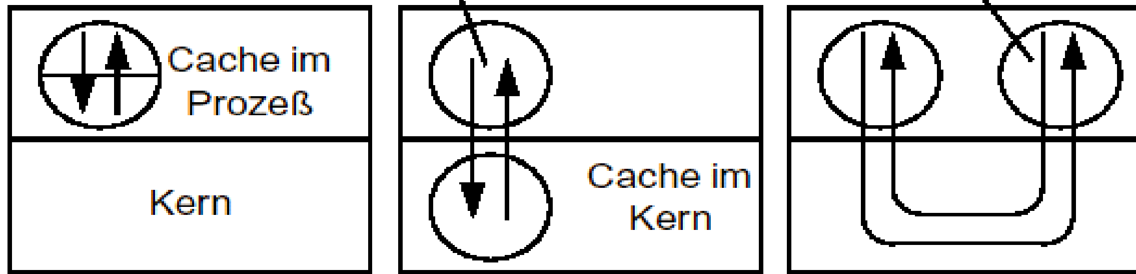
Hier werden Varianten je nach Zuordnung des Cachespeichers unterschieden:

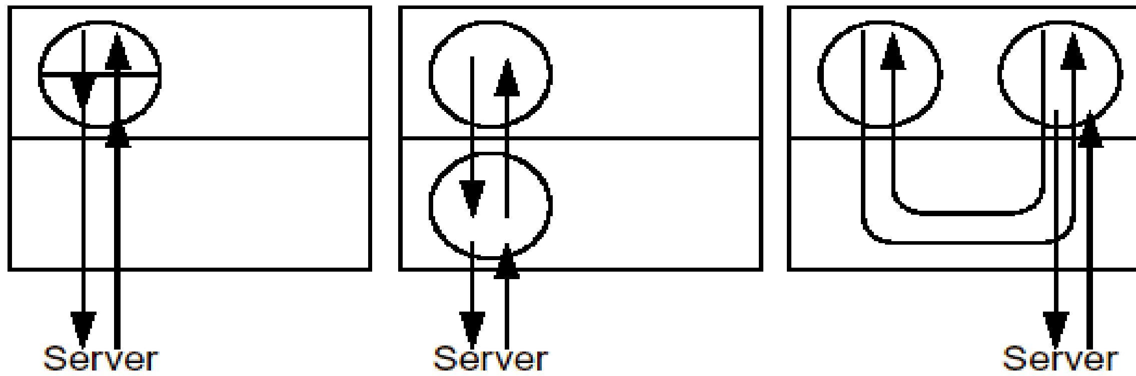
- Caching im **Klientenprozess**
  - Jeder Klientenprozeß verwaltet seinen eigenen Cache.
  - Die Caches anderer Prozesse sind dabei nicht sichtbar.
  - Dieses Verfahren benötigt nur einen geringen Verwaltungsaufwand.
  - Es ist speziell für Anwendungen geeignet, in denen eine Datei vom gleichen Prozess mehrmals geöffnet und geschlossen wird oder lange in Benutzung ist.
- Caching im **Betriebssystemkern**
  - Alle Prozesse teilen sich einen Cache, der vom Betriebssystem verwaltet wird.
  - Der Cachespeicher kann größer dimensioniert werden, als bei den einzelnen Caches für jeden Prozess.

- Mehrere Prozesse können von den Cacheinhalten profitieren, was eine höhere Trefferrate nach sich zieht.
- Jeder Dateizugriff ist ein Systemaufruf, auch bei einem Cachetreffer.

- Caching in einem **Cachemanager**
  - Das Caching wird von einem speziellen Anwendungsprozess übernommen.
  - Dadurch wird der Betriebssystemkern von spezifischem Dateisystem-Code freigehalten.
  - Damit die Effizienz des Caches nicht durch Paging des Betriebssystems zunichte gemacht wird, sollte der Cachemanager seine Cacheseiten gegenüber der Auslagerung sperren können.

Die folgende Abbildung stellt diese Varianten des **Caching im Speicher des Client** gegenüber:


Cachetreffer
Anwendungsprozess
Cachemanager
Cache im Prozeß
Kern
Cache im Kern


Cachefehler
Server
Server
Server

4.3.1.Algorithmen zur Cachekohärenz

Verfahren, die **Inhalt** des **Caches** und des **Dateisystem** auf dem Server in einem **konsistenten Zustand** zu halten, nennt man Cachekohärenz Algorithmen.

Unterschieden werden folgende Ansätze:

Write-Through

- Findet ein **Schreibzugriff** auf eine Datei statt, die im Cache steht, wird immer sowohl in den **Cache**, als auch in die **Datei beim Server** geschrieben.
- Schreibzugriffe laufen quasi wie bei cachelosen Systemen ab.
- Beim **Öffnen einer Datei aus dem Cache** besteht die Möglichkeit, dass diese **nicht mehr aktuell** ist.
- Deswegen muss der **Klient beim Server die Aktualität überprüfen**, oder der Server teilt jeweils seinen Klienten mit, dass eine Datei woanders geändert wurde.
- Je nach Sharingsemantik tritt dieses Problem auch beim Schreiben auf.
- Die Write-through-Strategie ist eher **ungeeignet für verteilte Dateisysteme**.

Delayed-Write

- Um häufige Server- und Netzwerkzugriffe zu sparen, werden **Änderungen gesammelt** und in so genannten **Bursts** zurück **geschrieben**.
- Dadurch erhält man eine **unklare, zeitabhängige Semantik**.

Write-on-Close

- Write-on-Close entspricht einer Sitzungssemantik.
- Durch verzögertes Write-on-Close kann ein nachfolgendes Löschen der Datei noch berücksichtigt und ein unnötiges Zurückschreiben verhindert werden.

Zentrale Koordinierung

- Eine **zentrale Koordinierungsstelle kennt alle Cacheeinträge** aller Klienten.
- Dieser Koordinator wird von jedem Schreibzugriff in Kenntnis gesetzt und kann damit alle Klienten informieren, die eine betreffende Datei im Cache halten.
- Diese Strategie ist weder robust gegen Ausfälle, noch gut skalierbar.

4.4.Replikation

Eine weitere Möglichkeit, die Leistung eines verteilten Dateisystems zu erhöhen, ist die Replikation von Daten auf verschiedenen Servern.

Dadurch wird zudem eine bessere Zuverlässigkeit und Verfügbarkeit erreicht.

Replikation kann wie folgt beschrieben werden:

- **Daten** werden (bei Servern) **mehrfach** zur Verfügung gestellt.
- Dies geschieht weitestgehend unabhängig davon, welche Daten gerade momentan von Nutzern benötigt werden.
- Replikation ist eine bewusst gesteuerte Informationskopie.
- Replikate werden per Nachrichtenaustausch angelegt und auch modifiziert.

Um die Daten der Replikate konsistent zu halten, ist ein **Replikationsmanagement** erforderlich. Dabei sollte beachtet werden, dass

- es nicht immer sinnvoll ist, alle Daten zu replizieren;

- nur die häufig benötigten Daten sollten repliziert werden sollten.

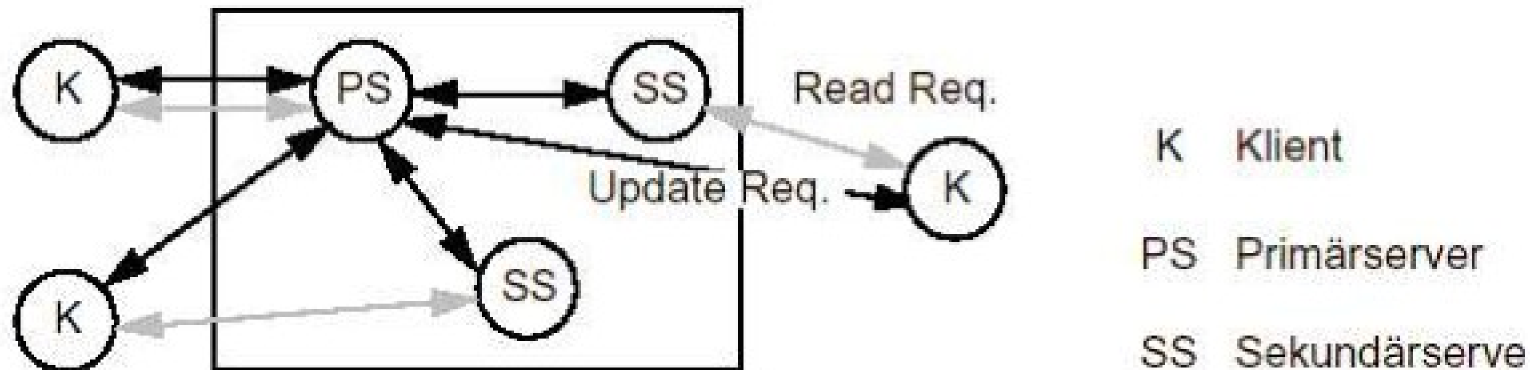
Man unterscheidet folgende **Anlagestrategien für Replikate**:

- **Explizite** Replikation
  - Der **Klient** steuert die Replikation.
  - Er legt die **Replikate** explizit bei **mehreren Servern** an.
  - Beim Zugriff verwendet er eines der Replikate.
  - Da die Replikate von einem Klienten angelegt werden, hat auch nur dieser die Kenntnis über sie.
  - Die **Konsistenzwahrung** liegt beim **Klienten** selbst.
  - **Andere Klienten** können **nicht** von den Replikaten **profitieren**.
- **Lazy** Replikation
  - Ein **Hauptserver** wird bei der Erzeugung eines neuen Objekts angesprochen
  - Dies legt irgendwann später **Replikate** bei **anderen Servern** an, ohne dass der Klient davon erfährt.
  - Die **Konsistenzwahrung** liegt beim **Hauptserver**.

- Replikation in einer **Servergruppe**
  - Per **Gruppenkommunikation** wird an **mehrere Server** gleichzeitig die Aufforderung ein Objekt und somit **jeweils ein Replikat** anzulegen gesendet.
  - Alle **Schreibzugriffe** müssen an die **gesamte Gruppe** gehen.
  - Die **Konsistenzwahrung** liegt beim **Gruppenkommunikationssystem** und ist durch dessen Zuverlässigkeits- und Ordnungssemantik bestimmt.

Nachdem die Strategie für die Anlage von Replikaten bestimmt ist, ist festzulegen, wie die Replikate verwaltet werden; man kann folgende **Verwaltungsstragegien** unterscheiden:

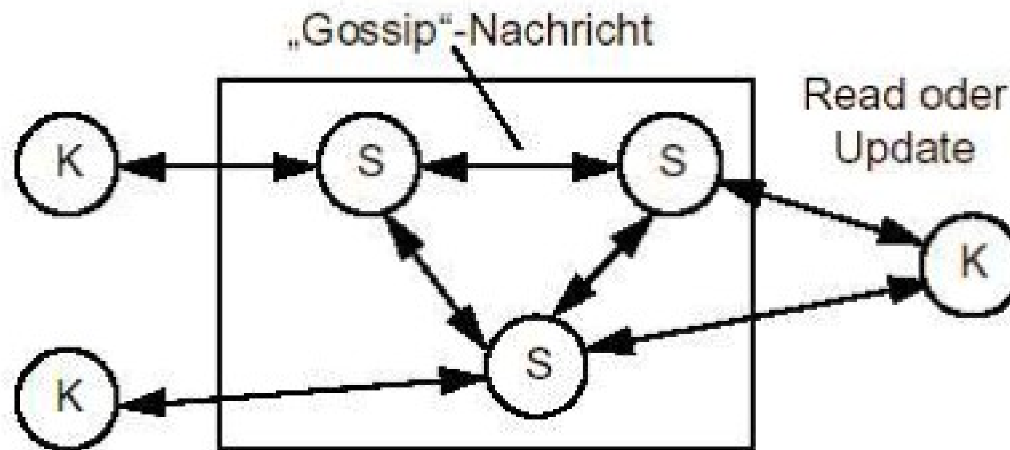
- **Primärkopie**



- Alle **Schreibzugriffe** werden immer an einen **Primärserver** geschickt.
- Dieser sorgt dafür, dass auch alle **Sekundärserver** auf den **aktuellen Stand** gebracht werden.
- **Lesezugriffe** können auch von **Sekundärservern** bedient werden.
- Je nachdem, wie und wann der Primärserver veränderte Daten an die Sekundärserver propagiert, können schwache bis starke Konsistenzmodelle realisiert werden.
- Soll immer konsistent auch bei Sekundärservern gelesen werden können, muss die Veränderung atomar im Serververbund erfolgen
- Bei nicht atomarem Aktualisieren können Klienten bei Sekundärservern „alte“ Werte lesen.
- Bei einem Absturz des Primärservers sollte ein Sekundärserver dessen Rolle übernehmen.

* Lazy-Update

  - Alle Server können Lese- und Schreibzugriffe bedienen.



- Erhält ein Server eine Aktualisierungsnachricht, schickt er diese nicht sofort zu den anderen Servern. Er fasst mehrere Updates zusammen und sendet sie dann periodisch an die anderen.

5. Network File System

Das Network File System (NFS) von Sun ist der defacto Standard bei verteilten Dateisystemen.

Das ursprüngliches Ziel war es, ein verteiltes Dateisystem für

- heterogene Hardwareplattformen und
- Betriebssysteme

zu realisieren.

Die größte Verbreitung liegt in Unix-Umgebungen.

5.1. NFS Architektur

Die NFS-Klienten- und -Serversoftware ist jeweils in den **Betriebssystemkern** eingebunden.

In den Unix-Implementierungen ist die Client/Server Relation **symmetrisch**, d.h. Klienten können gleichzeitig Server sein und umgekehrt.

Server und Klient kommunizieren über das **NFS-Protokoll**.

Als Abstraktionsschicht über den lokalen Dateisystemen und NFS liegt ein **virtuelles Dateisystem VFS**.

- Alle Dateioperationen eines Anwendungsprozesses gehen zuerst an das virtuelle Dateisystem.

- VFS entscheidet, ob sich die Datei lokal oder entfernt befindet.

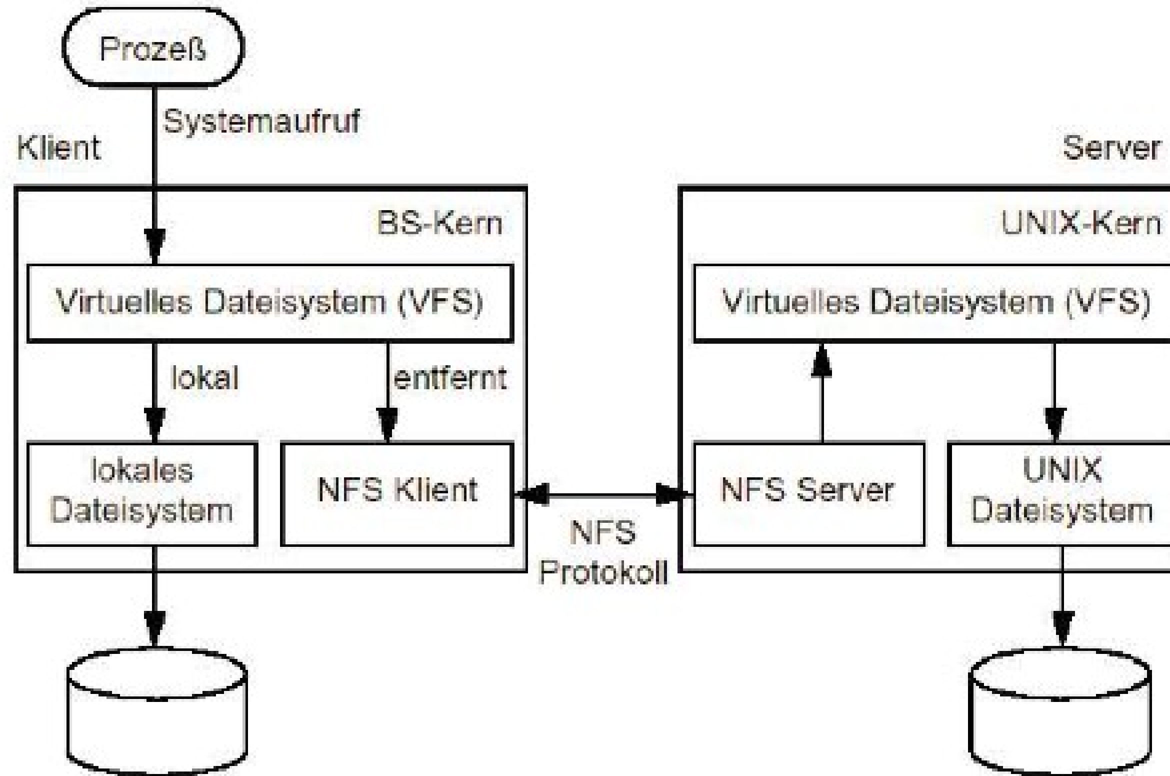
- Lokale Anforderungen werden an das lokale Dateisystem weitergereicht und von dort bedient.

- Bei entfernten Dateien wird der Aufruf an den NFS-Klienten gegeben.Dieser kommuniziert dann über das Netzwerk mit dem NFS-Server, der den Aufruf entfernt bedient.

NFS erlaubt es, plattenlose Klienten zu betreiben, die über kein eigenes lokales Dateisystem verfügen.

Die NFS Architektur kann wie folgt veranschaulicht werden:

Das VFS ist eine Erweiterung des Unix-Dateisystem mit folgender Charakteristik:



- VFS enthält **v-nodes** (virtual nodes) als Datei-Handles, die auf
  - **i-nodes** (Datei-Handles des lokalen Dateisystems), oder auf
  - r-nodes (remote nodes) zeigen.

- Die r-nodes liegen im NFS-Klienten und beinhalten die eigentlichen Datei-Handles, die vom entfernten Server geliefert wurden.

- Der Anwendungsprozess erhält beim Öffnen einer Datei einen Dateideskriptor, der einem v-node im VFS zugeordnet ist.

- Um eine Datei physikalisch zu lokalisieren, muss das virtuelle Dateisystem eine mehrfach verzeigerte Struktur durchlaufen (Unix-Vorlesung).

5.2.Remote Mounting

Ein Server bietet in der Datei **`/etc/exports`** eine **Exporttabelle** aller Verzeichnisse an, die er für NFS zur Verfügung stellt.

Ein Klient hängt die gewünschten Verzeichnisse in seine eigene Verzeichnisstruktur durch den Aufruf eines Mount-Dienstes.

Der Mount-Dienst versorgt das virtuelle Dateisystem mit den entsprechenden Informationen.

Das Mounting eines entfernten Verzeichnisses erfolgt beim Klienten entweder

- interaktiv durch den Benutzer, der dazu jedoch Root-Rechte haben muss, oder

- alle gewünschten Mountaufrufe sind im Startupscript `/etc/rc` gespeichert. Dann wird das Mounting beim Hochfahren des Systems erledigt.

Mounten erfolgt durch

- `mount (server, directoryname, mountpoint)` kontaktiert den Server.

- Der Server schickt den Datei-Handle des Verzeichnisses zum Klienten.

- Der Datei-Handle beinhaltet den Dateisystemtyp, die angesprochene Platte, die i-node Nummer der Datei bzw. des Verzeichnisses und die Schutzbits.

- Die Verwaltung dieser Datei-Handles liegt bei VFS.

5.3.NFS Protokoll

Das NFS-Protokoll ist für eine **zustandslose** Implementierung der NFS-Server ausgelegt.

Jeder Auftrag an einen Server enthält sämtliche für die Bearbeitung notwendigen Informationen des Klienten und der Datei, die der Klient bearbeitet.

Im Protokoll ist kein Öffnen und Schließen von Dateien vorgesehen.

Neben Verwaltungsoperationen des Verzeichnisdienstes gibt es drei Grundoperationen:

- `lookup (dirfilehandle, filename)`
  Im angegebenen Verzeichnis wird die entsprechende Datei gesucht. Der Datei-Handle und die Attribute der Datei werden zurückgeliefert.

- `read (filehandle, offset, count)`
  In der angegebenen Datei werden ab der Position `offset` `count` Bytes geliefert.

- `write (filehandle, offset, count, data)`
  In der angegebenen Datei werden ab der Position `offset` `count` Bytes mit dem Inhalt von `data` beschrieben.

Die Protokolle für das **Mounting**, sowie für den **Verzeichnis- und Dateizugriff** werden über Sun-**RPC** abgewickelt.

Die Authentisierung der Klienten kann über öffentliche Schlüssel (aus dem Network Information Service, NIS) zusätzlich stärker geschützt werden, als es die üblichen Zugriffskontrollbits (`rwx`) zulassen..

5.4.Implementierungsaspekte

Folgende Grundprinzipien sind implementiert:

- Das **Kommunikationssystem** ist **UDP** mit Sun-**RPC**.
- Als **Fehlersemanit** ist "**At-Most-Once-Semantik**" realisiert.
- Der **Transfer** zwischen Klient und Server erfolgt in **8 Kilobyte Blöcken**.
- Um hohe Trefferwahrscheinlichkeit zu haben, liest der NFS-Klient liest zusätzlich den darauffolgenden Block (**Read-ahead-Caching**).
- Durch **Delayed-Write** werden Schreibzugriffe verzögert, bis ein 8 Kilobyte großer Block zusammengekommen ist.
- Wird eine **Datei geschlossen**, werden sofort alle **Änderungen zum Server** geschickt.
- **Caching**
  - Die Server cachen für die NFS-Klienten transparent schon gelesene Dateien in ihrem Speicher. write-through Semantik lässt keine Konsistenzprobleme auftreten.
  - Die Klienten führen
    - einen Cache für Datei-Handles und

      - einen Cache für Dateien

   ohne Konsistenzwahrung, d.h die Inhalte werden nicht unter den Klienten abgeglichen.

- **Datei-Inkonsistenzen** werden durch folgende Mechanismen **vermieden**:
  - Ist eine Datei beim Öffnen im Cache des Klienten, wird der lokale **Zeitstempel** mit dem Modifikations-Zeitstempel beim Server verglichen.
  - Ist die Kopie älter, wird sie invalidiert und erneut geladen.
  - Jeder **Cacheblock** besitzt eine **Uhr**, die für Dateiblöcke nach 3 Sek. und für Verzeichnisse nach 30 Sek. ablaufen.
  - Bei jedem Ablaufen wird beim Server auf Aktualität geprüft.
  - Beim Schreibzugriff wird der entsprechende Block im Cache als „dirty" markiert. Die markierten **Blöcke** werden alle **30 Sek.** zum Server propagiert (**sync**).

5.5.Bewertung

- Der Zugang zum NFS ist auf Programmebene transparent.

    - Die Aufrufschnittstelle ist durch das virtuelle Dateisystem identisch zur Schnittstelle zum lokalen Dateisystem.

    - **Ortstransparenz** ist möglich, falls bei allen Klienten eine global eindeutige Verzeichnisstruktur angelegt ist wie im Labor).

- NFS ist **nicht** tatsächlich **fehlertransparent**.

    - Die **Server** arbeiten **zustandslos**, also ist ein Absturz eines Klienten unkritisch.

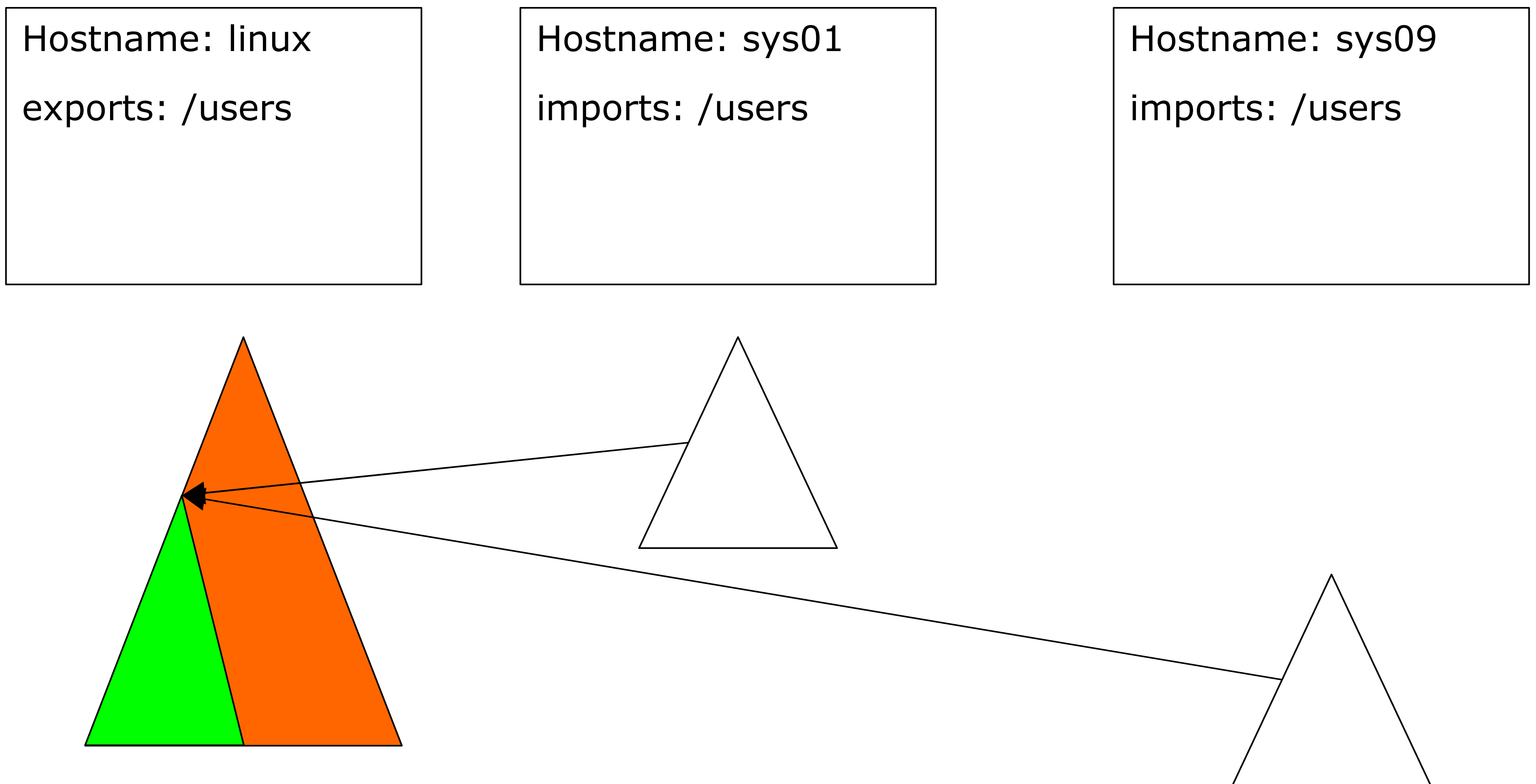
- Zur Leistungssteigerung findet Caching auf Klienten- und Serverseite statt.

- Es gibt keine vom System unterstützte **Replikation**.

- Die Sharing-Semantik will sich an die Einzelkopie-Semantik anlehnen ist jedoch durch das periodische Aktualisieren unscharf. Der gemeinsame Zugriff sollte deshalb über eine externe Synchronisation geregelt werden, z.B. durch Dateisperren.

- NFS ist ursprünglich für **wenige Stationen** ausgelegt (<50), die den gleichen Dateibaum benutzen. Die Skalierung auf größere Umgebungen ist nur mit Administrationsaufwand sinnvoll.

5.6.Beispiel: Unix-Labor



Die Heimatverzeichnisse aller Benutzer sind auf dem Server (linux) abgelegt.

Der Server hat dieses Verzeichnis exportiert (/users).

Jeder Client (sys01 – sys09) hat das Verzeichnis /users des Servers gemountet. Deshalb ist der Zugriff auf die Daten von allen Client-Rechner aus möglich.

5.7.Arbeitsweise von NFS

Ein Client versucht, ein Verzeichnis von einem entfernten Host an sein lokales Verzeichnis zu mounten, genau wie er dies mit einem physikalischen Gerät machen würde. Allerdings ist die für ein entferntes Verzeichnis zu verwendende Syntax anders. Soll zum Beispiel das Verzeichnis */tmp* vom Host "linux" an */xxx* auf "sys02" gemountet werden, führt der Administrator auf "sys02" das folgenden Kommando aus:

```
sys02# mount -t nfs linux:/tmp /xxx
```

mount versucht, über **RPC** eine Verbindung mit dem Mount-Dämon (***mountd***) auf "linux" herzu-stellen. Der Server überprüft, ob "sys02" die Erlaubnis besitzt, das fragliche Verzeichnis zu mounten; wenn ja, liefert er einen Datei-Handle zurück. Dieser Handle wird bei allen weiteren Anfragen nach Dateien unter */xxx* verwendet.

Greift jemand über NFS auf eine Datei zu, setzt der Kernel einen **RPC**-Aufruf an ***nfsd*** (den NFS-Dämon) auf der Server-Maschine ab. Dieser Aufruf enthält den Datei-Handle, den Namen der ge-wünschten Datei und den Benutzer- und Gruppen-ID als Parameter. Diese werden verwendet, um die Zugriffsrechte auf die angegebene Datei zu ermitteln. Um das Lesen und Modifizieren von Dateien durch nicht-autorisierte Benutzer zu verhindern, müssen die **Benutzer**- und Gruppen-IDs auf **beiden Hosts identisch** sein.

Bei den meisten UNIX-Implementierungen wird die NFS-Funktionalität sowohl für Clients als auch für Server durch Dämonen realisiert, die auf Kernel-Ebene angesiedelt sind und während der Bootphase vom Benutzerspeicher aus gestartet werden. Dies ist auf dem Serverhost der NFS-Dämon (*nfsd*) und auf dem Client-Host der *Block I/O Dämon* (*biod*). Um den Durchsatz zu verbessern, arbeitet *biod* mit asynchroner Ein-/Ausgabe unter Gebrauch von Read-Ahead und Write-Behind. Häufig werden auch mehrere *nfsd*-Dämonen gleichzeitig verwendet.

Die **NFS-Implementierung von Linux** ist etwas anders, weil der Client-Kode eng mit dem VFS-Layer (»**Virtual File System**«) des Kernels verbunden ist und daher keine zusätzliche Kontrolle durch *biod* benötigt. Auf der anderen Seite läuft der Server vollständig im Benutzerspeicher, d.h. es ist aufgrund der damit verbundenen Synchronisations-Fragen nahezu unmöglich, mehrere Kopien des Servers zur selben Zeit laufen zu lassen. NFS vorbereiten

Bevor man NFS benutzen kann, sei es nur als Client oder als Server, müssen Sie zuerst sicherstellen, daß die entsprechende **NFS-Unterstützung** in den **Kernel** integriert (kompiliert) ist. Neuere Kernel haben dazu ein einfaches Interface im Dateisystem *proc*, nämlich die Datei */proc/filesystems*, die Sie einfach mit *cat* ausgeben können:

```
$ cat /proc/file systems
nodev   bdev
nodev   proc
nodev   sockfs
nodev   tmpfs
nodev   shm
nodev   pipefs
        ext2
        minix
        msdos
        vfat
        iso9660
nodev   nfs
nodev   devpts
nodev   usbdevfs
nodev   autofs
```

Fehlt `nfs` in dieser Liste, muss der Kernel neu kompiliert und dabei NFS mit eingebunden werden.

5.8. Ein NFS-Volume mounten

NFS-Volumes werden fast genauso gemountet wie die normalen Dateisysteme auch. Man verwendet dabei für *mount* die folgende Syntax:

```
# mount -t nfs nfs_volume local_dir options
```

Eine ganze Reihe zusätzlicher Optionen kann beim Mounten eines NFS-Volumes angegeben werden. Diese können entweder auf die Option *-o* in der Kommandozeile folgen, oder im Optionsfeld von */etc/fstab* für dieses Volume stehen. In beiden Fällen werden mehrere Optionen durch Kommata voneinander getrennt. In der Kommandozeile angegebene Optionen überschreiben immer die in *fstab* stehenden Werte.

Nachfolgend ein Beispiel-Eintrag aus */etc/fstab*:

```
# Volume                Mountpunkt       Typ    Optionen
news:/usr/spool/news  /usr/spool/news   nfs   timeo=14,intr
```

Dieses Volume kann mit dem folgenden Befehl gemountet werden:

```
# mount news:/usr/spool/news
```

Fehlt ein solcher *fstab*-Eintrag, sehen NFS-*mount*s aufwendiger. Nehmen wir zum Beispiel an, daß Sie die Home-Verzeichnisse der Benutzer von einer Maschine namens moonshot aus mounten. Um eine Blockgröße von 4 K für Schreib/Lese-Operationen zu verwenden, benutzen Sie den folgenden Befehl:

```
# mount moonshot:/home /home -o rsize=4096,wsize=4096
```

Eine vollständige Liste aller gültigen Optionen ist in der *nfs(5)*-Manpage beschrieben, die bei Rick Sladkeys NFS-fähigem *mount*-Tool enthalten ist.

Nachfolgend ein Teil der Optionen:

rsize=n und wsize=n

Bestimmt die bei Schreib- bzw. Leseanforderungen von NFS-Clients verwendete Datagramm-Größe. Im Moment ist diese aufgrund der oben beschriebenen Einschränkungen in der UDP-Datagramm-Größe auf 1024 Byte voreingestellt.

timeo=n

Bestimmt die Zeit (in Zehntelsekunden), die ein NFS-Client auf den Abschluß einer Anforderung wartet. Der voreingestellte Wert ist 7 (0,7 Sekunden).

hard

Markiere dieses Volume explizit als »hart« gemountet. Per Voreinstellung aktiviert.

soft

»Weiches« Mounten des Verzeichnisses (im Gegensatz zu hartem Mounten).

intr

Unterbrechung von NFS-Aufrufen über Signale möglich. Nützlich, wenn ein Server nicht antwortet und der Client abgebrochen werden muß.

Mit Ausnahme von `rsize` und `wsize` wirken sich all diese Optionen auf das Verhalten des Client aus, wenn der Server kurzfristig nicht erreichbar sein sollte. Sie spielen auf folgende Weise zusammen: Sendet der Client eine Anforderung an den NFS-Server, erwartet er, daß die Operation innerhalb einer bestimmten Zeit abgeschlossen ist. Diese Zeit kann mit der Option `timeout` fest-

gelegt werden. Wird innerhalb dieser Zeit keine Bestätigung empfangen, kommt es zu einem sogenannten *kleinen Timeout*. Die Operation wird nun wiederholt, wobei das Timeout-Intervall verdoppelt wird. Wird der maximale Timeout von 60 Sekunden erreicht, tritt ein *großer Timeout* auf.

Per Voreinstellung gibt ein Client bei einem schwerwiegenden Timeout eine Fehlermeldung auf der Console aus und versucht es erneut. Dabei wird das ursprüngliche Timeout-Intervall verdoppelt. Theoretisch könnte dies immer so weitergehen. Volumes, die eine Operation störrisch wiederholen, bis der Server wieder verfügbar ist, werden als *fest gemountet* (hard-mounted) bezeichnet. Die andere Variante wird als *weich gemountet* (soft-mounted) bezeichnet und generiert einen I/O-Fehler für den rufenden Prozess, wenn ein schwerwiegender Fehler auftritt.

5.9. Die NFS-Dämonen

Wenn man anderen Hosts NFS-Dienste anbieten will, müssen die *nfsd*- und *mountd*-Dämonen auf der Maschine laufen. Als RPC-basierte Programme werden sie nicht von *inetd* verwaltet, sondern werden während der Bootphase gestartet und registrieren sich selbst beim Portmapper. Daher muss man sicherstellen, daß die Programme erst gestartet werden, wenn *rpc.portmap* schon läuft.

5.10. Die exports-Datei

Für jeden Client werden die Zugriffsmöglichkeiten bestimmt, über die auf die Dateien auf dem Server zugegriffen werden kann. Dieser Zugriff wird in der Datei */etc/exports* festgelegt, die die gemeinsam genutzten Dateien enthält.

Per Voreinstellung erlaubt *mount* niemandem, Verzeichnisse des lokalen Hosts über NFS zu mounten, was eine sehr vernünftige Einstellung ist. Um einem oder mehreren Hosts den NFS-Zugriff auf ein Verzeichnis zu erlauben, müssen Sie es *exportieren*, d. h. in der Datei *export* eintragen. Eine Beispieldatei könnte so aussehen:

```
# exports-Datei für linux
/home               *(rw) vstout(rw) vlight(rw)
/usr/TeX            sys??(ro) vstout(ro) vlight(ro)
```

Jede Zeile definiert ein Verzeichnis und die Hosts, die es mounten dürfen. Ein Hostname ist üblicherweise ein voll qualifizierter Domainname, kann zusätzlich aber auch die Platzhalter `*` und `?` enthalten, die auf dieselbe Weise funktionieren wie bei der Bourne-Shell

Bei der Prüfung eines Client-Host gegen die exports-Datei ermittelt *mountd* den Hostnamen des Client mit Hilfe eines *gethostbyaddr*-Aufrufs. Unter DNS liefert dieser Aufruf den kanonischen Hostnamen des Client zurück, d. h. Sie dürfen keine Aliases in *exports* verwenden. Ohne DNS wird der erste Hostname aus *hosts* zurückgeliefert, bei dem die Adresse des Client übereinstimmt.

Dem Host-Namen kann eine Liste mit durch Kommata getrennten Optionen folgen, die in eckigen Klammern eingeschlossen sind. Diese Optionen können die folgenden Werte annehmen:

insecure

Auf diese Maschine ist nicht-authentisierter Zugriff erlaubt.

unix-rpc

Von dieser Maschine wird UNIX-Domain RPC-Authentisierung benötigt. Dies bedeutet einfach, daß Anforderungen von reservierten Internet-Ports (die Port-Nummer muß kleiner als 1024 sein) stammen müssen. Diese Option ist per Voreinstellung aktiviert.

secure-rpc

Von dieser Maschine wird Secure-RPC-Authentizierung benötigt. Dies ist bislang noch nicht implementiert. Siehe Suns Dokumentation zu Secure-RPC.

kerberos

Von dieser Maschine wird Kerberos-Authentizierung benötigt. Dies ist bislang noch nicht implementiert. Siehe die MIT-Dokumentation zum Kerberos-Authentizierungssystem.

ro

Mounte die Datei-Hierarchie ohne Schreibmöglichkeit.

rw

Mounte die Datei-Hierarchie mit Schreib-/Leserechten. Diese Option ist per Voreinstellung aktiv.

5.11.Der Linux-Auto-Mounter

Manchmal ist es Verschwendung, alle NFS-Volumes zu mounten, auf die ein Benutzer möglicherweise zugreifen möchte. Dazu existiert ein so genannter ***Auto-Mounter***. Dabei handelt es sich

um einen Dämon, der jedes NFS-Volume automatisch und völlig transparent mountet und nach einer gewissen Zeit wieder unmountet, wenn es nicht verwendet wurde.