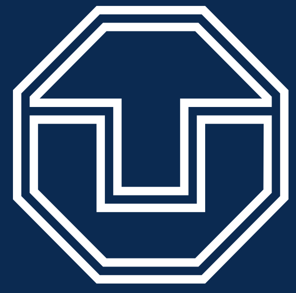


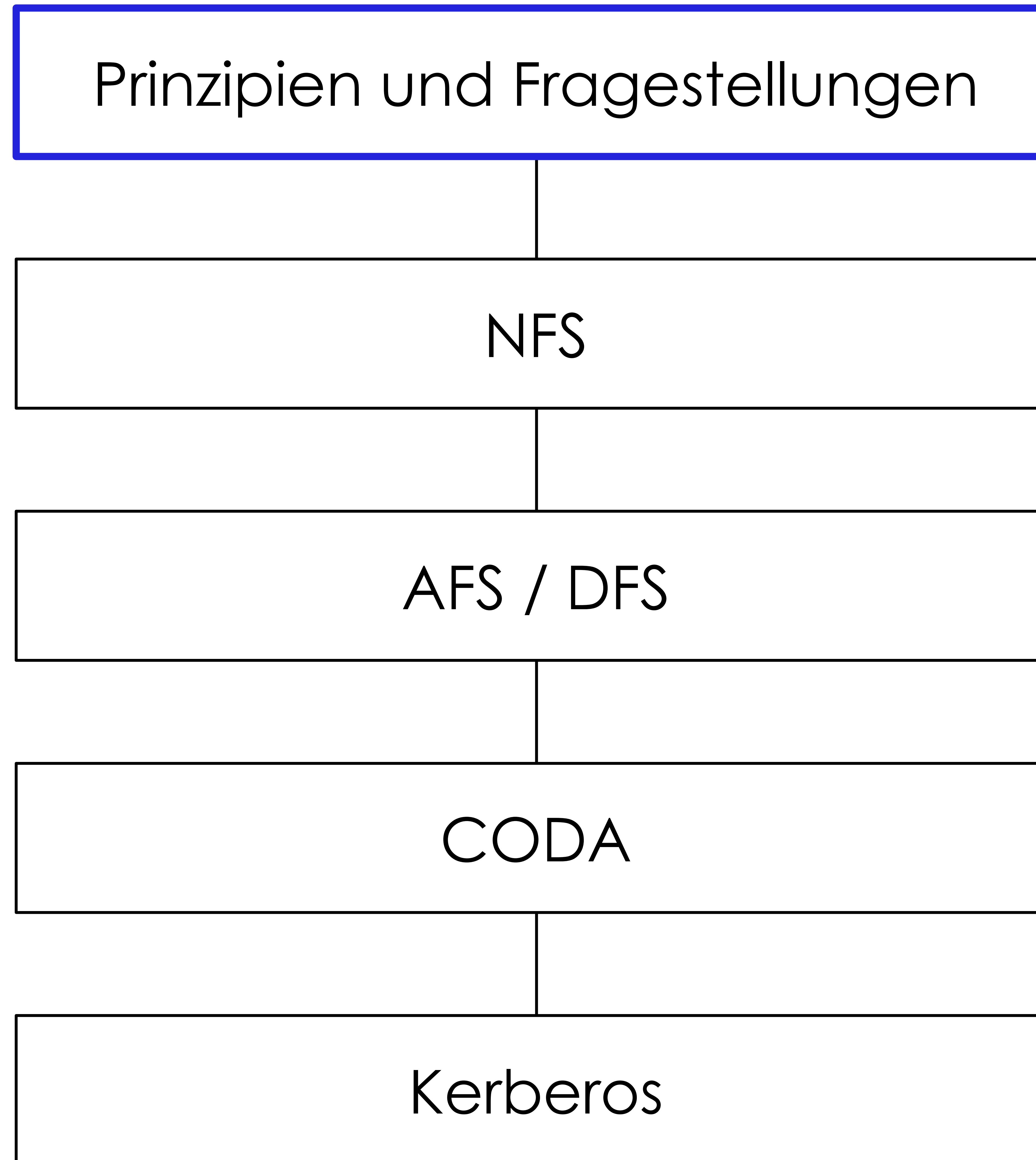
TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture, Operating Systems Group

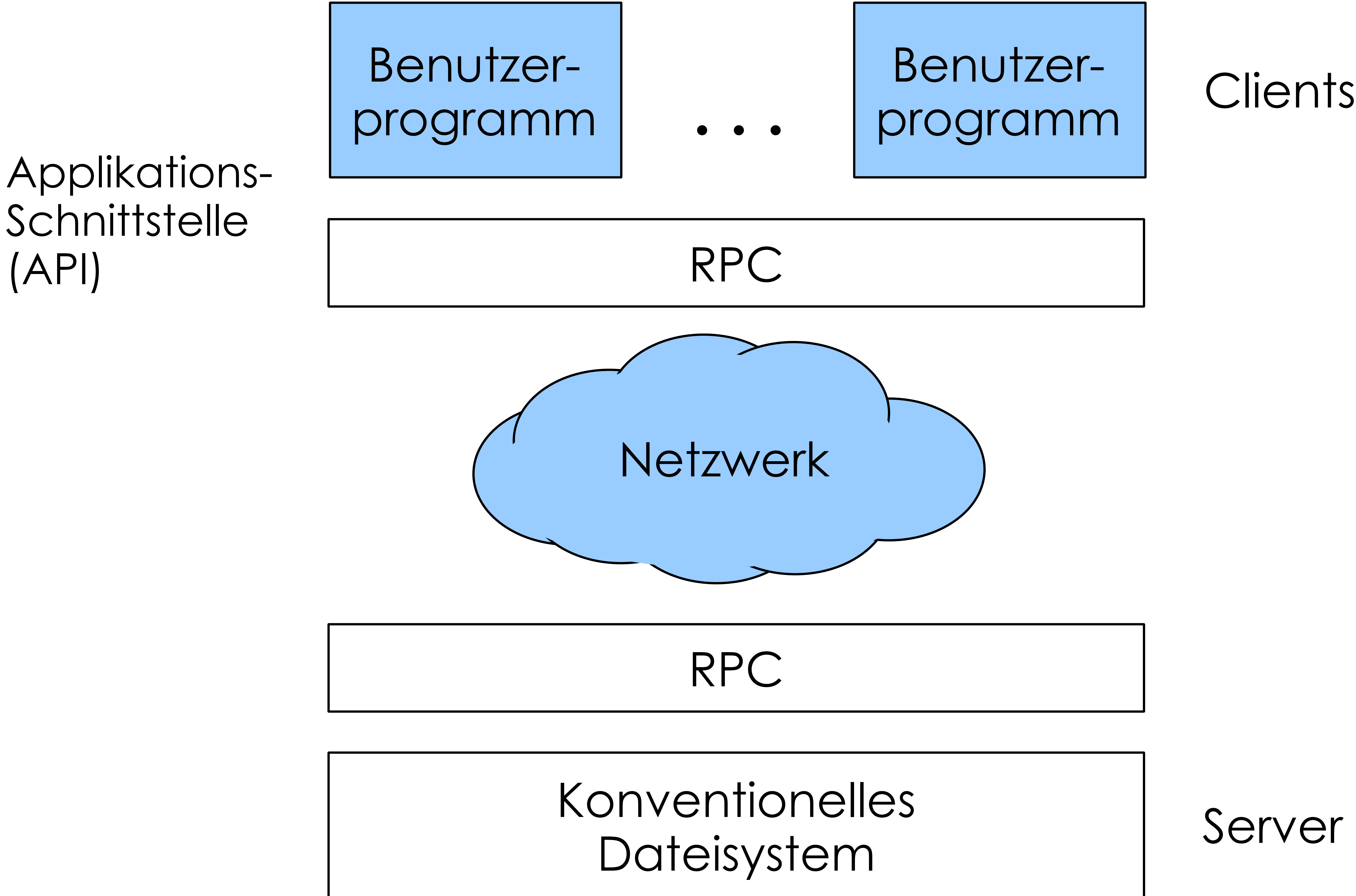
VERTEILTE DATEISYSTEME

MICHAEL ROITZSCH

Wegweiser



Naheliegend: Systemaufruf durch RPC ersetzen?



Nachteile der einfachen Variante

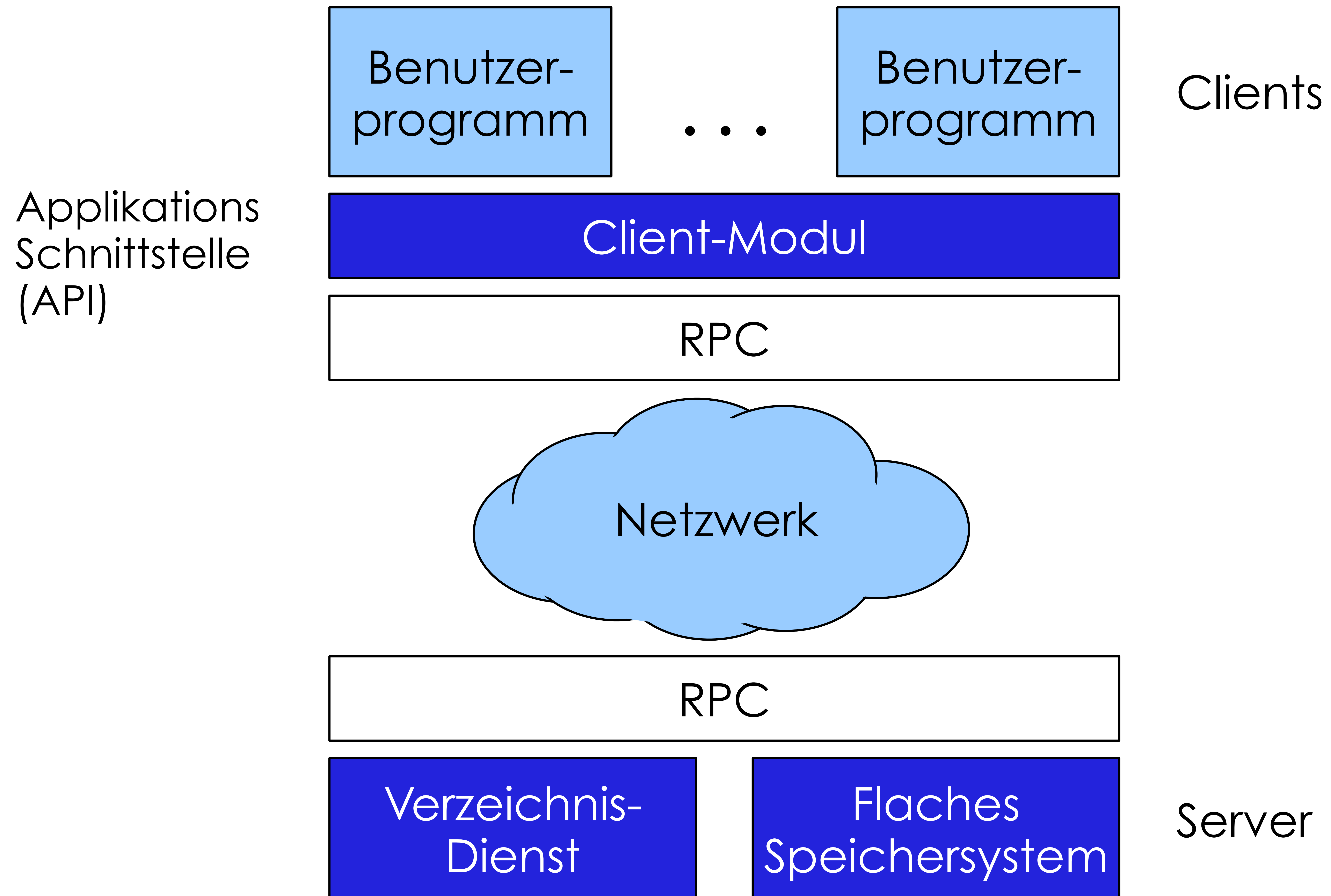
Leistungsfähigkeit mangelhaft

- zu viele Botschaften
- Overhead bei kleinen Lese-/Schreib-Einheiten zu groß
- Fehlersemantik nicht berücksichtigt

Nicht offen …

- für unterschiedliche Client-Betriebssysteme
- für unterschiedliche Namensformen für Dateisysteme

Architektur verteilter Dateisysteme



Aufgaben

Client-Modul

- Anpassung an Client-Betriebssystem
- Zusammenspiel von Verzeichnisdienst und flachem Speichersystem
- Fehlerbehandlung
- Caching

Flaches Speichersystem / Key-Value-Store / Chunk Storage

- Operationen zum Lesen, Schreiben, Erzeugen, Löschen …
  persistenter Daten auf einer (flachen) Menge von Speicherobjekten
- Zugriffssteuerung (Access Control)

Namensdienste / Directory Services

- Abbildung symbolischer Namen auf eindeutige Identifikatoren

Fragestellungen und Ziele bei Entwurf

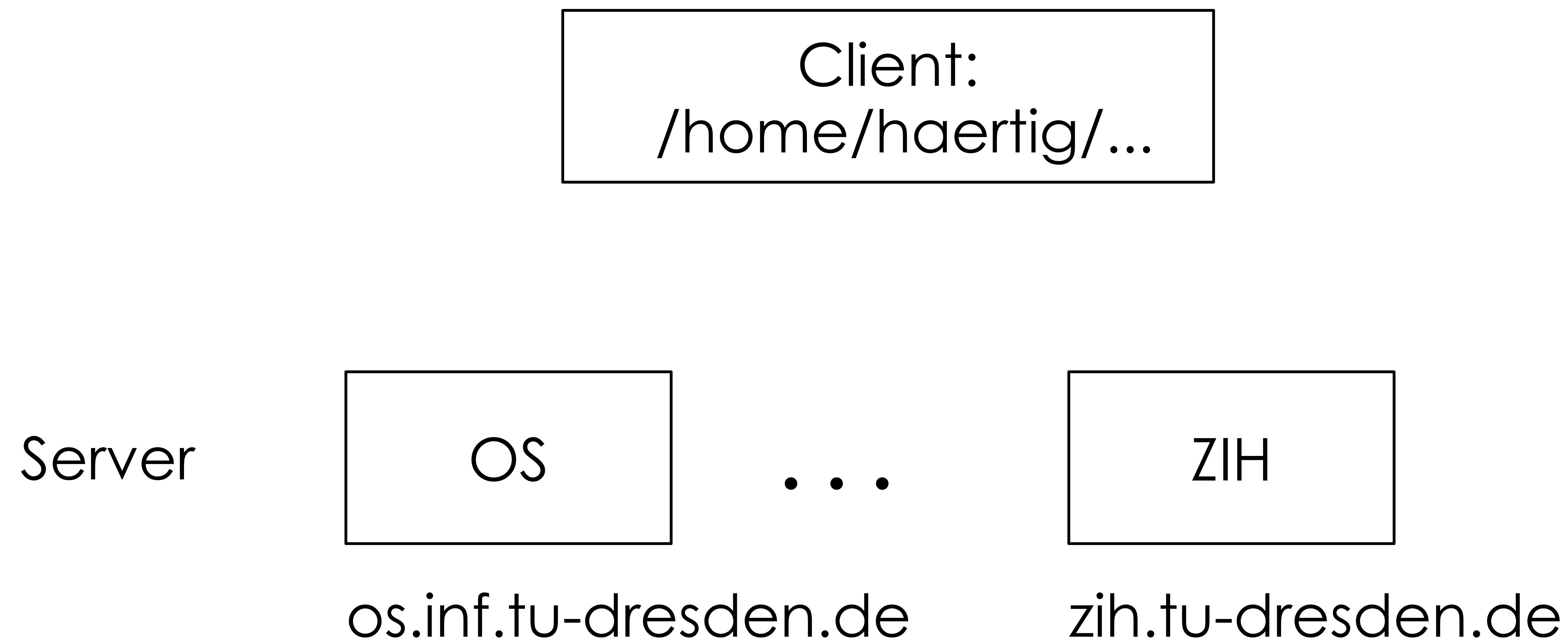
- Zugriff:
  gleichartiger Zugriff auf nahe und entfernte Dateien
- Ort:
  Grad der Unabhängigkeit von Netztopologie
- Fehler:
  Grad der Sichtbarkeit für Benutzer
- Replikation
- Skalierbarkeit:
  Anpassbarkeit an sehr große Zahlen von Clients und Servern
- Heterogenität:
  unterschiedliche Basis-Hardware und Betriebssystem

Fragestellungen und Ziele bei Entwurf

- Schreib-Konsistenz
  Sichtbar-Werden bei parallelen Schreiboperationen
- Effizienz
  Bandbreite, Latenz
- Administration
  zentral / dezentral
- Echtzeitfähigkeit
  Zusagen für Bandbreite oder Latenz (Quality of Service)
- Schutz
  Angreifermodell, Schutzziele

Namen im verteilten Dateisystem

Grundsätzlicher Ausgangspunkt:
hierarchische Namensräume (Pfadnamen)



Namen im verteilten Dateisystem

Variante 1: Globale Namen

- Servernamen (und Benutzernamen) werden Bestandteil des Dateinamens
- eingeschränkte Ortstransparenz:
  man muss Servernamen kennen
- skaliert weltweit

Beispiel: SSH / SCP

- `scp os.inf.tu-dresden.de:/home/otto/tmp/bla ./`
- `git clone mr188034@login.zih.tu-dresden.de:dev/L4`

Namen im verteilten Dateisystem

Variante 2: Client-System oder -Prozess baut Namensraum auf

- Namen danach ortsunabhängig
- Clients können Dateien (transparent) zwischen Speicherorten migrieren
- für Programme ändert sich nichts

Beispiel: Remote Mount

- `mount os.inf.tu-dresden.de:/usr /home/haertig/os/`

Namen im verteilten Dateisystem

Variante 3: Darstellung eines einheitlichen Bildes

- alle Clients haben dieselbe Sicht auf das verteilte Speichersystem
- gegebenenfalls eingeschränkt durch Zugriffsrechte
- ortsunabhängige Namen
- Administratoren der Server können Dateien migrieren

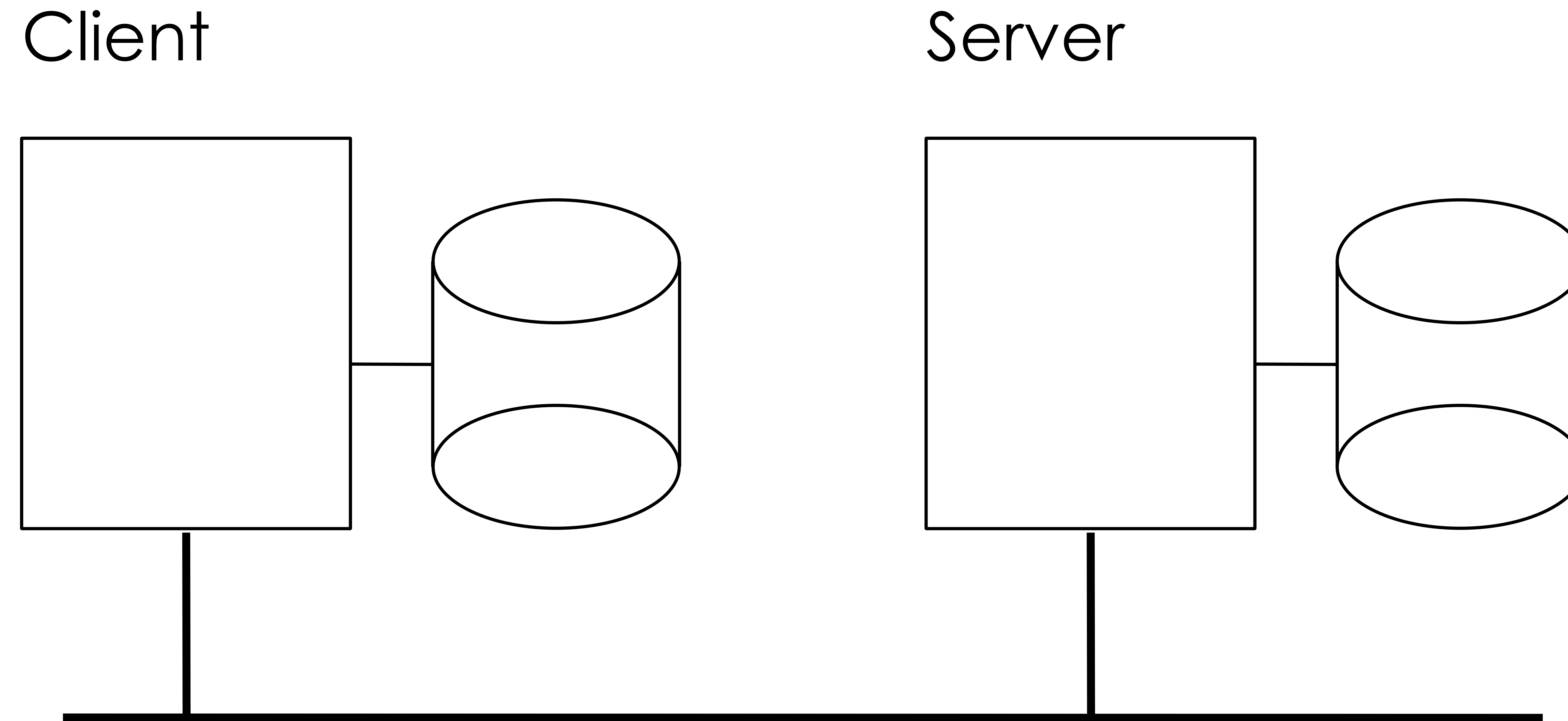
Beispiele: Dropbox, GitHub

Flaches Speichersystem

Benennung von Speicherobjekten

- Finden: eindeutiger Identifikator → Adresse
  - für jede Datei
  - für Dateigruppen
    (Menge von Dateien, die nur als Ganzes migriert)
- Eindeutigkeit:
  - zentrale Autorität
  - große Zufallszahl
- Fälschungssicherheit
- erlaubte Operationen

Caching



Fragestellungen der Cache-Platzierung

- Persistenz des Caches nach Abstürzen
- Zugreifbarkeit der Dateien bei Netzpartitionierungen
- Effizienz

Caching

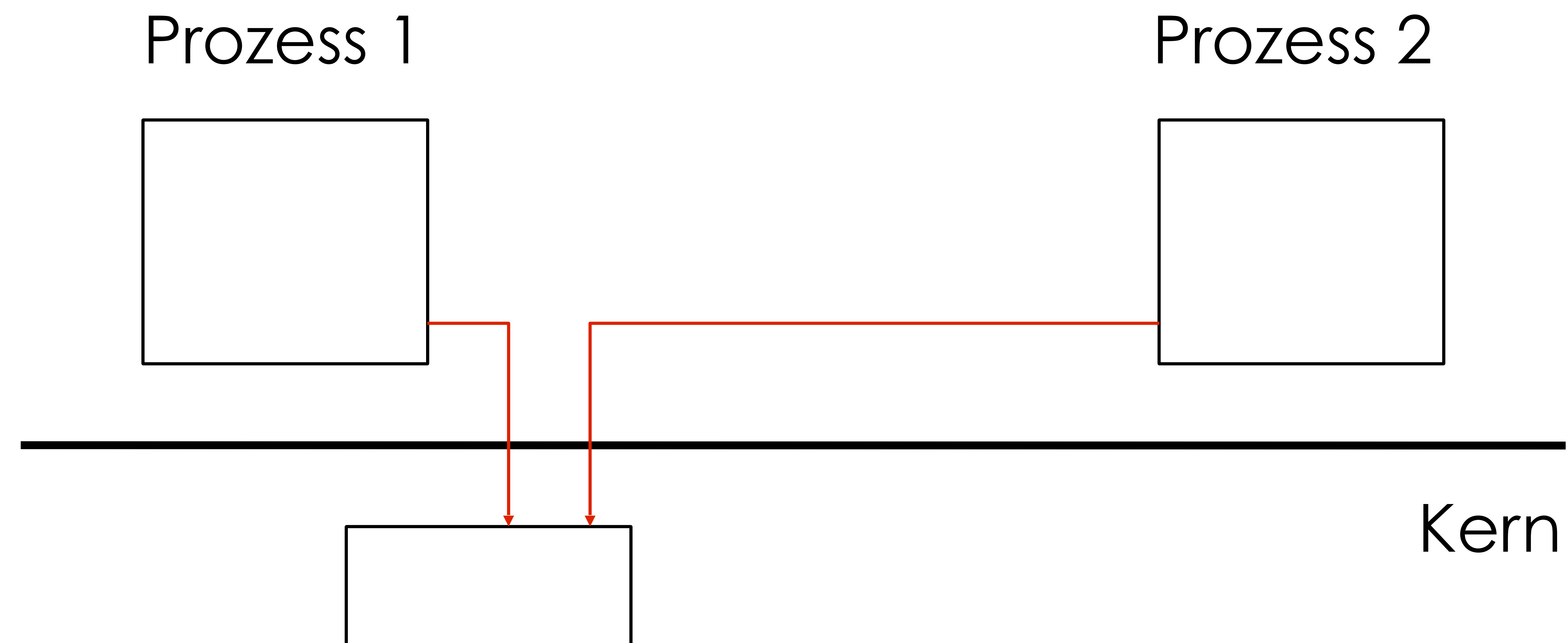
Granularität der Zwischenspeicherung

- Variante 1: kein Cache
  - jede Operation durch RPC
- Variante 2: ganze Dateien
  - vollständige Übertragung bei erstem Zugriff
  - Speicherung auf Client-Platte
  - Zurückschreiben bei `close()`
- Variante 3: Blöcke
  - asynchrones Zurückschreiben

Schreib-Konsistenz

Variante 1: one copy semantics

- Jeder schreibende Zugriff wird in allen Prozessen in derselben Reihenfolge gesehen
- z. B. Unix-Dateisystem lokal



Schreib-Konsistenz

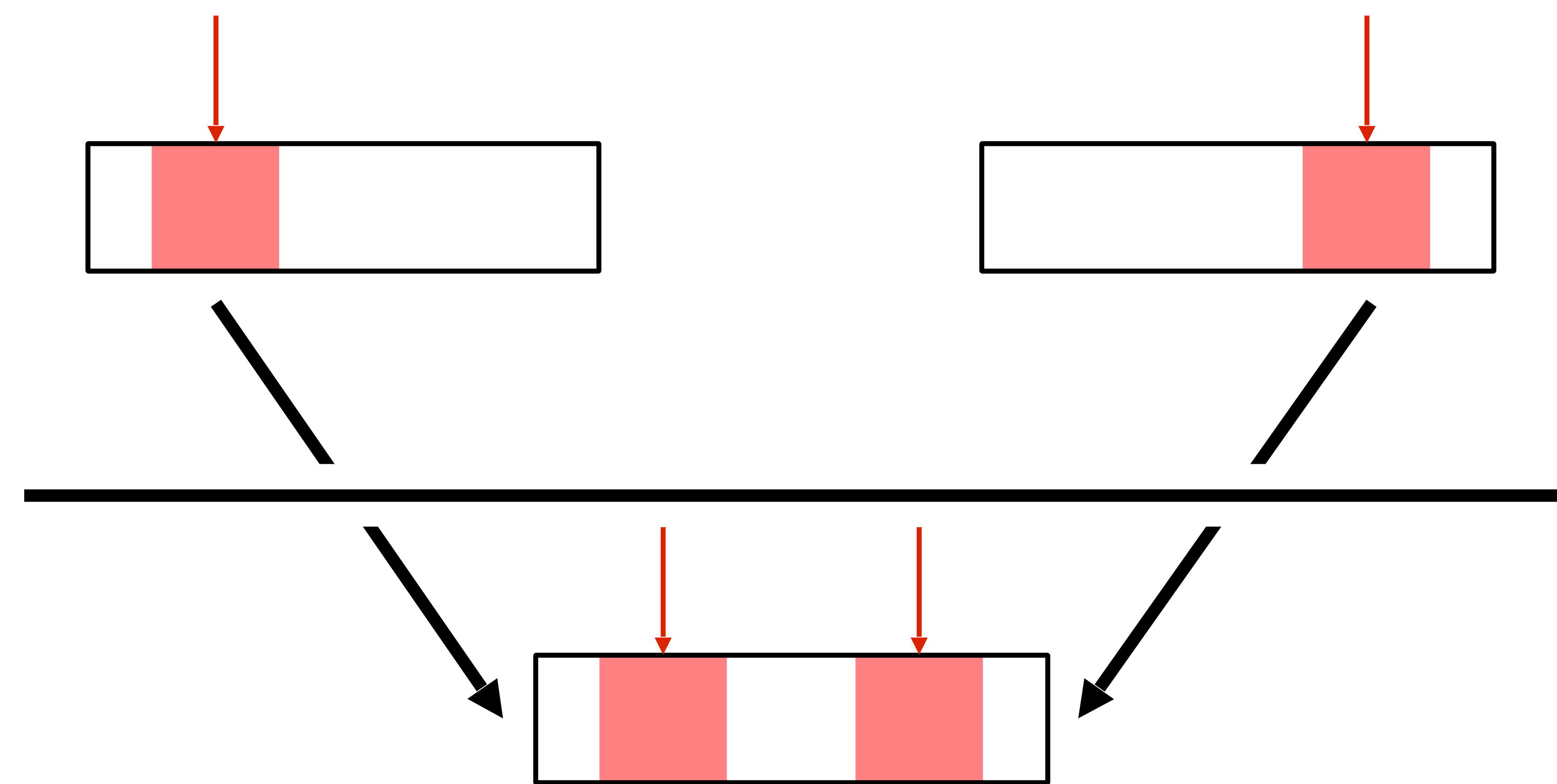
Variante 2: Sitzungssemantik / Session Semantics

- Sitzung: open (read/write)* close
- jeder Prozess sieht:
- die von ihm verursachten Änderungen innerhalb einer Sitzung
- die von anderen Prozessen verursachten Änderungen
  - bei Beginn der eigenen Sitzung (beim Öffnen)
  - nach dem Ende der Sitzung anderer Prozesse

Schreib-Konsistenz

Variante 3: keine Konsistenz

- false sharing: Überschreibung benachbarter Bereiche



→ bei Zurückschreiben des ganzen Blocks wird eine Schreiboperation ungültig

Variante 4: Bindung an Synchronisationsoperationen

- Sperren von Bereichen
- „Commit"-Operationen (explizites Schreiben)

Zustandslos oder Zustandsbehaftet

Zustandsinformationen über Clients bei Servern

- ja → stateful
- nein → stateless
- Beispiel für klientenspezifischen Zustand: Dateiposition
- stateful: **`read(fd, buf, count)`**
- stateless: **`read(ufid, position, buf, count)`**

Zustandslos oder Zustandsbehaftet

Vorteile (nach Tanenbaum)

stateless	stateful
Fehlertoleranz	kürzere request-Botschaften
keine OPEN/CLOSE – Operation nötig	effizienter
keine aufwendigen Tabellen auf Server	Vorauslesen möglich
keine Limitierung der Zahl offener Dateien	Nicht-idempotente Operation möglich
keine Probleme bei Klienten-Abstürzen	File-locking möglich

Replikation

Mehrere Kopien von Dateien

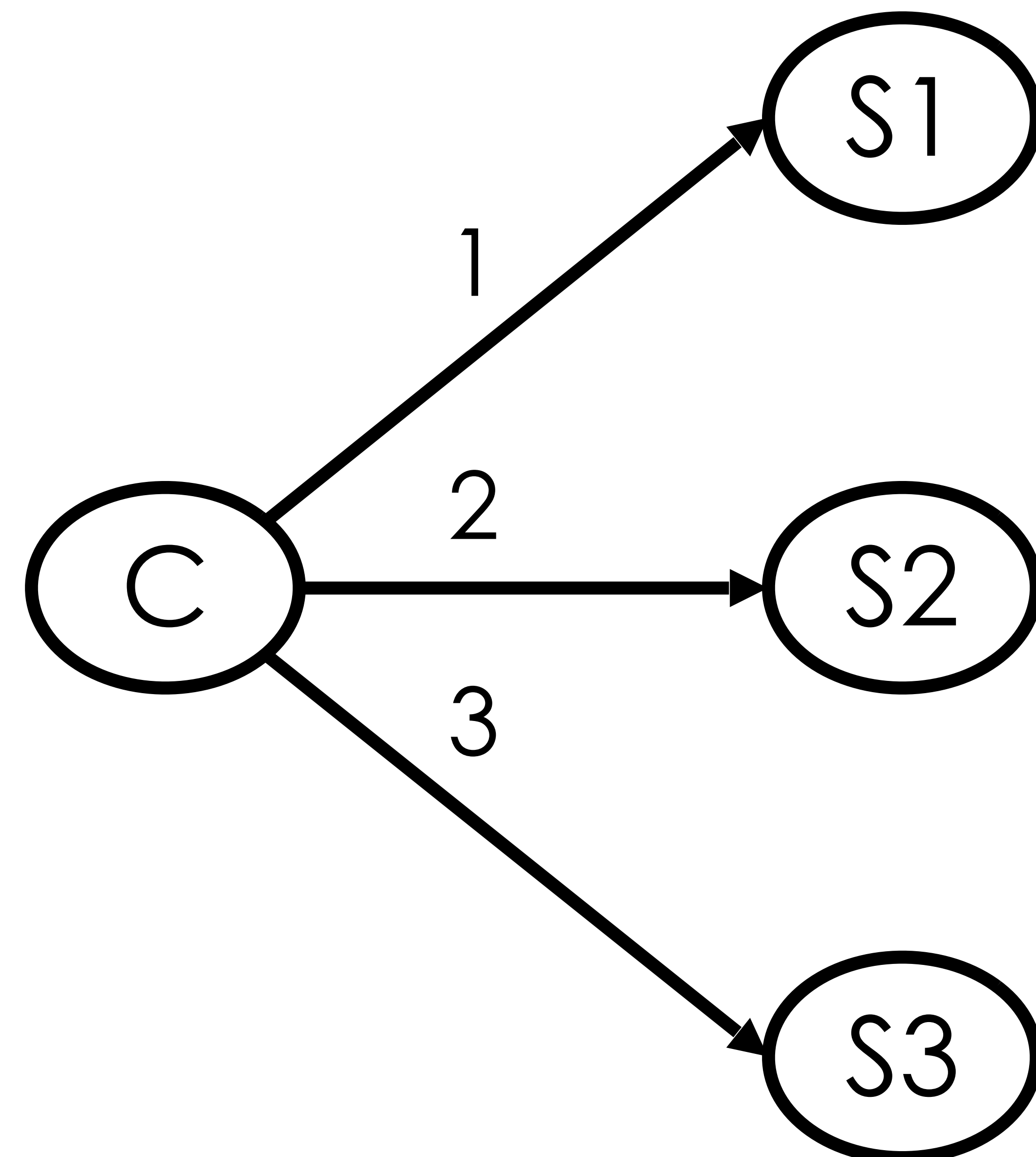
- Lesen: von einer Kopie
- Schreiben: alle Kopien

Motivation

- Effizienz: Zugriff auf weniger belastete Server
- Zuverlässigkeit, Verfügbarkeit
  - Tolerierung von Serverausfällen
  - Tolerierung von Netzpartitionierung

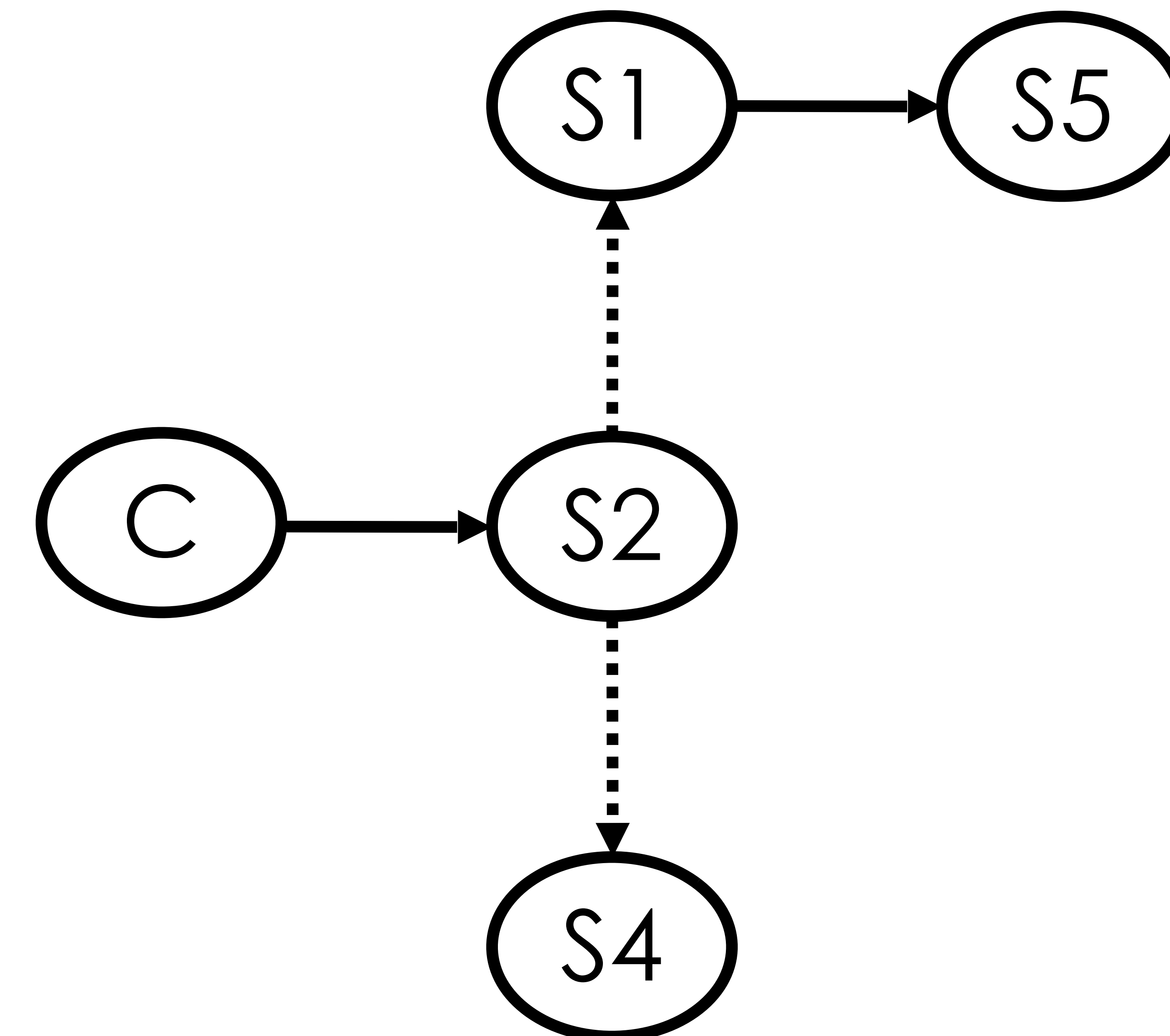
Replikationsmodelle

Client-organisierte Replikation



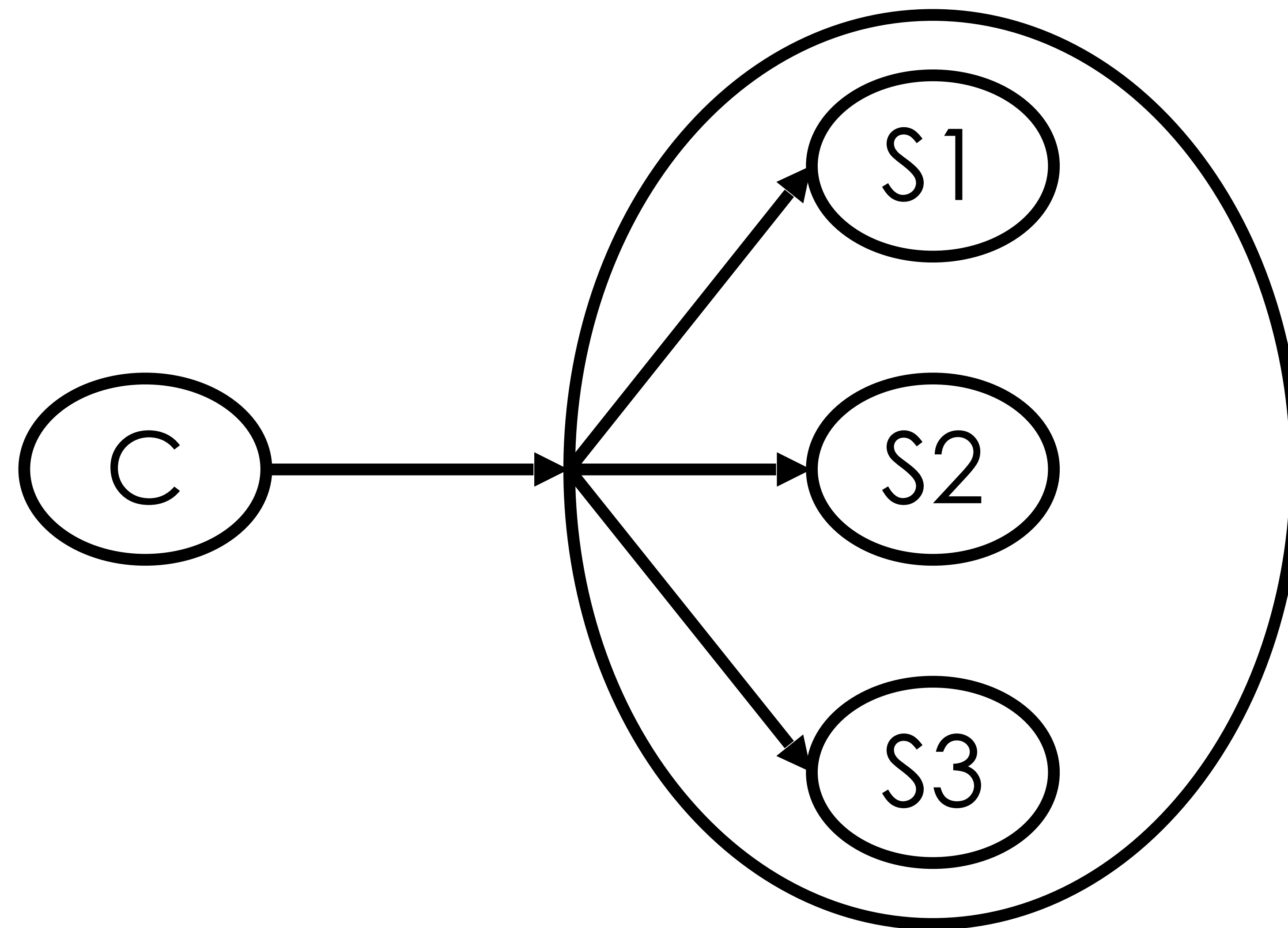
Server-organisierte Replikation (lazy replication)

grundsätzlich vs. nur im Fehlerfall



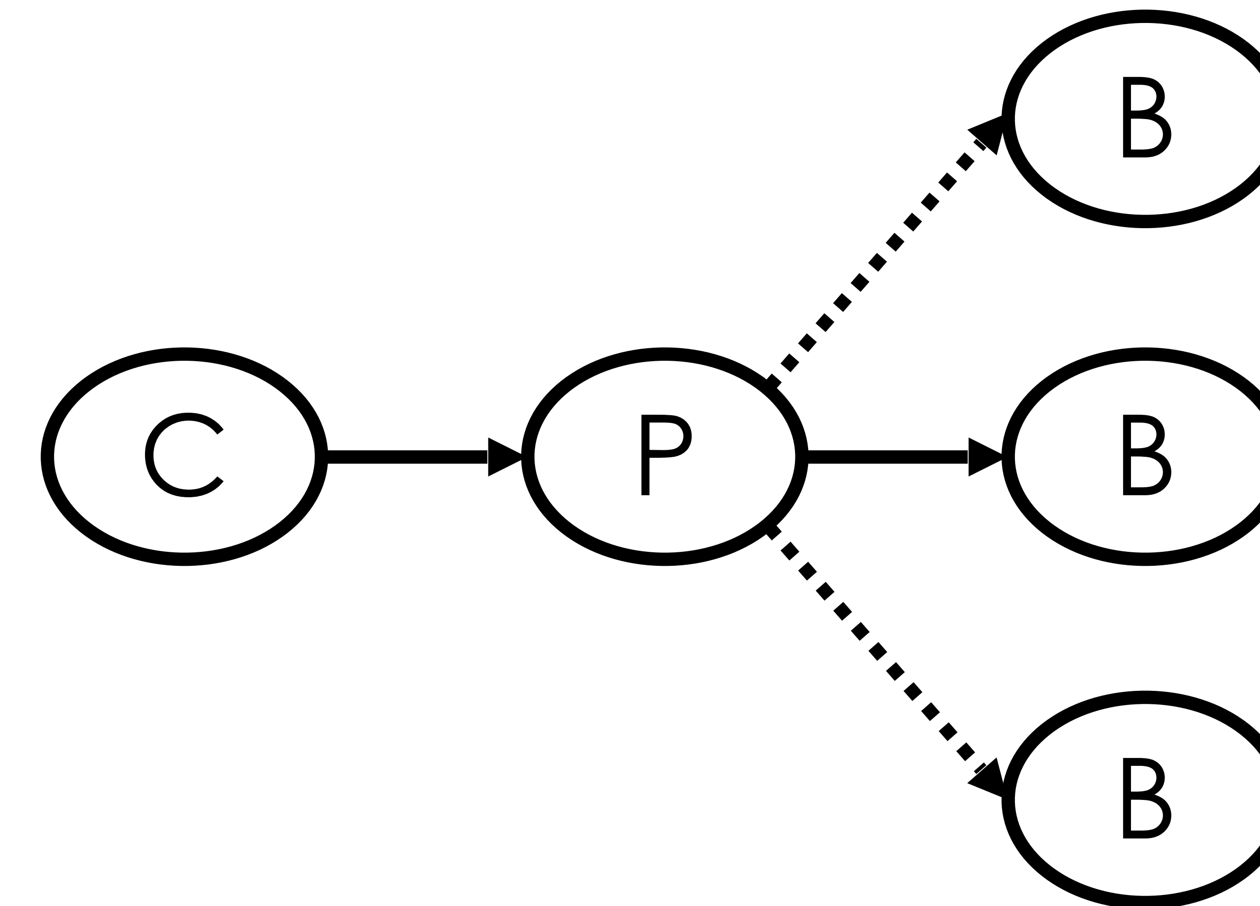
Replikationsmodelle

gleichberechtigte Gruppenbildung



Server organisiert Replikation sofort auf der Basis von Multicast-Kommunikation

„Primary-Backup“-Verfahren



Fehlerfälle:

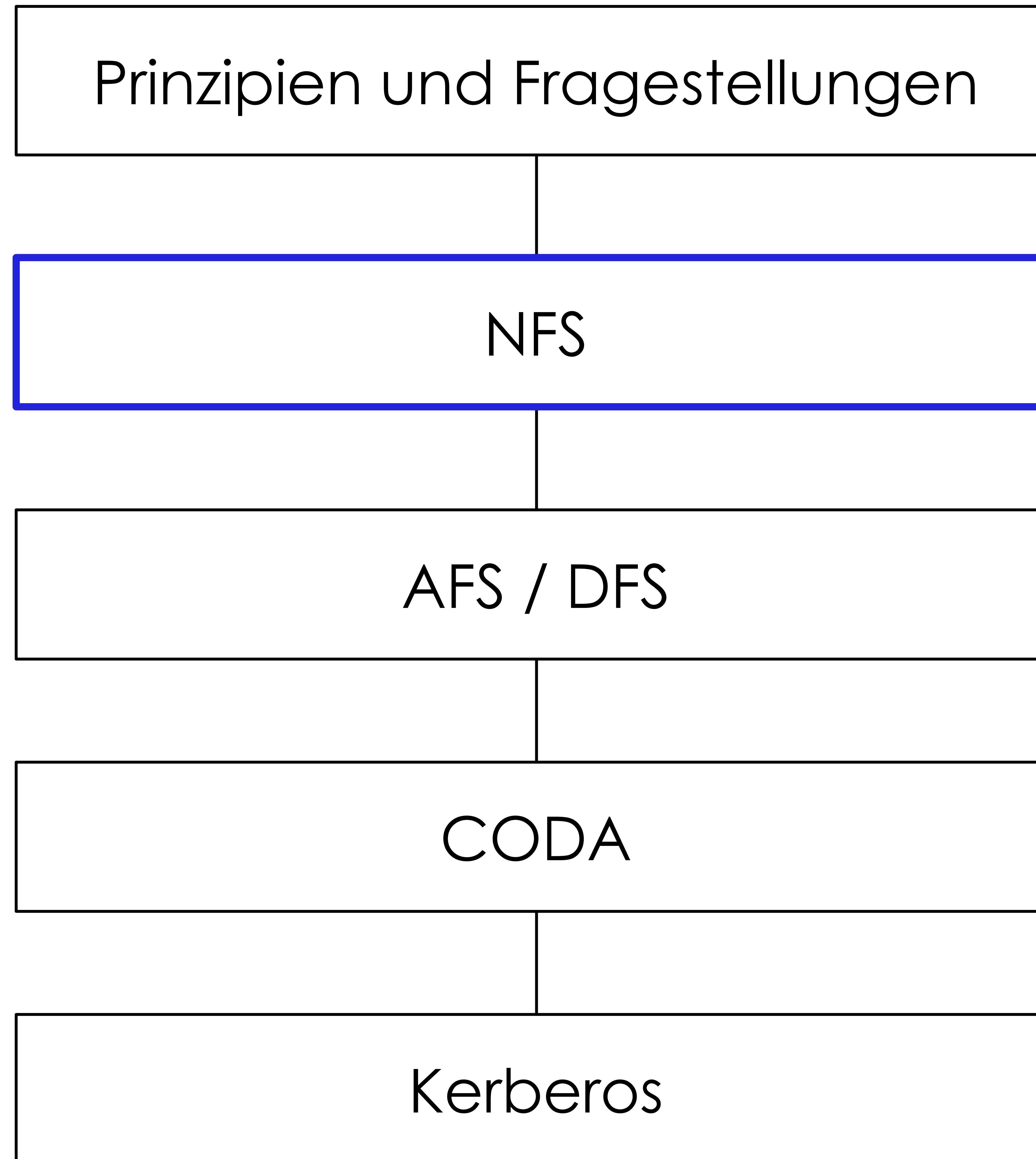
- Gruppenmitglied fällt aus
- Primary fällt aus: Neuwahl

Replikationsmodelle

Replikation: pessimistisch vs. optimistisch

- pessimistisch:
  nur dann schreiben, wenn alle Replikate erreichbar sind
- optimistisch:
  auch schreiben, wenn eine Teilmenge nicht erreichbar ist
  Konfliktauflösung beim Lesen (z.B. Voting, Alter, ...)

Wegweiser



Fallbeispiel 1: NFS

Network File System, UC Berkeley / Sun Microsystems, 1985

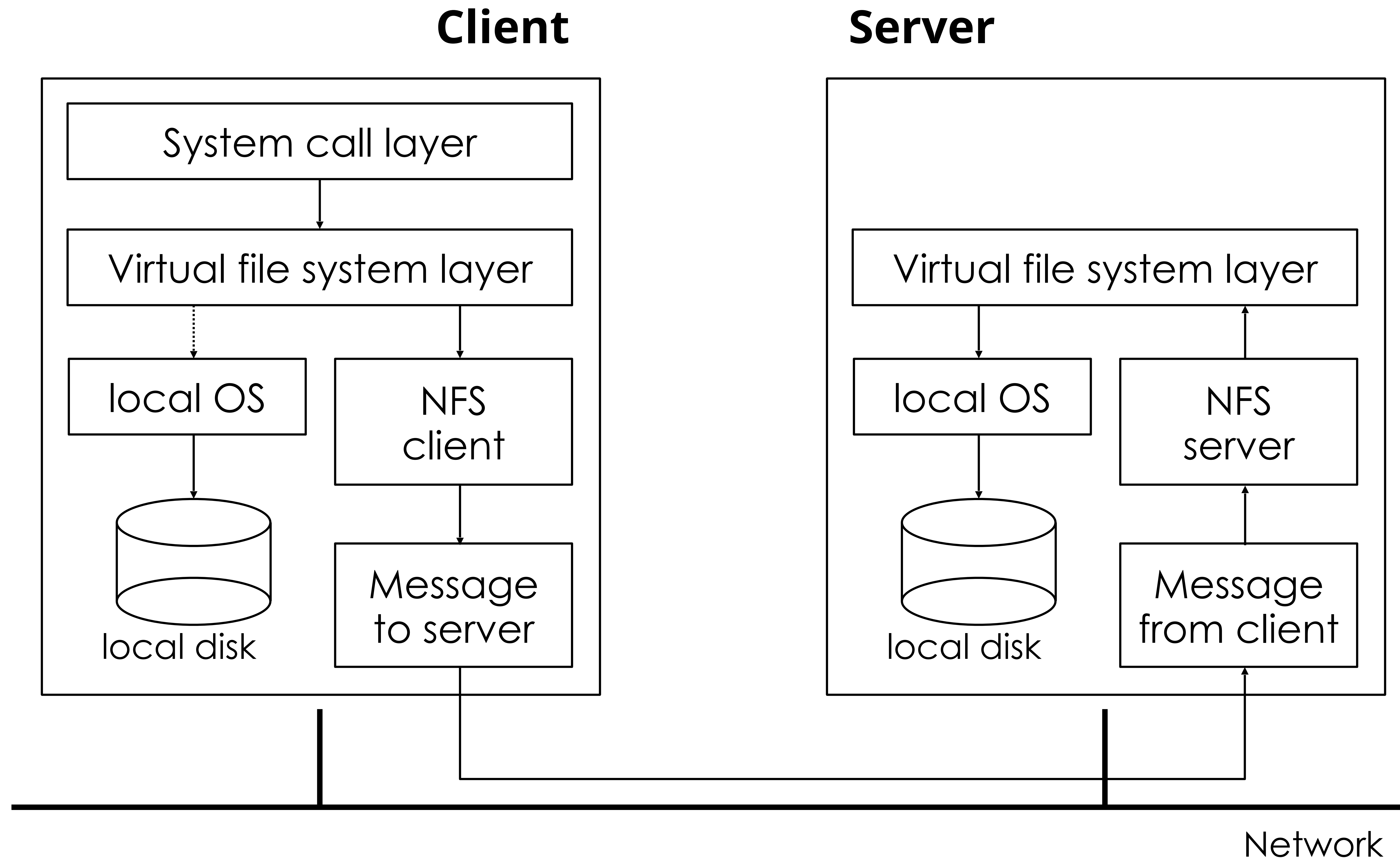
Entwurfsziele

- transparenter Zugriff auf entfernte Dateien
- keine besonderen Bibliotheken (Unix-System-Calls)
- Ortstransparenz
- Heterogenität hinsichtlich Hardware und Betriebssystem

Protokolle

- NFS-Protokoll (Program 100003, Version 2/3)
- Mount-Protokoll (100005, 2/3)
- Network Lock Manager (100021,4)

Unix-Implementierung



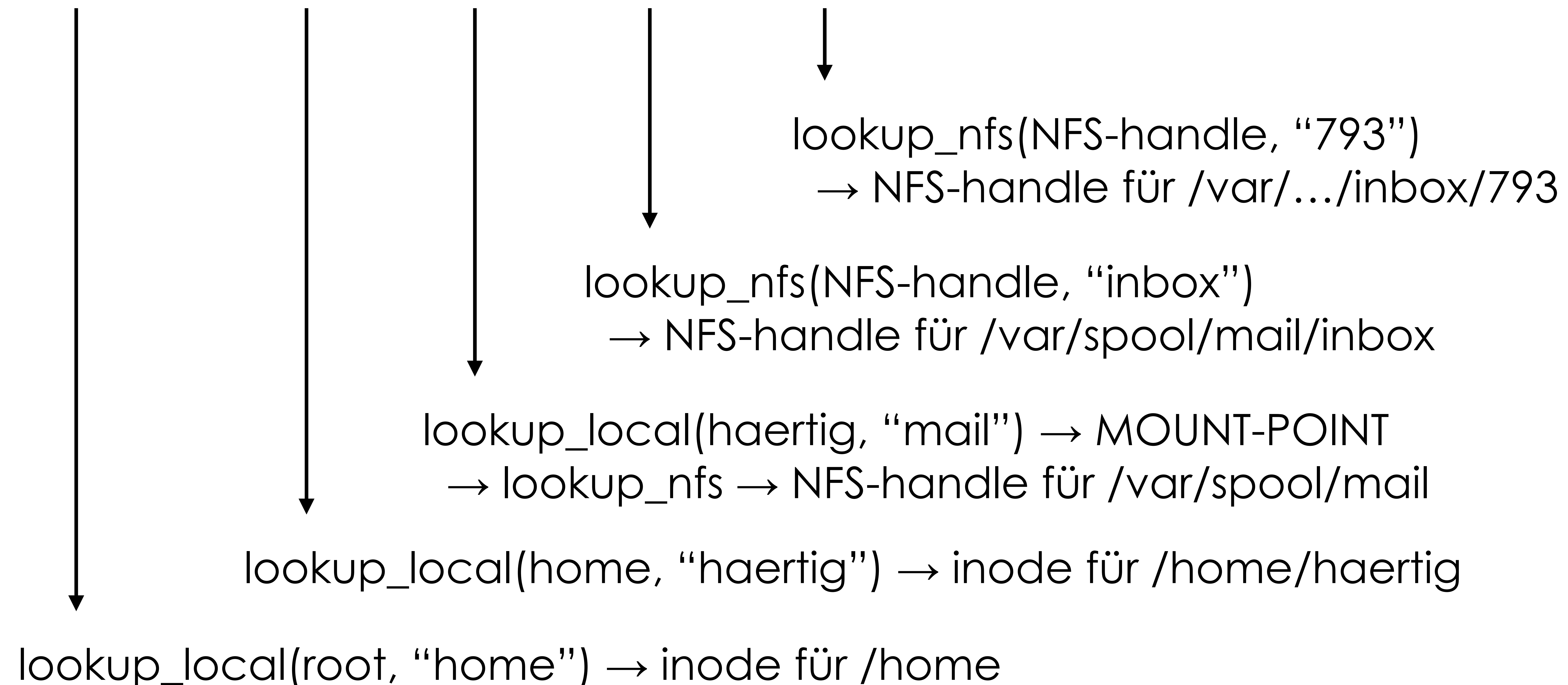
Anwendersicht (API)

- Server: exportiert Dateisysteme (DS)
  - `/etc/exports`: enthält Liste von Dateisystemen
  - Mount-Rechte der Clients: ACL
- Clients: mounten exportiertes Dateisystem
  - danach sichtbar im eigenen Namensraum
  - manche Implementierungen erlauben auch mounten von Teilbäumen
- Zugriff transparent über Pfadnamen
  - Schnittstelle des Clients nicht standardisiert, plattformabhängig

Pfadnamen-Analyse

```
mount -t nfs zih:/var/spool/mail /home/haertig/mail

/home/haertig/mail/inbox/793
```



Unix-Schnittstellen

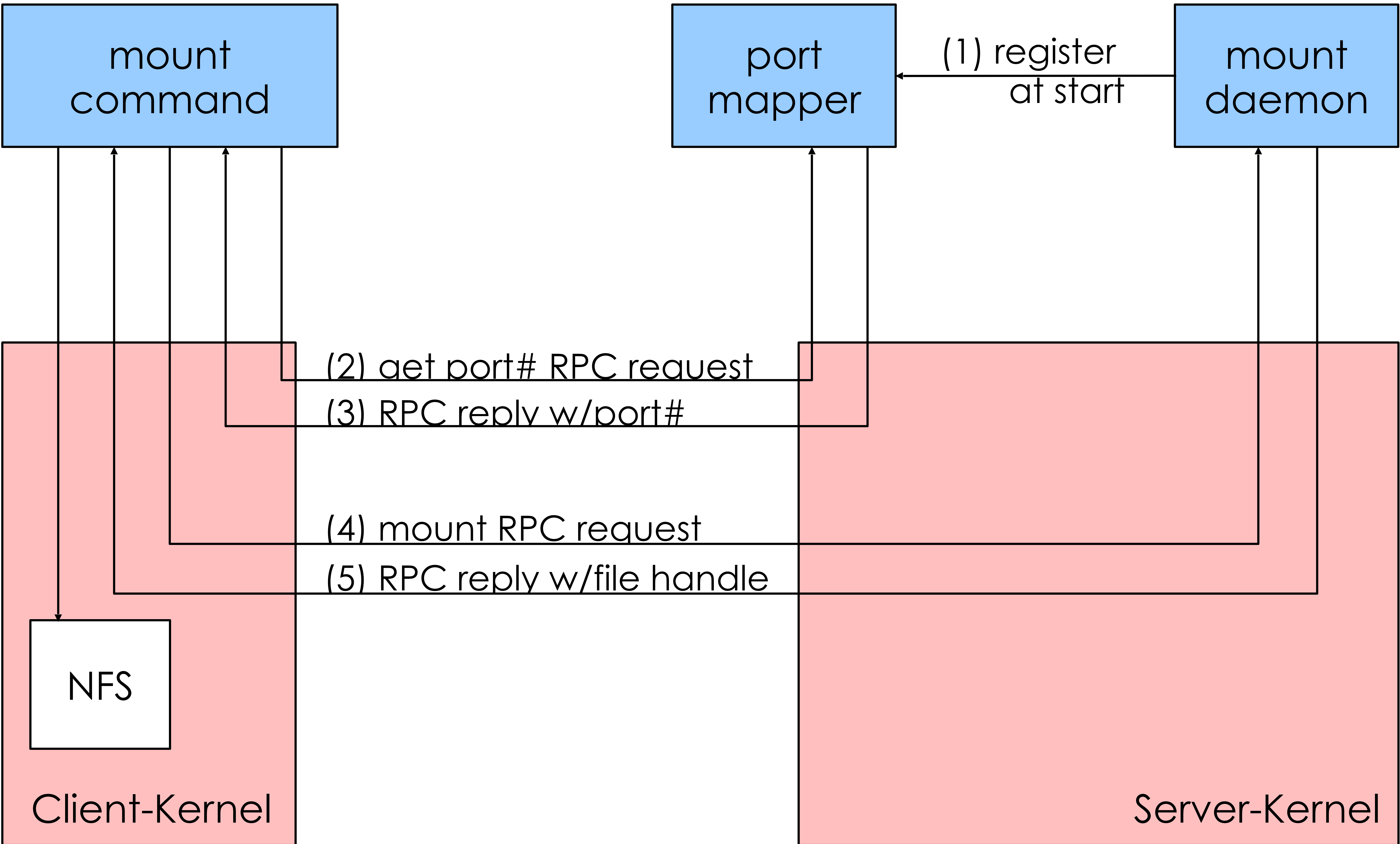
Open

- lokales abwandern des Pfadnamens
- bei jedem Teil-Namen:
  prüfen ob Mount-Punkt erreicht
- bei Erreichen eines NFS-Mount-Punktes:
  NFS-Lookup für jeden weiteren Teilnamen
- am Ende des Pfadnamens:
  File Descriptor wird erzeugt, Verbindung zum NFS-Handle wird vermerkt

Read / Write

- für NFS-Datei: NFS-Handle nachschlagen
- NFS-Protokoll: **`read(`**handle, offset, …**`)`**

Mount-Protokoll



NFS-Protokoll

Dateiattribute

- Type
- Mode
- Eigentümer (UId), Gruppe (GId)
- Number of Links
- File System Id
- File Id
- Atime - letzter Zugriff
- Ctime - letzte Modifikation eines Dateiattributes
- Mtime - letzte Modifikation der Datei

NFS-Protokoll

Auszug (insgesamt 21 Operationen)

- **`lookup(dir_handle, name)`** → (file_handle, attribute)
- **`read(handle, offset, count)`** → (attr, data, count, eof)
- **`write(handle, offset, count, stable)`** → (result, attr, committed)
  - committed: Bestätigung dass Daten auf Platte sind
- **`commit(handle, offset, count)`** → (attr)

Bemerkungen

- idempotent: offset als expliziter Parameter
- stateless: keine Zustandsinformation über offene Dateien
- keine Synchronisationsoperationen (lock)

Unix-Implementierung

Read

- prüft, ob Block in lokalem Cache-Speicher
- falls nicht: Block vom Server lesen
- read ahead möglich
- Inkonsistenz möglich bei mehreren schreibenden Clients

Write

- write through cache: Schreib-Operationen erst dann abgeschlossen, wenn Daten auf Server-Platte geschrieben
- oder: zunächst in Client-Puffer schreiben, mehrere writes sammeln, asynchrones Auslösen der Schreib-Operationen an Server

Cache-Konsistenz

Schwache Konsistenz / Weak Consistency

- Client prüft Cache-Gültigkeit in „freshness“-Intervallen:
  - beim Öffnen einer Datei
  - bei Datenblöcken alle 3 Sekunden
  - bei Verzeichnis-Blöcken alle 30 Sekunden
- alle geänderten Pufferinhalte werden nach 30 Sekunden zum Server geschrieben

Fehlermodell

- Toleriert werden sollen: transiente Fehler, Server-Abstürze
- nicht aber: permanente Fehler auf Speichermedium

Umgang mit Fehlern

- Anwendung blockiert, bis Server wieder verfügbar
- Operationen werden solange wiederholt, bis gelungen
- nutzt „at least once“ RPC-Semantik
- am häufigsten auftretende Operationen sind idempotent

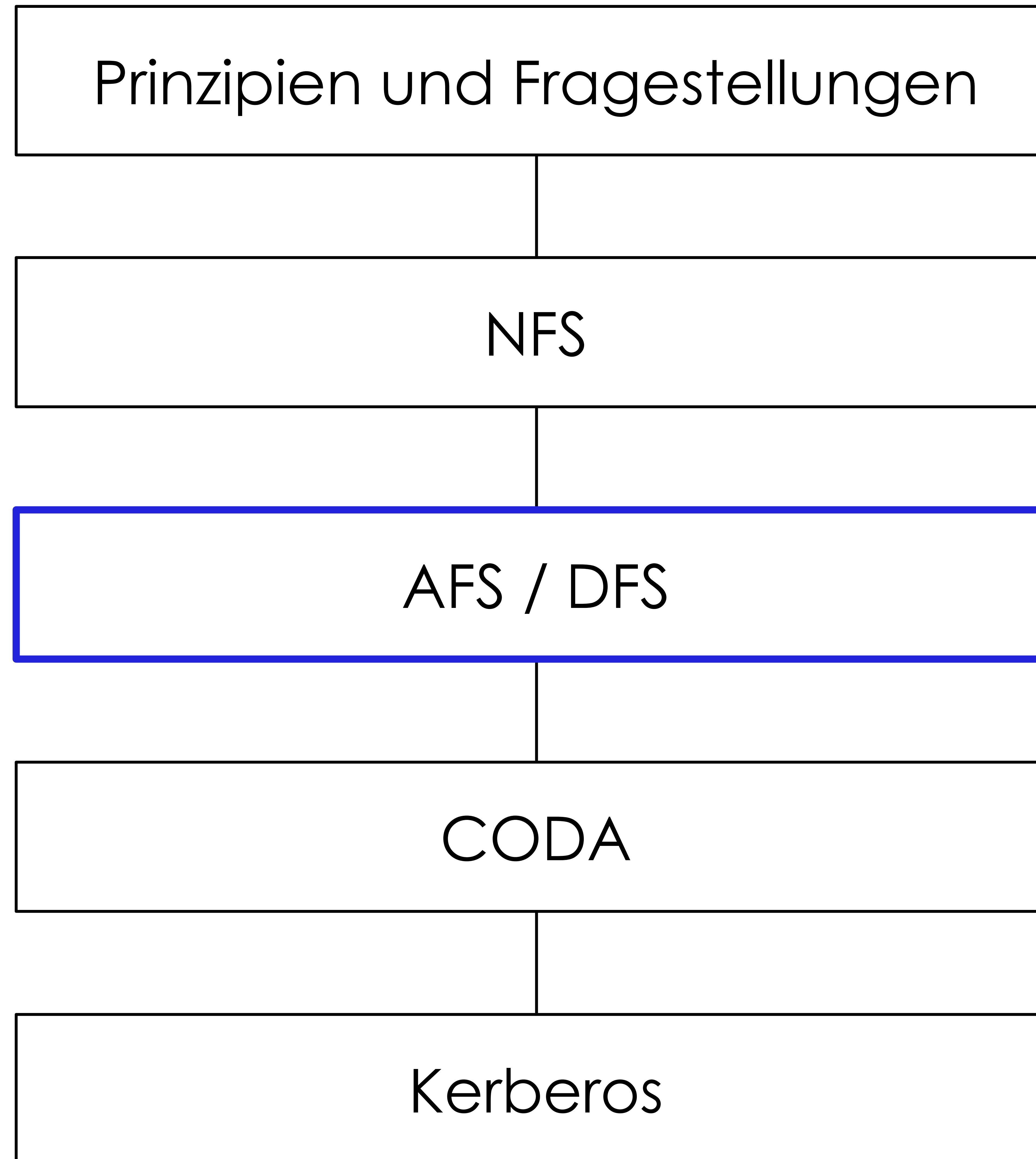
Zugriffsschutz und Authentfikation

- Basiert wie Unix auf Zugriffssteuerlisten (ACL)
- Drei Varianten:
  - UId und GId Teil jeder Botschaft im Klartext
  - UId und GId per DES verschlüsselt
  - Kerberos
- erste zwei Varianten:
  jeder mit Netz-Zugriff oder Root-Rechten auf Arbeitsstation kann Pakete mit „geeigneter“ UId/GId zusammenbauen
- Daten werden grundsätzlich nicht verschlüsselt

Bewertung NFS

- einfache, klare Schnittstelle
- historisch sehr erfolgreich
- Protokoll ist Internet-Standard (RFC 1094)
- auf vielen Betriebssystemen verfügbar
- freie Implementierungen verfügbar
- geringe Skalierbarkeit
- eingeschränkte Leistungsfähigkeit

Wegweiser



Fallbeispiel 2: AFS/DFS

- Andrew File System, Carnegie Mellon University
- Hauptautor: Mahadev Satyanarayanan, ca. 1989
- Weiterentwicklung: DFS

Ziele

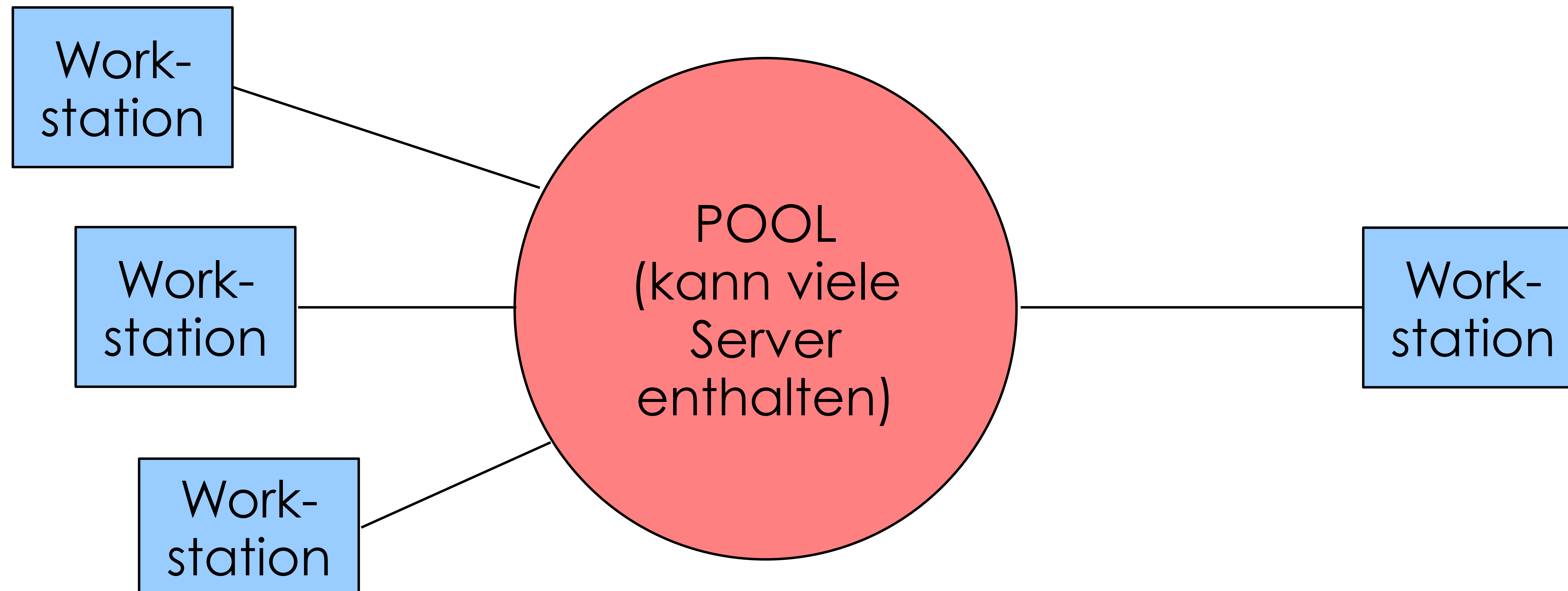
- Skalierbarkeit: 5.000 – 10.000 Arbeitsstationen
- einfache Administration

Annahmen / Beobachtungen

Beobachtungen (ca. 1990)

- Dateien sind in der Regel klein
- Leseoperationen sind häufiger als Schreiboperationen
- Sequentieller Zugriff ist häufiger als direkter
- Die meisten Dateien haben nur einen Benutzer
- Bei mehreren Nutzern schreibt meist nur einer
- Dateizugriffe erfolgen in Bursts

Architektur von AFS



Prinzipien

- ganze Dateien in Caches der Clients
- Client-Caches sind dauerhaft (auf lokaler Platte)
- sehr große Dateien werden nicht berücksichtigt

Skalierbarkeit

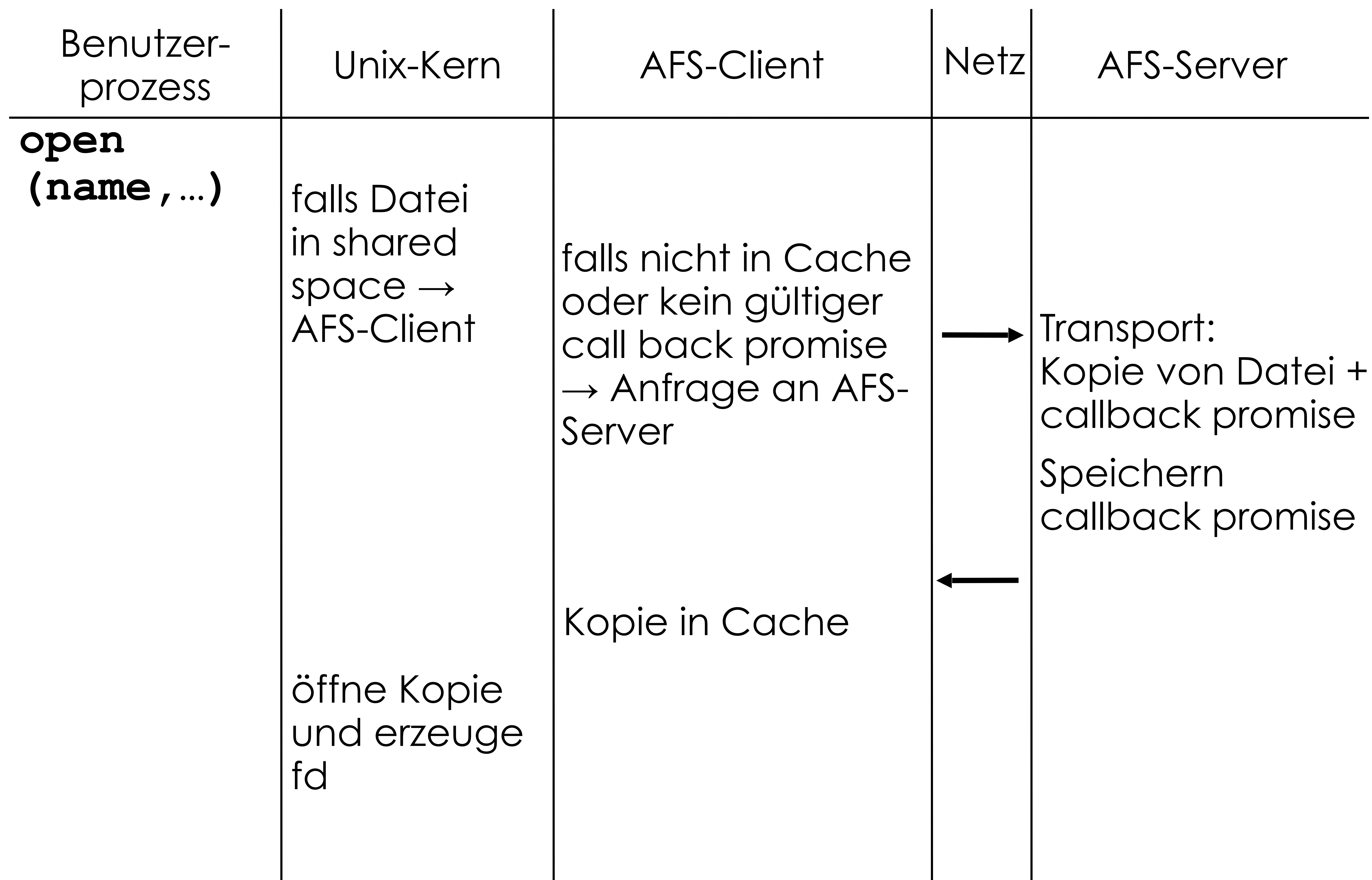
Replikation

- auf allen Servern
- Lesen von beliebigen Replikaten
- Schreiben über Master

Caching

- ganze Dateien
- Hauptspeicher und Platte des Clients als Cache
- enthalten “Working Set” von Dateien
- Tausch nach LRU

Protokoll



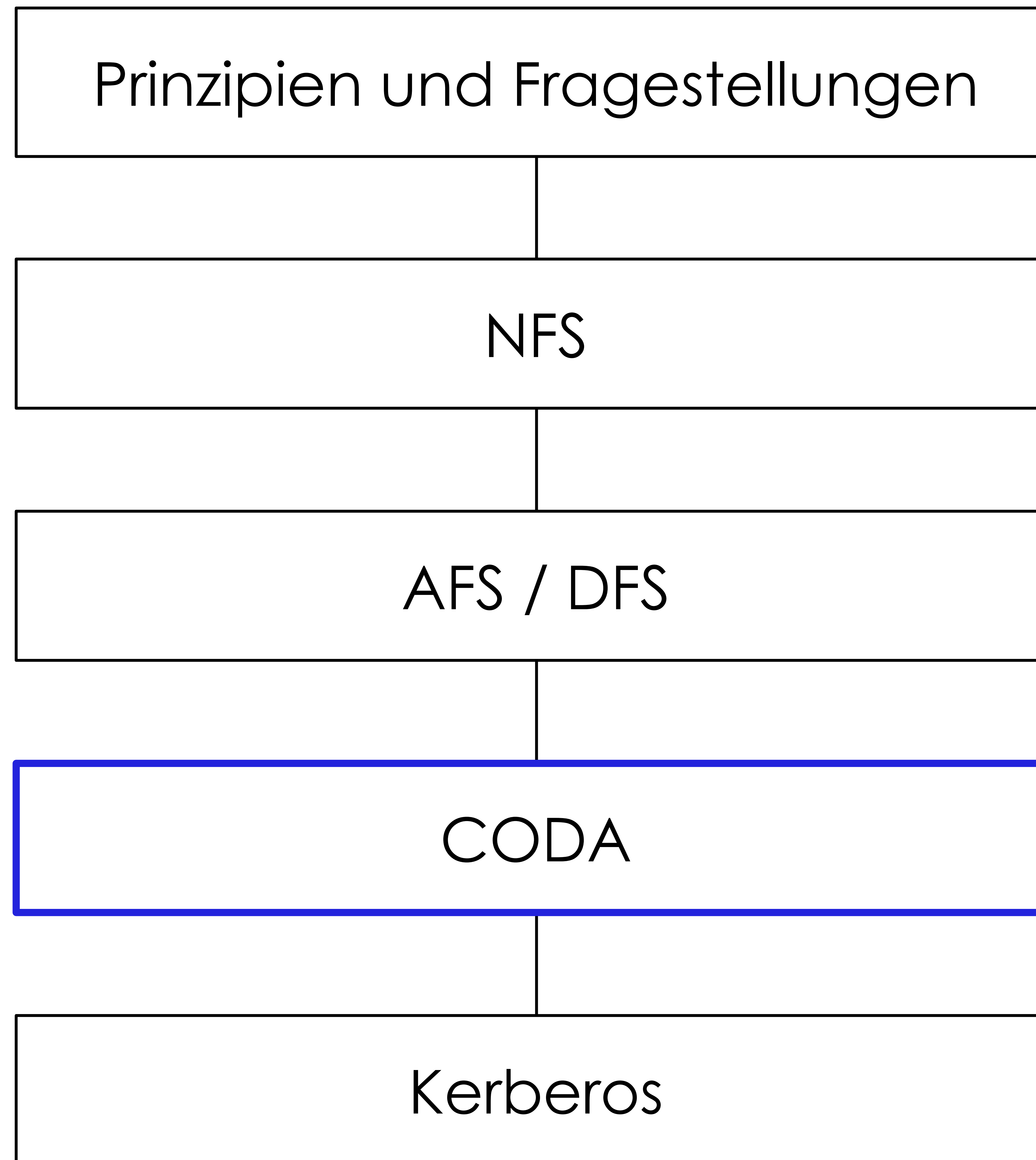
Protokoll

Benutzer-prozess	Unix-Kern	AFS-Client	Netz	AFS-Server
`close(fd)`	schließen lokale Kopie	falls lokale Kopie geändert → Kopie transportieren an AFS-Server	→	ersetze Datei, sende callback an alle Clients, die callback promise registriert haben
`read/write`	normaler Unix-Syscall auf Kopie			

Konsistenz

- bei Übertragung einer Datei von Server an Client wird „callback promise“ mitgegeben
- bei Änderung an einer Datei auf Server:
  - callback-RPC an alle registrierten Clients
  - Wirkung: Cache wird verworfen
- nach Neustart eines Clients:
  - Cache Validation Request an Server mit Modifikations-Zeitstempel
  - falls ungültig: neue Kopie holen
- Callback wird bei **`open`** erneuert, falls seit Zeit T keine Kommunikation mit Server stattfand (T = einige Minuten)

Wegweiser



Fallbeispiel 3: CODA

- Carnegie Mellon University
- Satyanarayanan, Kistler 1992
- Weiterentwicklung von AFS

Ziele

- Erhöhung der Verfügbarkeit auch bei Netz-Partitionierung
- Mobile Computing als Spezialfall von Netz-Partition

Techniken

- Replikation
- abgekoppelter Betrieb (Disconnected Operation)

Replikation

- Volumes werden repliziert
  VSG: Volume Storage Group
  Menge der Server, die Replikate enthalten
  AVSG : available VSG
  AVSG $\subseteq$ VSG
  AVSG = $\emptyset$: disconnected operation
- open-Operationen:
  - ein Server des AVSG liefert Datei-Inhalt
  - alle Server werden wg. Status-Info angefragt
- close-Operationen:
  - multicast geänderter Dateien an alle Server aus AVSG

Replikation

Optimistisches Replikationsverfahren

- Modifikation werden in jedem Fall bei den Servern der AVSG ausgeführt
- Beseitigung von Inkonsistenzen bei Feststellung durch Klienten nach Aufhebung der Partition
- wenn möglich, automatisch, sonst manuell

Replikation

Coda Version Vector (CVV)

- wird für jedes Replikat einer Datei verwaltet
- $CVV_i := (S_{i1}, \ldots, S_{in})$
  S_{ik}: Anzahl der Modifikationen von Replikat i auf Server k
- CVV_i dominiert $CVV_j \Leftrightarrow \forall\, k = 1 \ldots n : S_{ik} \geq S_{jk}$ (ist aktueller als)
- bei jeder Modifikation (close-Operation) durch Server k:
  $S_{ik} := S_{ik} + 1$ für alle $i \in$ AVSG
- beim Lesen (lazy) kann eine Inkonsistenz automatisch repariert werden, wenn:
- $\exists\, i \in$ AVSG so, dass gilt: $\forall j$: CVV_i dominiert CVV_j
  (ist aktueller als alle anderen)

Beispiel

Datei F

VSG = { S1, S2, S3 }

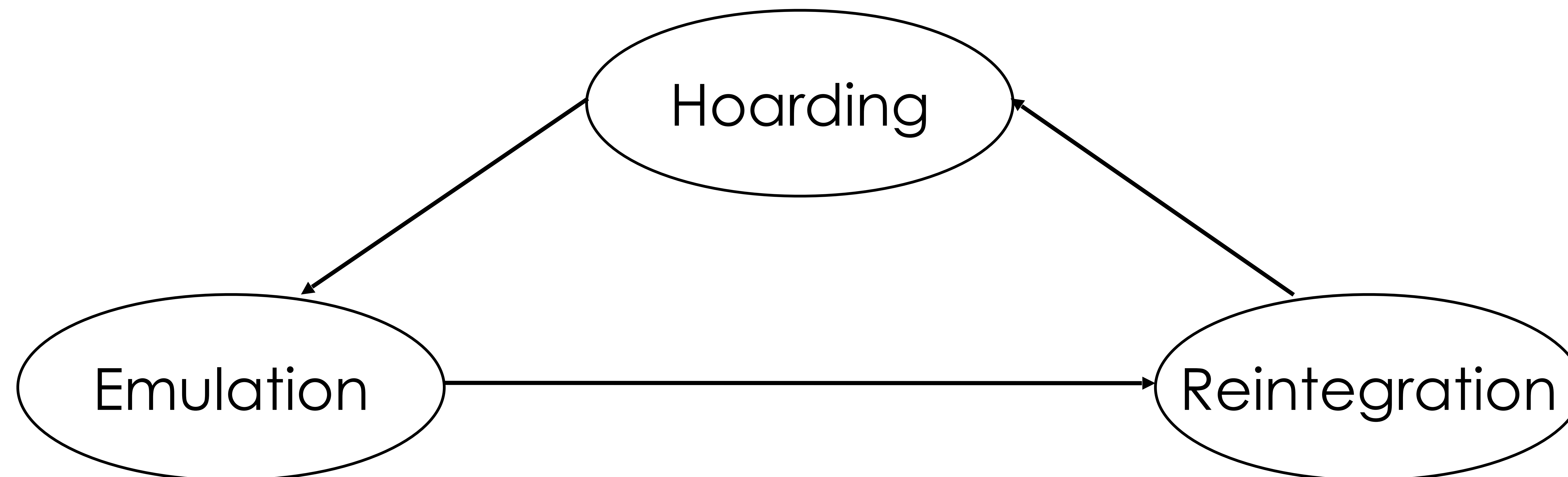
Klienten: C1, C2

Beispiel

Ereignis	AVSG	CVV_1	CVV_2	CVV_3
start	{S1, S2, S3}	1, 1, 1	1, 1, 1	1, 1, 1
C1 : öffnet, modifiziert schließt		2, 2, 2	2, 2, 2	2, 2, 2
S3 : fällt aus	{S1, S2}			
C1 : modifiziert		3, 3, 2	3, 3, 2	2, 2, 2
S3 : Neustart	{S1, S2, S3}			
C1 : liest		3, 3, 2	3, 3, 2	3, 3, 2
S3 : fällt aus	{S1, S2}			
C1 : modifiziert		4, 4, 2	4, 4, 2	3, 3, 2
S3 : Neustart	{S1, S2, S3}			
S1, S2 : fallen aus	{ S3 }			
C2 : modifiziert		4, 4, 2	4, 4, 2	3, 3, 3
S1 : Neustart	{S1, S3}			
C1 : liest				

Abgekoppelter Betrieb

- AVSG = 0, z. B. Mobile Computing
- Problem: LRU-Verfahren für Datei-Caching funktioniert nicht gut bei längeren Abkoppelungen
- Benutzer können Dateien konfigurieren (hoard database)
- Werkzeug: ermittelt Dateien für eine Anwendung



Abgekoppelter Betrieb

Phasen des Betriebes

- modifizierte Dateien aufheben bis zur Reintegrationsphase
- “replay-log”
- Reintegration: führt Aktion aus replay log aus

Bewertung

- wenig genutzt
- sehr seltenes Auftreten nicht automatisch behebbarer Konflikte

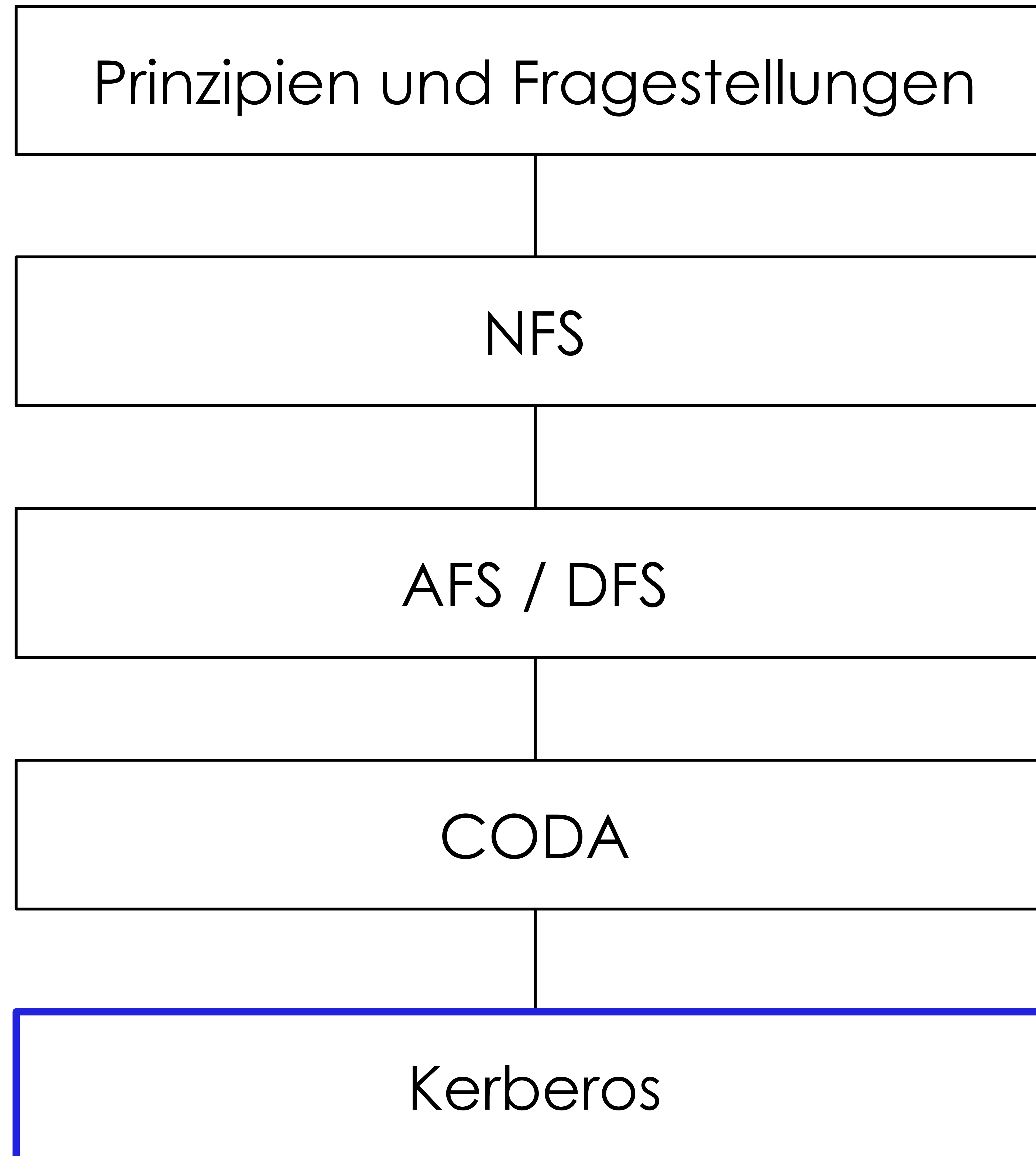
Zusammenfassung

- Beispiel für Anwendung von RPC in verteilten Systemen
- Entwicklung: NFS – AFS – Coda ...
- praktisch werden verwendet: NFS, CIFS

Aktuelle verteilte Speichersysteme

- Abtrennung der Metadaten
- hohe Verfügbarkeit und Skalierbarkeit durch Replikation
- Inhalts- oder Abfrage-basierte Adressierung
- Einschränkung der Konsistenz (eventual consistency)
- Konfliktbehebung durch die Anwendung

Wegweiser



Was bisher geschah …

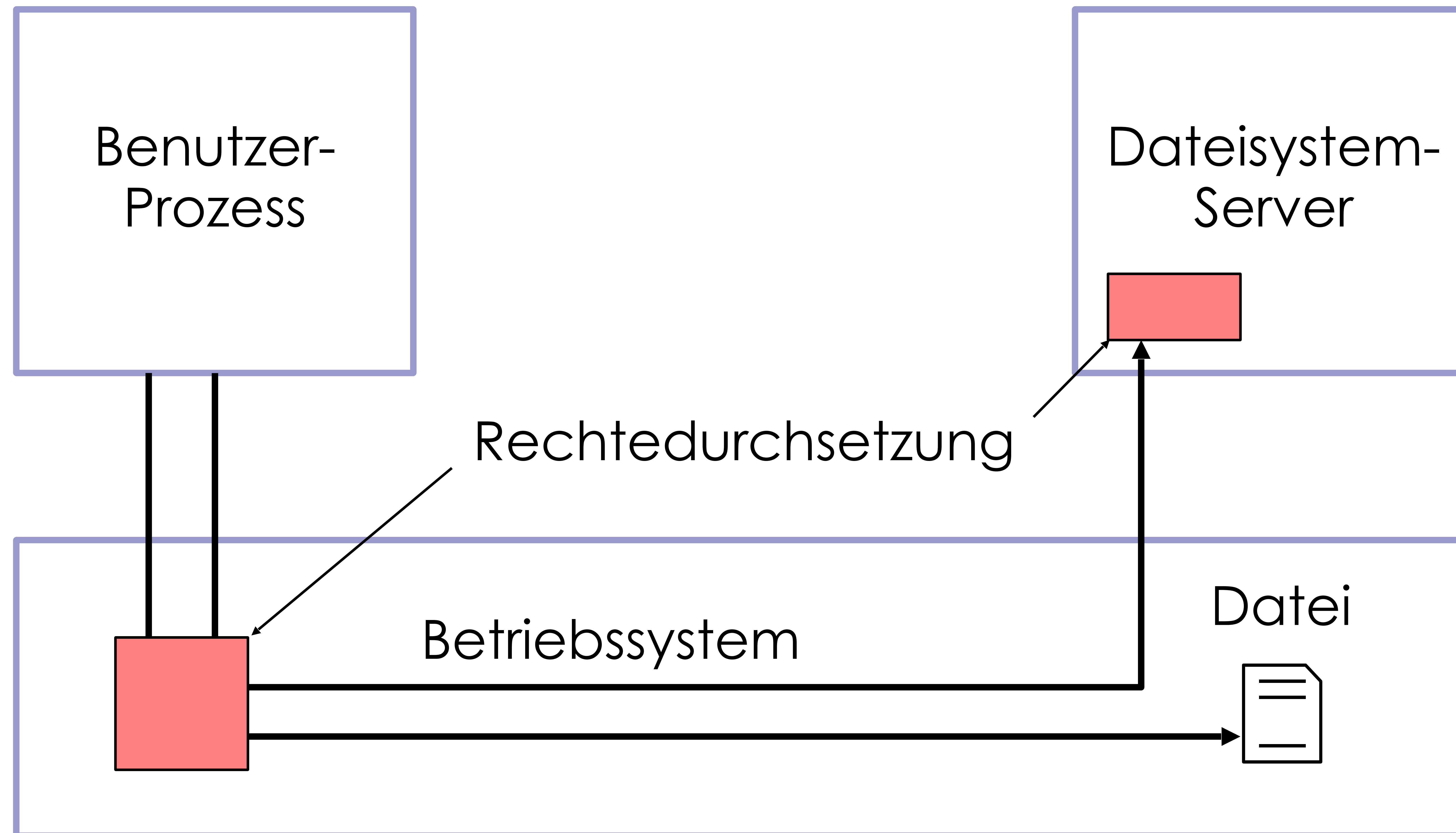
Schutzziele

- Vertraulichkeit (Confidentiality)
- Integrität (Integrity)
- Verfügbarkeit (Availability)
- Wiederherstellbarkeit (Recoverability)

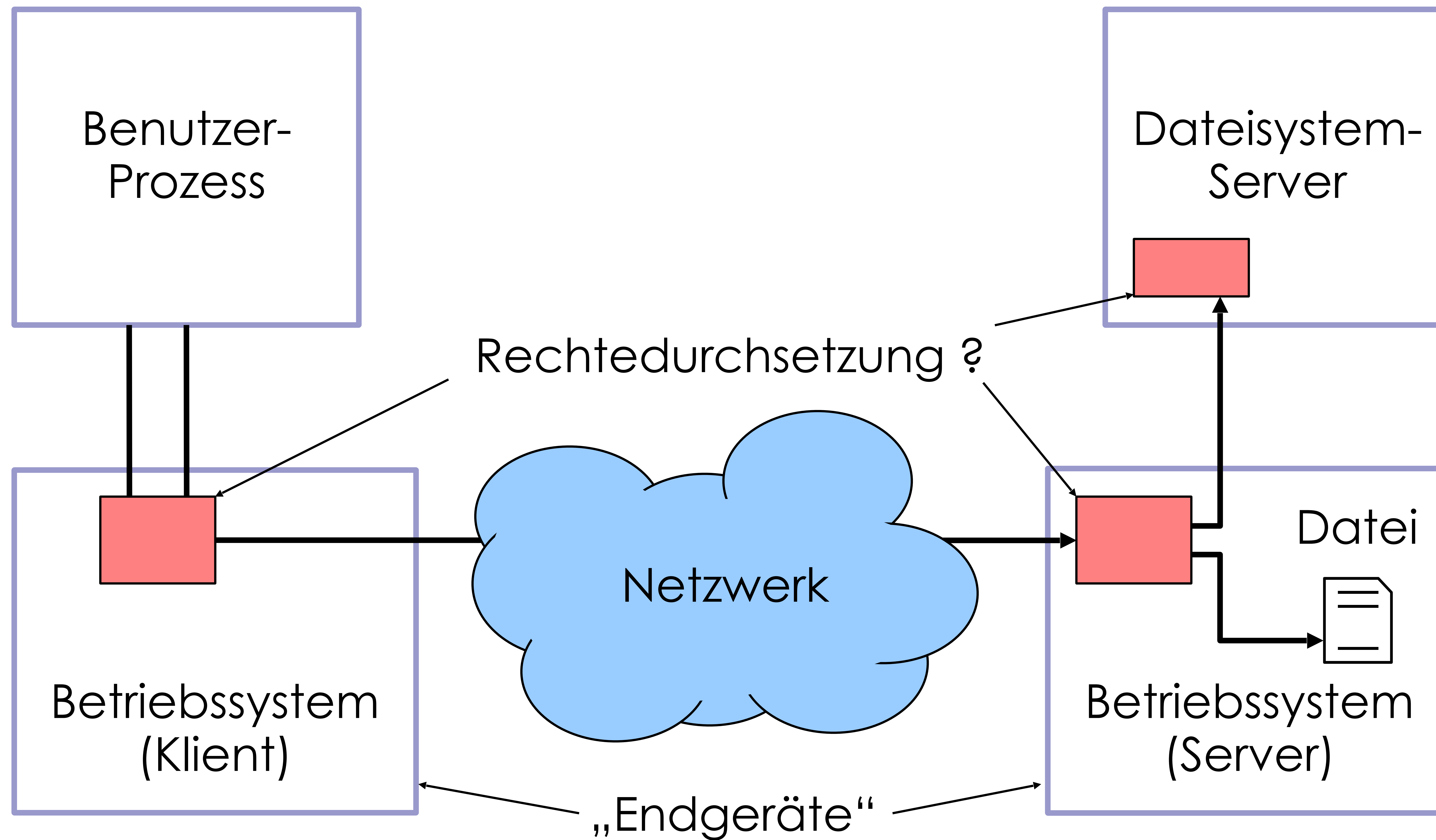
Grundlagen und Mechanismen

- Prozesse als Repräsentanten von Benutzenden
- Isolation (Adressräume)
- Schutzmechanismen (ACL, Capabilities, Sandboxing)
- Authentifikation von Benutzern (Login)

Lokale Lösung



Verteilte Lösung



Bedrohungen

Ein Angreifer kann ...

- in Kommunikationspfade eindringen
- Nachricht abhören, ändern, hinzufügen, löschen
- abgehörte Nachricht später wiederholen
- mehrere Nachrichten vertauschen
- Quell- oder Zieladresse einer Nachricht verändern
- seinen Rechner so modifizieren, dass er sich für einen anderen Rechner mit einer anderen Adresse ausgibt

Einfache Anwendungen von Kryptoverfahren

Vertraulichkeit

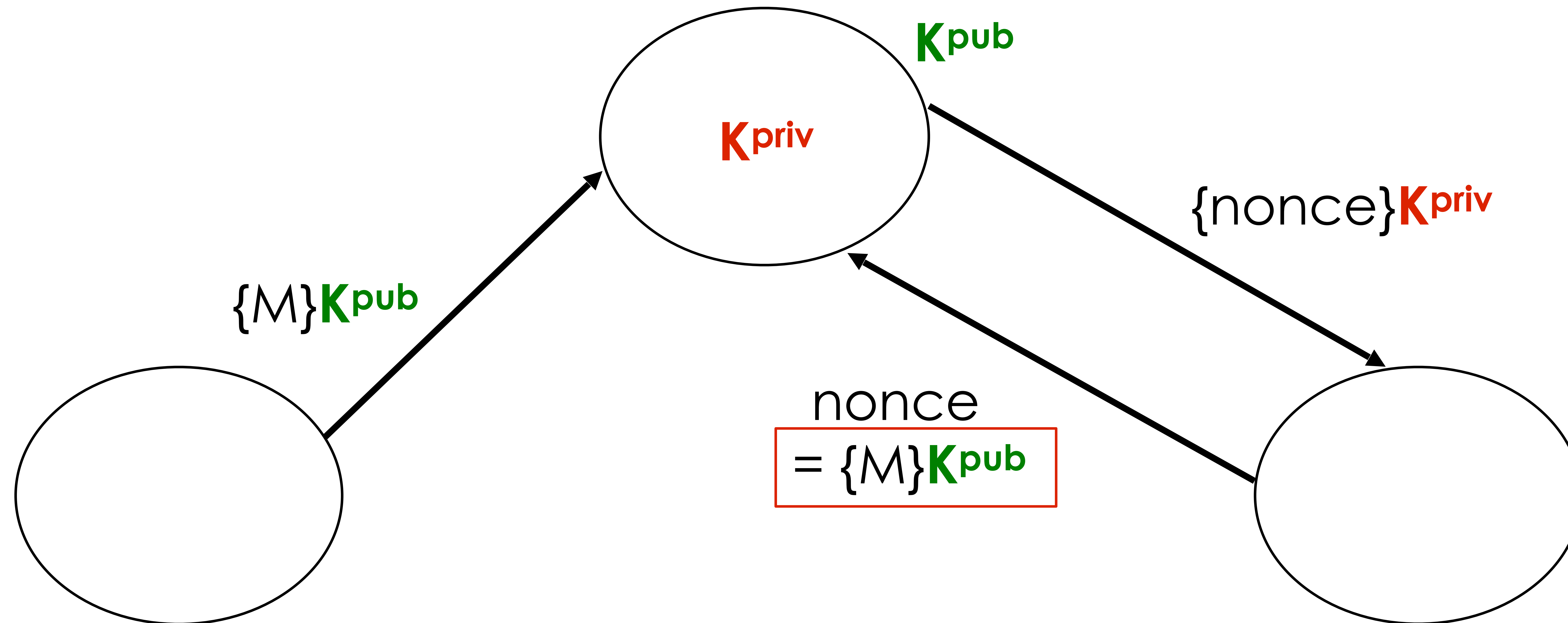
Sender: {Botschaft} K^{pub}
Empfänger: {{Botschaft} K^{pub} } K^{priv}

Feststellung einer Identität

Sender: nonce (challenge)
Antwort von A: {nonce, t} K^{priv}
Sender: {{nonce, t} K^{priv} } K^{pub} enthält nonce

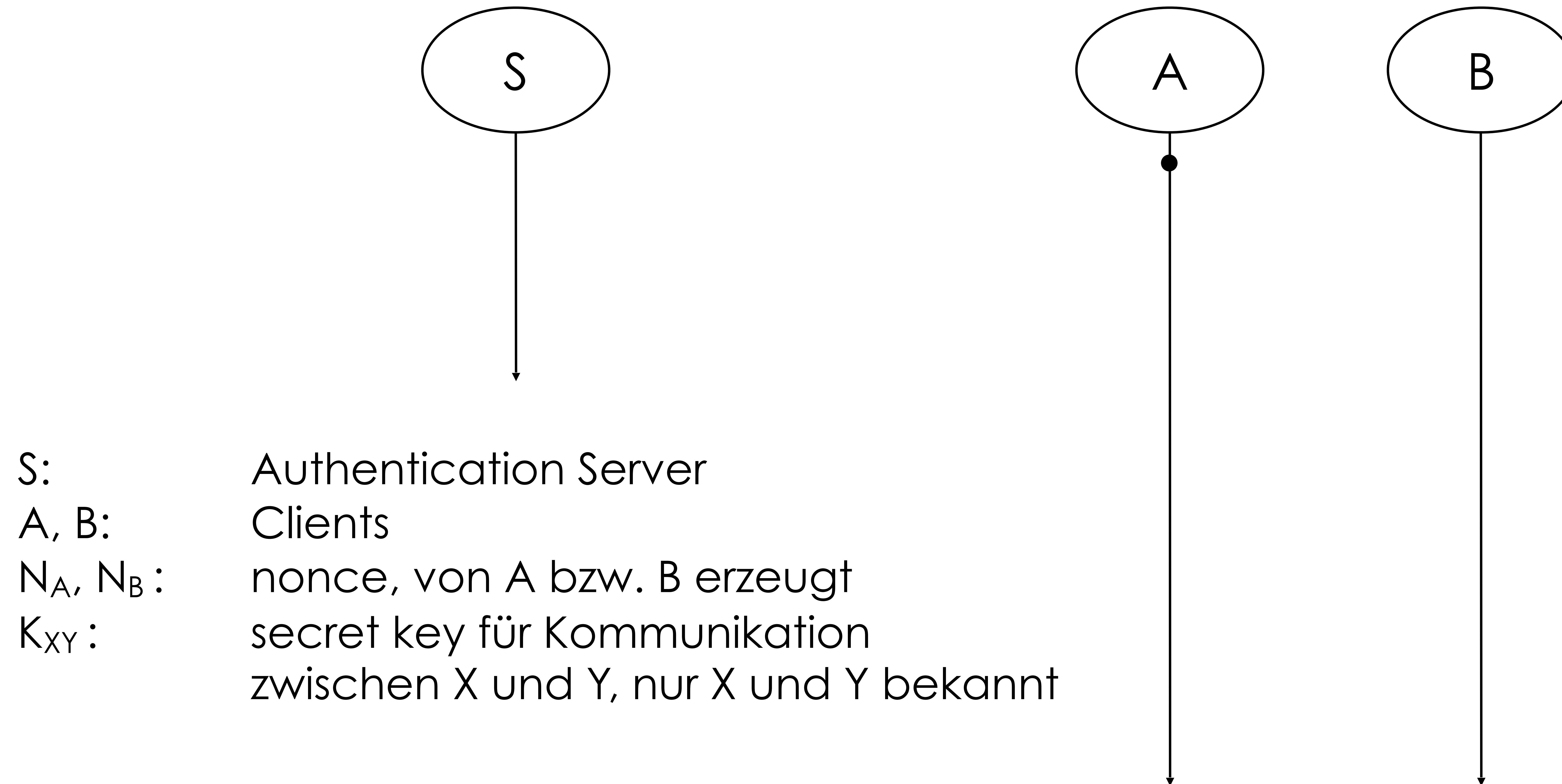
Voraussetzung:
Sender muss wissen, dass K^{pub} zu A gehört

Beispiel

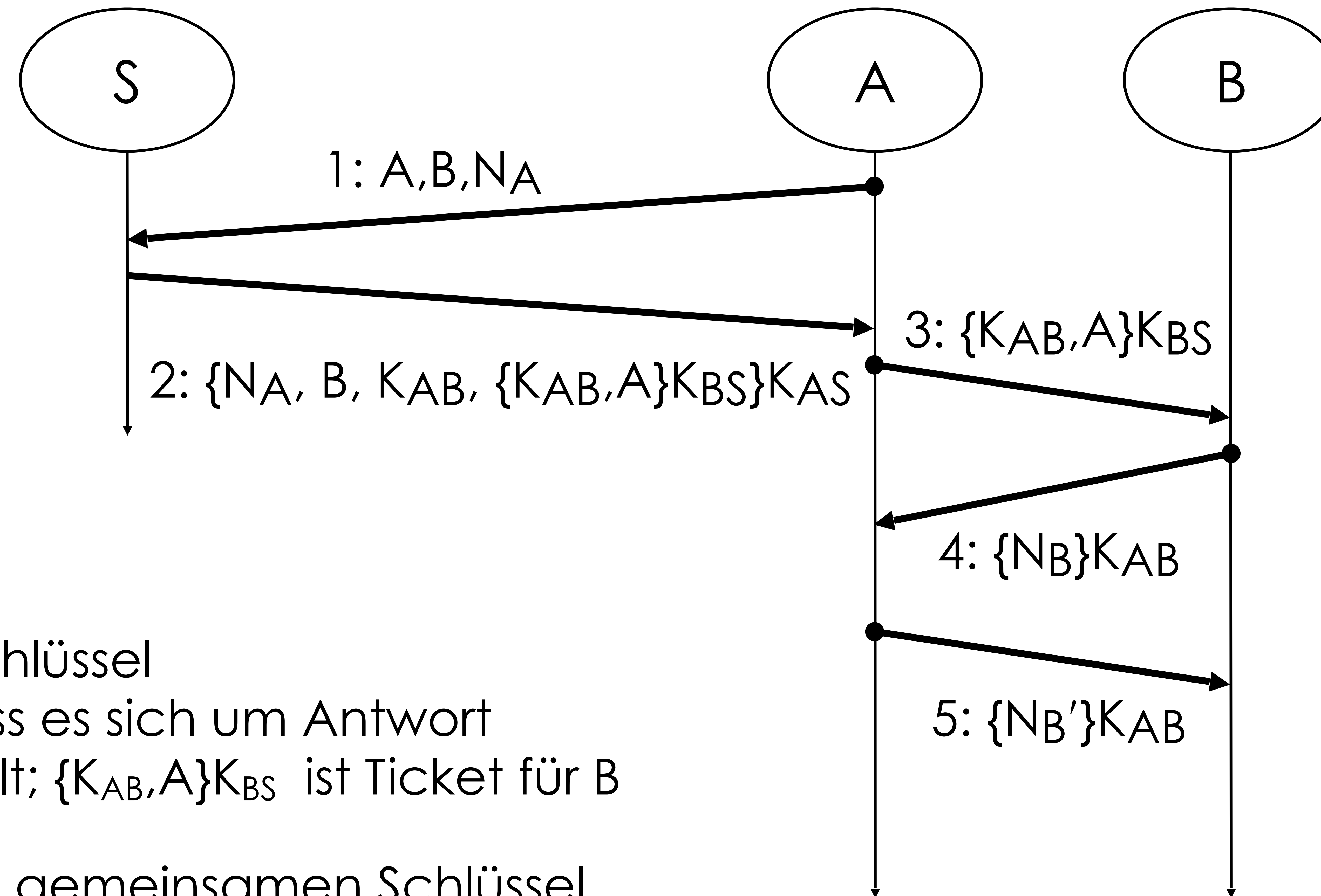


Falsch!

Symmetric-Key Needham–Schroeder



Symmetric-Key Needham–Schroeder



1: key request

2: K_{AB} neuer Schlüssel
N_A zeigt, dass es sich um Antwort auf 1 handelt; $\{K_{AB},A\}K_{BS}$ ist Ticket für B

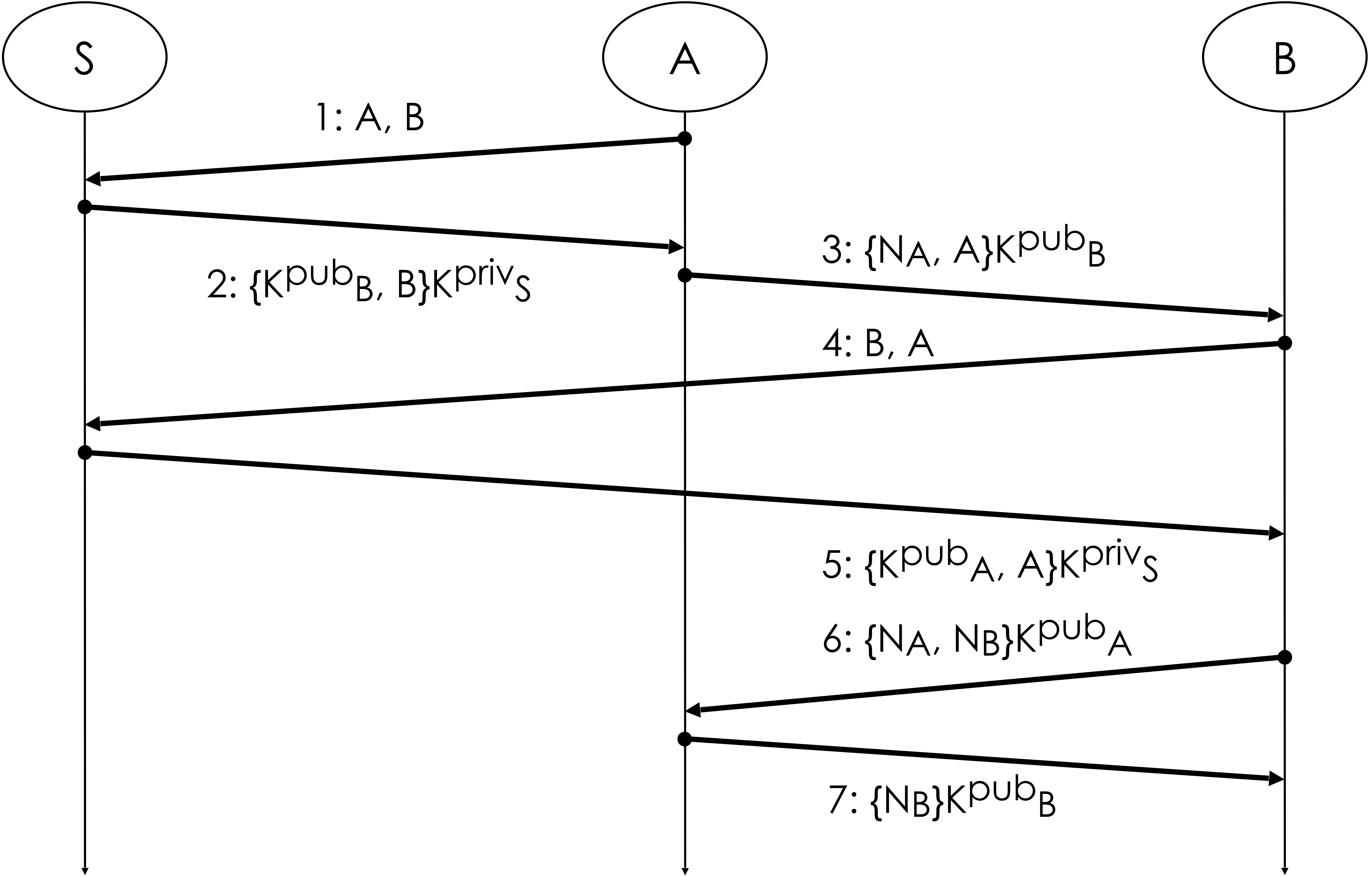
3: A gibt B den gemeinsamen Schlüssel

4/5: durch $\{N_B'\}$ (verändertes nonce) zeigt A Kenntnis von K_{AB}

Public-Key Needham–Schroeder

- public-key-basiertes Verfahren
- Prinzipal A, B
  z.B. Benutzer, Dienste
  „Einheit“, der Rechte zugestanden werden
- Authentication Server S
  kennt *öffentliche* Schlüssel der Prinzipale
- $\mathbf{K^{priv}_X}$ privater Schlüssel von X
  $\mathbf{K^{pub}_X}$ öffentlicher Schlüssel von X
  N_X nonce, von X erzeugt

Public-Key Needham–Schroeder



Protokoll (public key NS)

1: A will public key von B

2: jeder kann KpubB erfahren mittels KpubS

3: nur B kann A und NA herausfinden

4,5: B besorgt sich A's public key von S

6,7: durch Bestätigung der nonces wird Aktualität gezeigt

Kerberos

Integration des Needham-Schroeder-Protokolls in Client-Server-Architektur als Netzwerk-Authentifizierungsservice

Annahmen

- offenes (unsicheres) Netz, Abhör- und Manipulierbarkeit
- kein Vertrauen in Rechnernamen, wechselnde Nutzer
- Passwort kurzfristig sicher eingebbar
- Passwort langfristig aber nicht sicher speicherbar
- Uhren synchronisiert

Konsequenzen für Passwörter

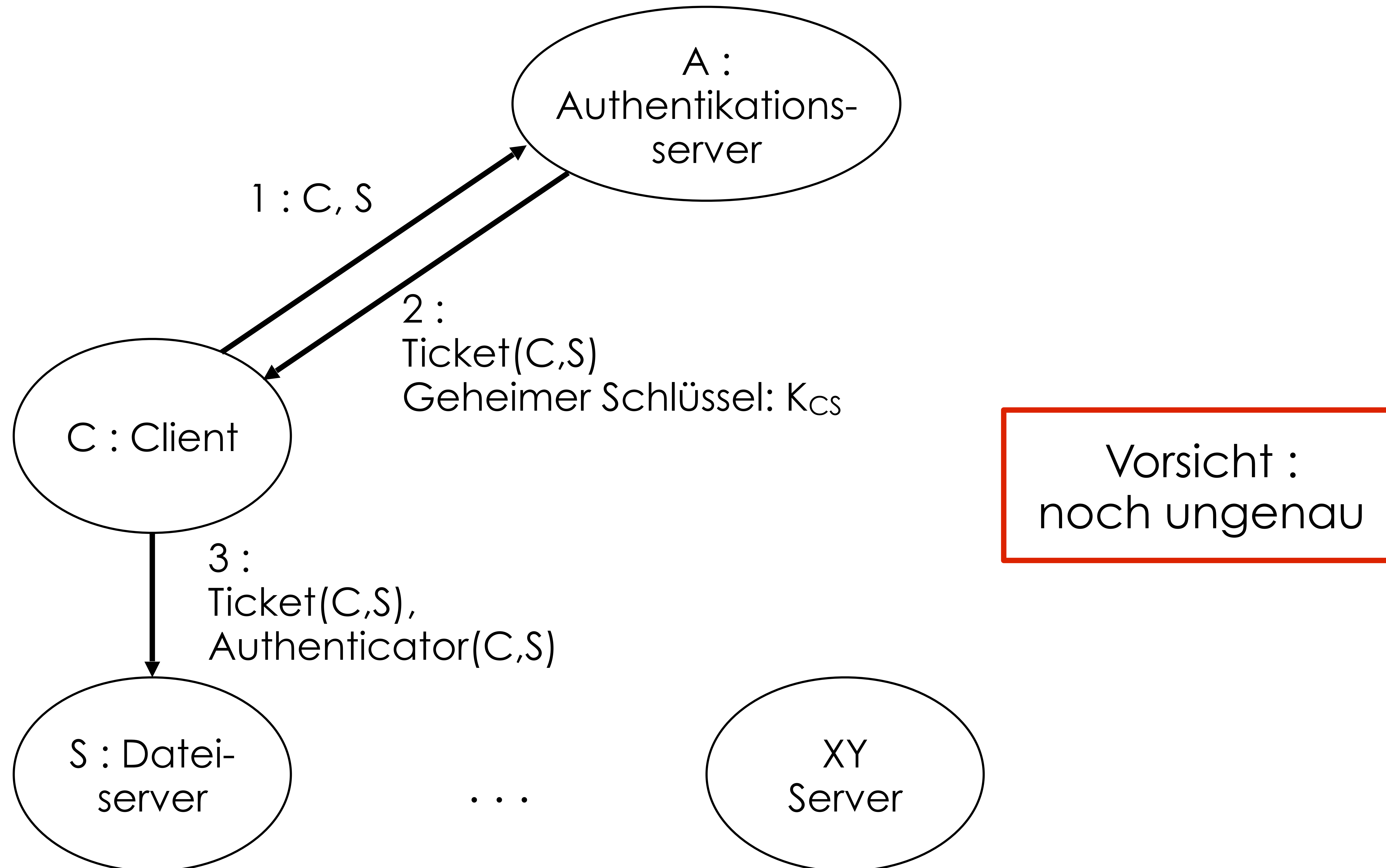
- werden nicht im Klartext über Netzwerk gesendet
- auf Clients nur sehr kurz in Gebrauch

Kerberos

Wesentliche Elemente

- Authenticator
  - beweist Identität des Absenders
  - enthält Zeitstempel jedoch kein Passwort
  - einmalig gültig
- Ticket
  - für ein Client-Server-Paar; beweist Authentifikation
  - beschränkte Gültigkeit
  - enthält Sitzungsschlüssel
- Sitzungsschlüssel
  - geheimer Schlüssel für Kommunikation mit Servern

Struktur



Login ohne Passwort auf Netzwerk

Voraussetzung

- Funktion f : Passwort → geheimer Schlüssel (z.B. PBKDF2)
- A kennt geheime Schlüssel aller Prinzipale

Protokoll

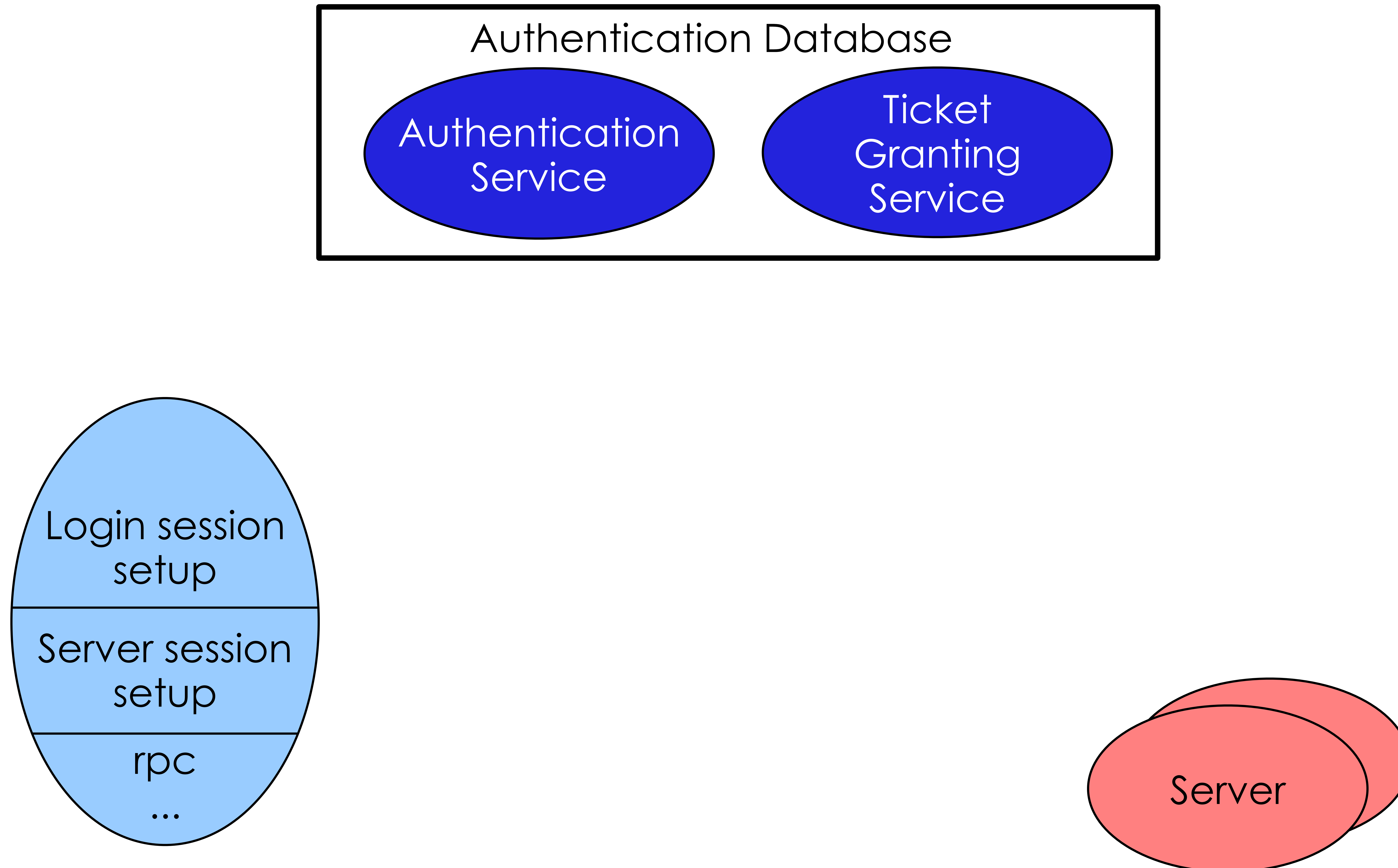
C → A : C, S, time, nonce
A holt K_C und erzeugt Session-Key K_{CS}

A → C : { K_{CS}, nonce, … }K_C + Ticket
C fragt Benutzer nach Passwort, berechnet K_C
packt K_{CS} aus und löscht Passwort & K_C wieder

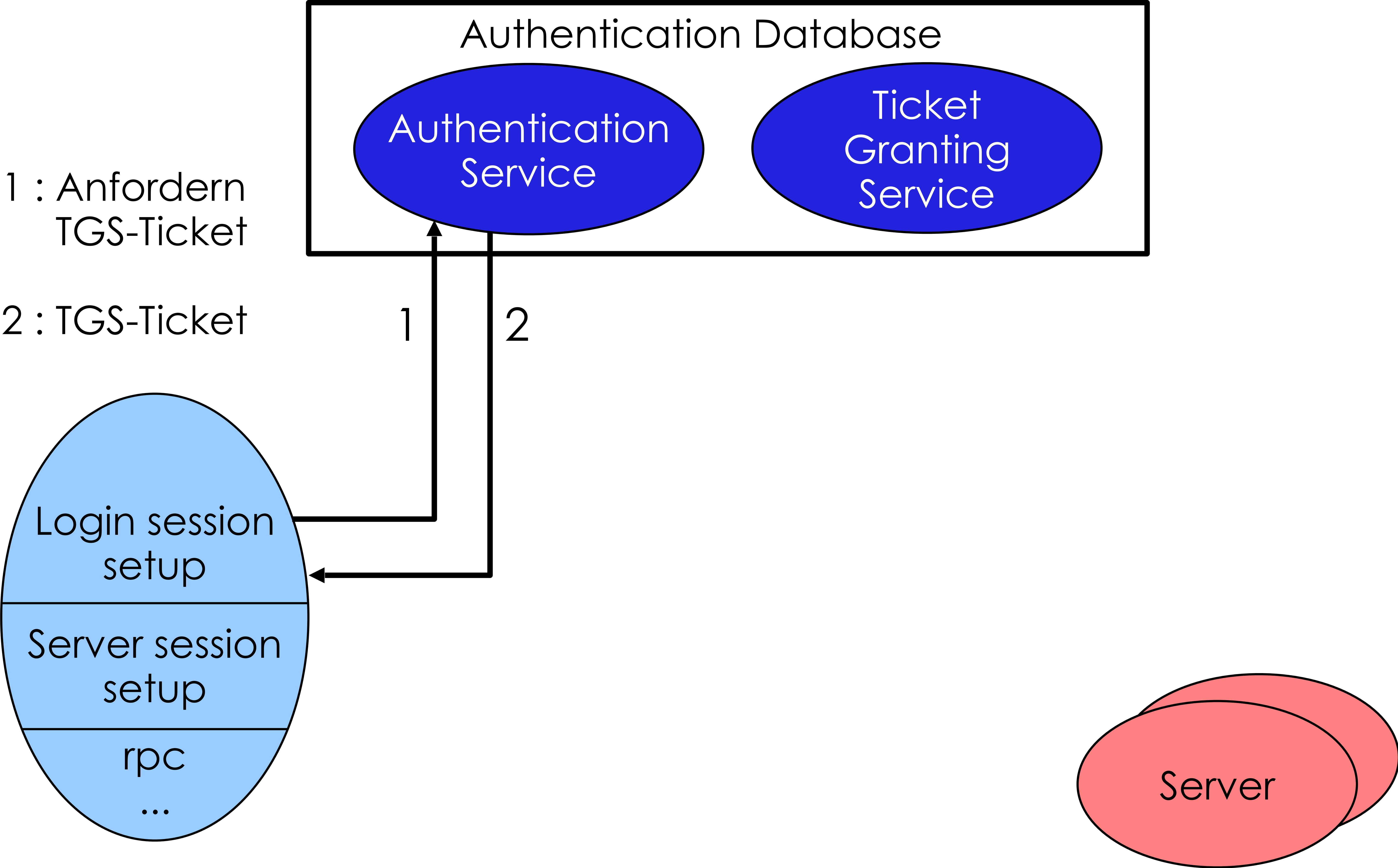
Verbesserung

Passwort nicht für jeden Server abfragen:
Indirektion über Ticket-Granting Service

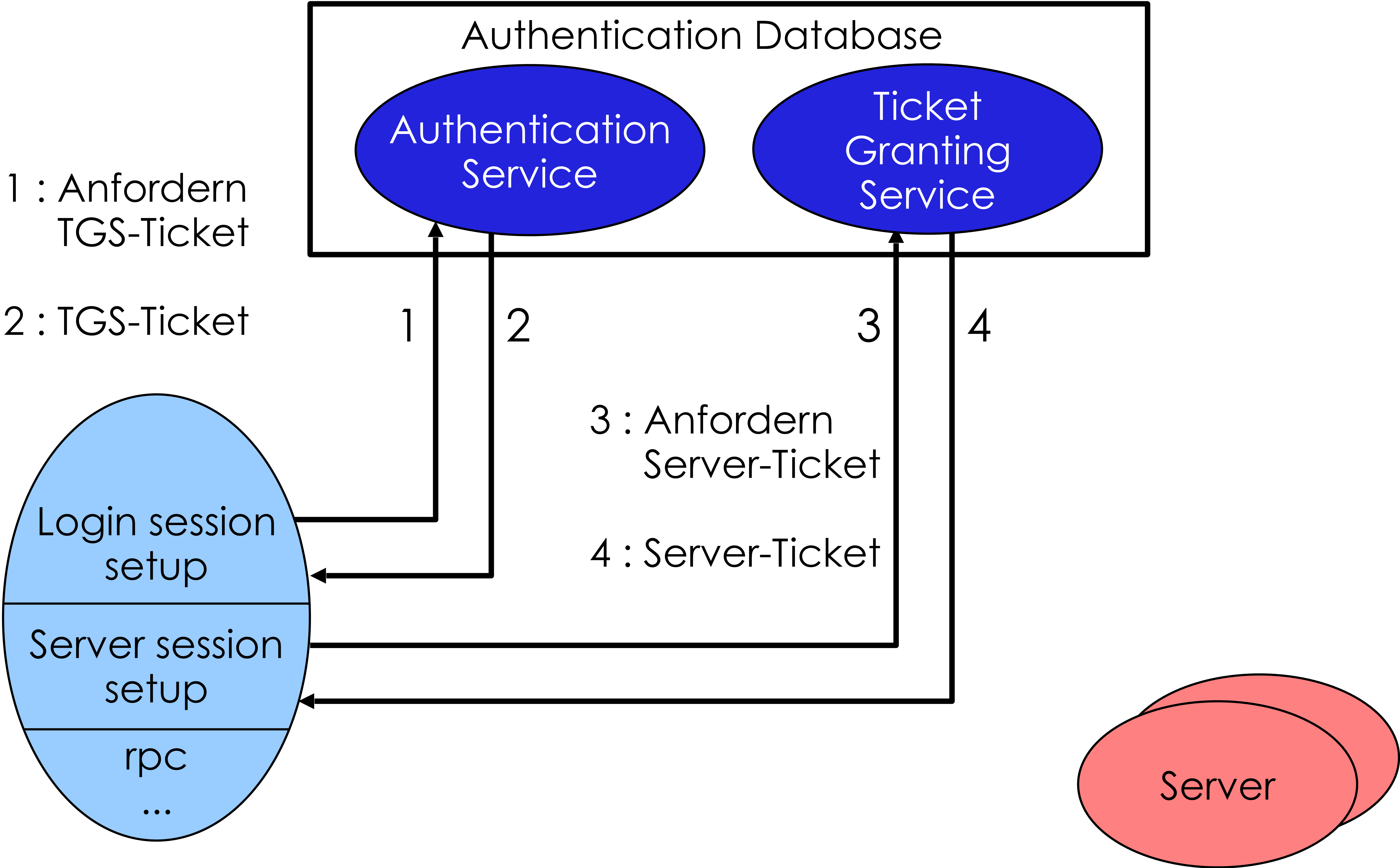
Key Distribution Center



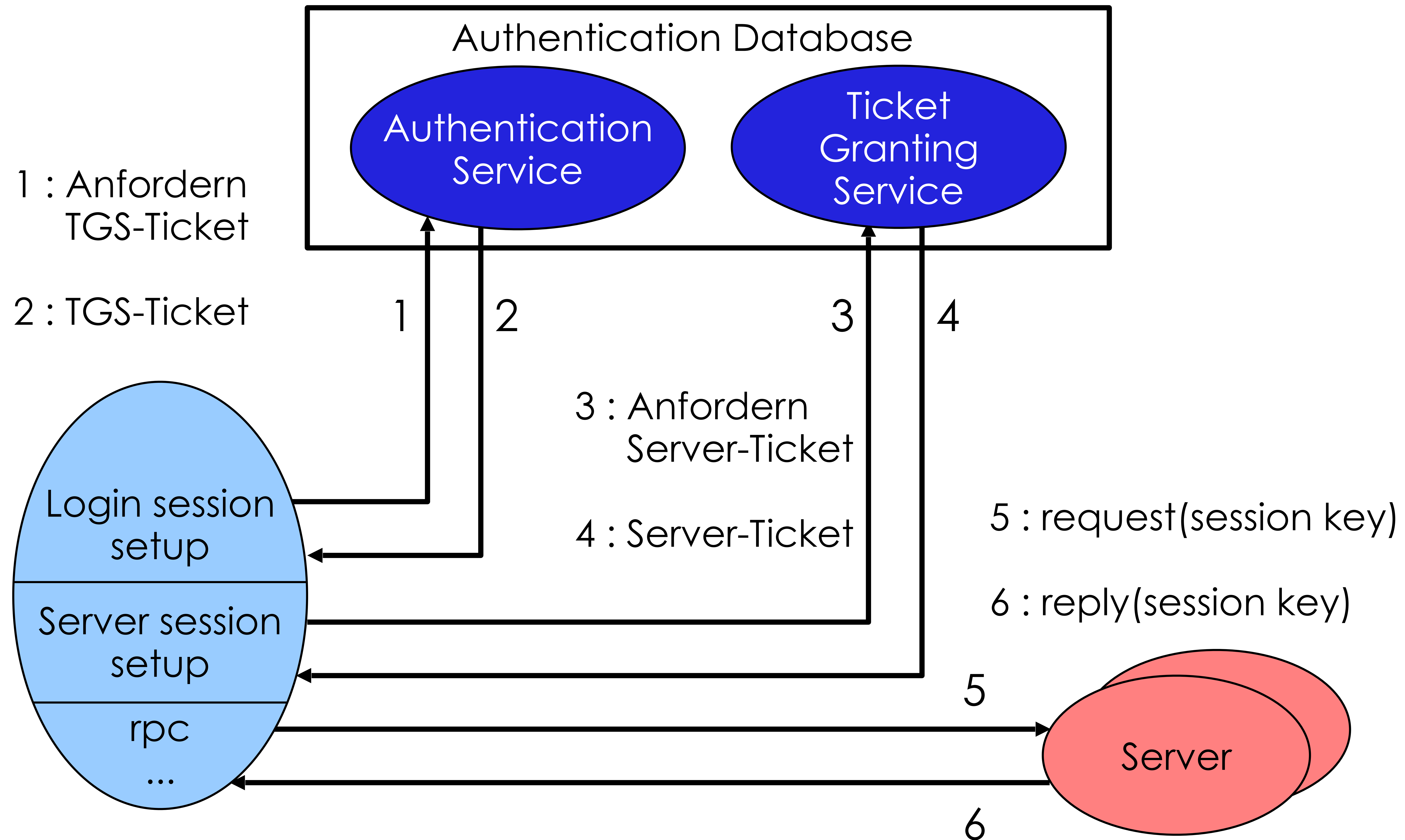
Key Distribution Center



Key Distribution Center



Key Distribution Center



Authentication und Ticket Granting

Authentication-Server

- kennt geheime Schlüssel aller Prinzipale,
  die aus deren Passwörtern erzeugt wurden
- kennt geheimen Schlüssel eines speziellen Servers:
  Ticket-Granting-Server

Ticket-Granting-Server

- kennt geheime Schlüssel der Server

Tickets

Ticket(C, S) : {C, S, t_1, t_2, K_{CS}}K_S

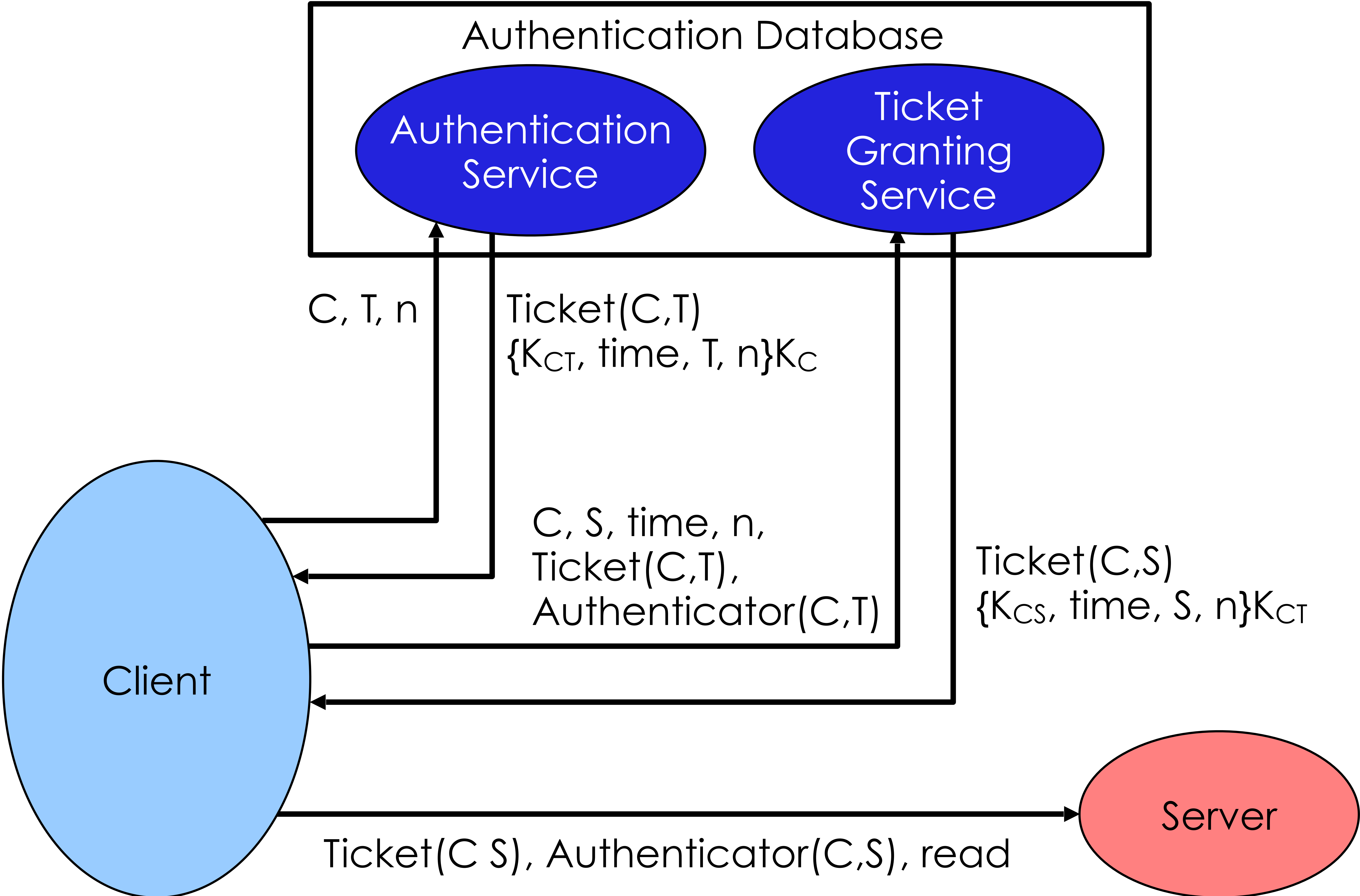
- Aufgabe: Nachweis dass kürzlich Authentication erfolgte
- Lebensdauer: t_1 bis t_2
- gültig für ein Client-Server-Paar: C, S
- nur S kann Ticket interpretieren/manipulieren da verschlüsselt durch K_S
- enthält session key: K_{CS}

Authenticator

Authenticator(C, S) : {C, t}K_{CS}

- Aufgabe: Identität des Absenders beweisen
- enthalten Zeitstempel: t
- gültig für ein Client-Server-Paar: C,S
- verschlüsselt durch session key
- werden laufend neu berechnet

Key Distribution Center



Ergänzendes zu Kerberos

Limitationen

- Schwachstelle Passwort erraten: K_C kann geprüft werden
- Dienste müssen Kerberos-Authentifikation unterstützen („Kerberizing“)

Realms

- administrative Einheiten mit AS und TGS

Verbreitung heute

- Microsoft Active Directory: eigene Kerberos-Erweiterungen
- Apple iCloud
- OAuth, FIDO: verwandte Ideen, andere Protokolle