

Verteilte Systeme 2

Prinzipien und Paradigmen

Hochschule Karlsruhe (HsKA)
Fakultät für Informatik und Wirtschaftsinformatik (IWI)
christian.zirpins@hs-karlsruhe.de

Kapitel 03: Prozesse

Version: 30. Oktober 2018



Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Inhalt

01: Einführung
02: Architekturen
03: Prozesse
04: Kommunikation
05: Benennung
06: Koordination
07: Konsistenz & Replikation
08: Fehlertoleranz
09: Sicherheit

Einführung in Threads

Grundidee

Wir erstellen **virtuelle Prozessoren** als *Software* basierend auf physikalischen Prozessoren:

Prozessor: Stellt eine Menge von Instruktionen bereit und kann Gruppen davon automatisch ausführen.

Thread: Minimaler Software Prozessor in dessen **Kontext** Gruppen von Instruktionen ausgeführt werden. Sichern des Thread Kontextes impliziert Stoppen der aktuellen Verarbeitung und Speichern aller Daten, um die Verarbeitung später fortzusetzen.

Prozess: Software Prozessor in dessen Kontext ein oder mehrere Threads ablaufen. Ausführung eines Threads meint Ausführung von Gruppen von Instruktionen im Kontext des Threads.

Kontextwechsel

Kontexte

- **Prozessor Kontext:** Minimale Sammlung von Registerwerten, die zur Ausführung einer Gruppe von Instruktionen genutzt werden (z.B. Stack Pointer, Adressregister, Programmzähler).
- **Thread Kontext:** Minimale Sammlung von Register- und Speicherwerten, die zur Ausführung einer Gruppe von Instruktionen genutzt werden (z.B. Prozessor Kontext, Thread Zustand, z.B. schlafend, blockiert etc.).
- **Prozess Kontext:** Minimale Sammlung von Register- und Speicherwerten, die zur Ausführung eines Threads genutzt werden (z.B. Thread Kontext, aber nun mindestens auch MMU Registerwerte).

Kontextwechsel

Beobachtungen

- 1 Threads teilen sich den gleichen Adressraum. Thread Kontextwechsel können komplett unabhängig vom Betriebssystem durchgeführt werden.
- 2 Prozesswechsel sind generell teurer, weil das Betriebssystem einbezogen wird (Umschalten vom Usermodus in den Kernelmodus).
- 3 Erzeugen und Zerstören von Threads ist viel billiger als bei Prozessen.

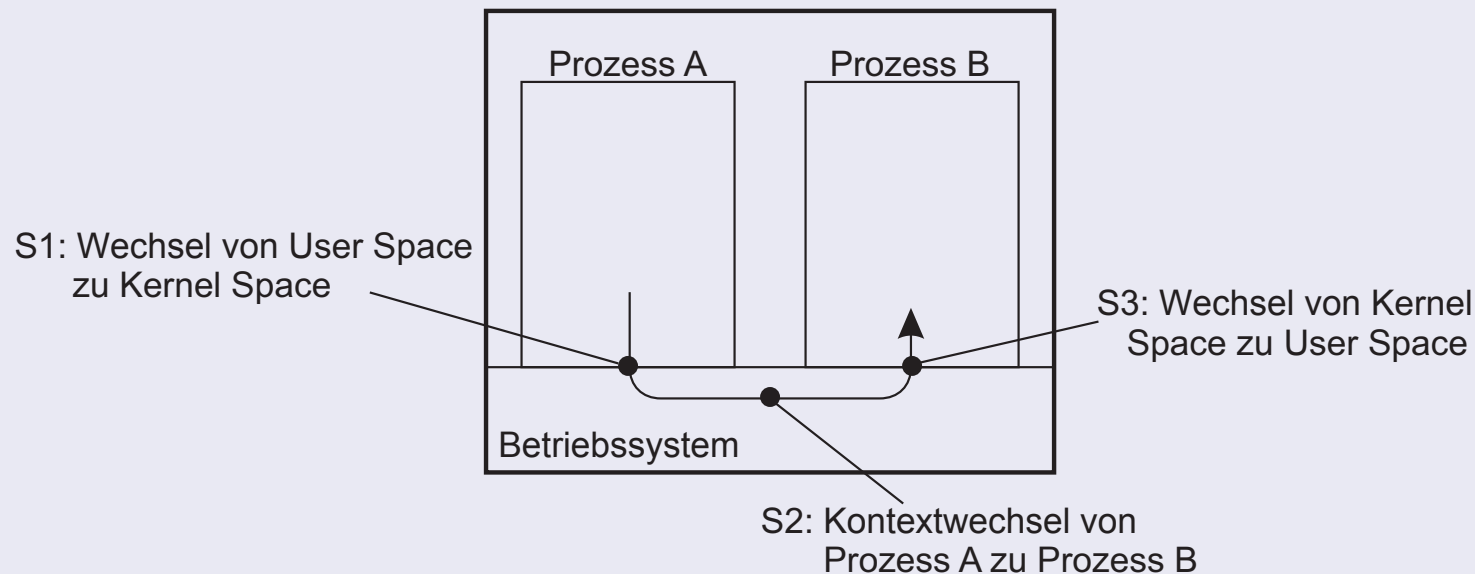
Warum nutzen wir Threads?

Einige Gründe

- **Keine Blockierung**: Single-Thread-Prozesse **blockieren** bei E/A Operationen. Bei Multi-Thread-Prozessen kann das BS die CPU auf einen anderen Thread im Prozess umschalten.
- **Parallelität nutzen**: Threads in Multi-Thread-Prozess können so geplant werden, dass sie parallel auf einem Multiprozessor oder Multicore-Prozessor System ausgeführt werden.
- **Keine Prozesswechsel**: Strukturierung großer Anwendungen nicht als Sammlung von Prozessen, sondern durch mehrere Threads.

Vermeide Prozesswechsel

Vermeide teure Kontextwechsel



Kompromisse

- Threads nutzen selben Adressraum: Fehleranfälliger.
- Keine Hilfe von BS/HW um Threads vor gegenseitigem Speicherzugriff schützen.
- Kontextwechsel von Threads ggf. schneller als von Prozessen.

Threads und Betriebssysteme

Grundproblem

Soll ein Betriebssystem Kernel-Threads bereitstellen, oder sollten diese als Pakete im Userspace implementiert werden?

Lösungen im Userspace

- Alle Operationen können **in einem Prozess** abgewickelt werden
⇒ Implementierungen sind extrem effizient.
- **Alle** Kerneldienste werden **stellvertretend für den Prozess** erbracht, in dem der Thread läuft ⇒ wenn der Kernel einen Thread blockiert, wird der ganze Prozess blockiert.
- Threads werden gerne bei vielen externen Ereignissen benutzt ⇒ wenn der Kernel keine Threads unterscheiden kann, wie kann er Ereignisse signalisieren?

Threads und Betriebssysteme

Lösungen im Kernel

Idee ist, das Thread-Paket im Kernelspace zu implementieren. Das bedeutet, dass alle Operationen zu Systemaufrufen führen.

- Blockierende Operationen im Thread sind kein Problem: **Kernel plant anderen verfügbaren Thread** im gleichen Prozess ein.
- Externe Ereignisse bearbeiten ist einfach: der **Kernel plant einen Thread in Assoziation mit dem Ereignis**.
- Das große Problem ist **Effizienzverlust**, da jede Thread Operation ein Umschalten in den Kernelmodus erfordert.

Schlussfolgerung – aber...

Kombiniere Threads auf User- und Kernel-Ebene – leider wiegt die Leistungssteigerung nicht die erhöhte Komplexität auf.

Nutzung von Threads auf Client-Seite

Multithread Web Client

Netzwerklatenz verstecken:

- Der Web Browser untersucht eine neue HTML Seite und erkennt, dass **weitere Dateien geladen werden müssen**.
- **Jede Datei wird in einem separaten Thread geladen**, der einen (blockierenden) HTTP Request bewirkt.
- Wenn Dateien ankommen, zeigt der Browser diese an.

Mehrere Request/Reply-Aufrufe zu entfernten Maschinen (RPC)

- Ein Client macht mehrere Aufrufe gleichzeitig in mehreren Threads.
- Er wartet dann, bis alle Resultate zurückgegeben wurden.
- Anmerkung: wenn die Aufrufe an verschiedene Server gehen, kann **lineare Beschleunigung** erreicht werden.

Nutzung von Threads auf Server-Seite

Steigerung der Leistung

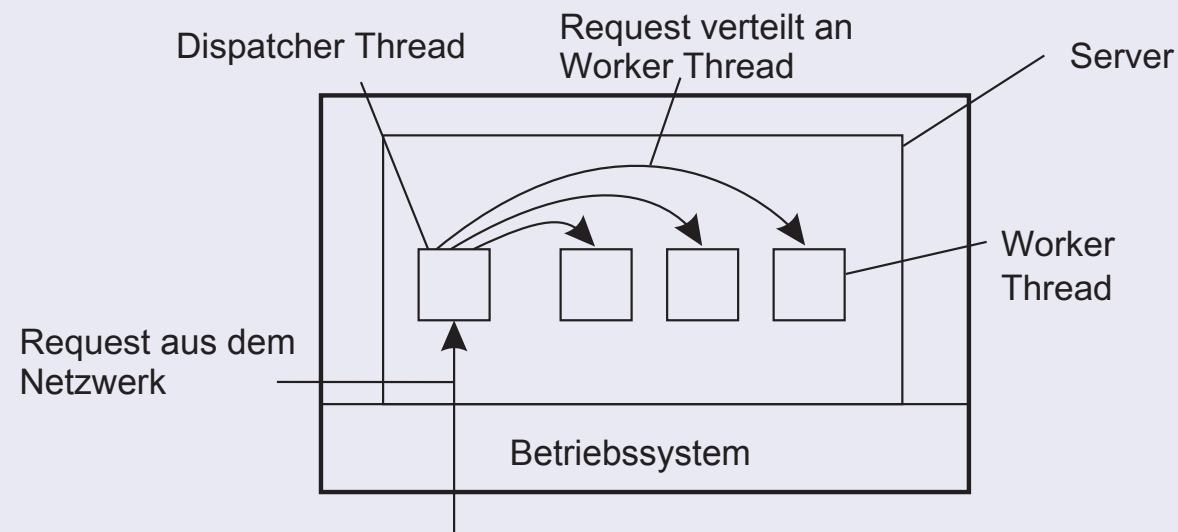
- Einen Thread zu starten ist **viel** billiger als einen neuen Prozess.
- Ein Single-Threaded Server kann nicht einfach auf ein **Multiprozessor System** skaliert werden.
- Wie bei Clients: **Netzwerklatenz verstecken** durch Abwicklung des nächsten Requests, während der vorherige noch beantwortet wird.

Bessere Struktur

- Viele Server sind E/A-lastig. Nutzung bekannter **blockierender Aufrufe** (in separaten Threads) vereinfacht die Struktur.
- Multithread Programme sind oft **kürzer und verständlicher**, da der **Kontrollfluss einfacher ist**.

Warum Multithreading beliebt ist : Organisation

Dispatcher/Worker Modell



Übersicht

Modell	Charakteristik
Multithreading	Parallelität, blockierende Systemaufrufe
Single-Thread Prozess	Keine Parallelität, blockierende Systemaufrufe
Endlicher Automat	Parallelität, nichtblockierende Systemaufrufe

Übung 3: Singlethread vs. Multithread Server


Fallbeispiel: Dateiserver

Wir vergleichen das Lesen einer Datei mit Singlethread- und Multithread Dateiserver.

Bei Daten im Cache dauert es 5 ms, um einen Request anzunehmen, zuzuweisen und die restliche Verarbeitung durchzuführen.

Annahme: in 1/3 der Fälle ist Festplatten-Operation nötig. Dann kommen 60 ms hinzu, in denen der Thread ruht (blockierender Aufruf).

Frage

1 Wieviele Anforderungen/Sek. schafft ein Singlethread-Server? 



2 Wieviele sind es bei einem Multithread-Server? 

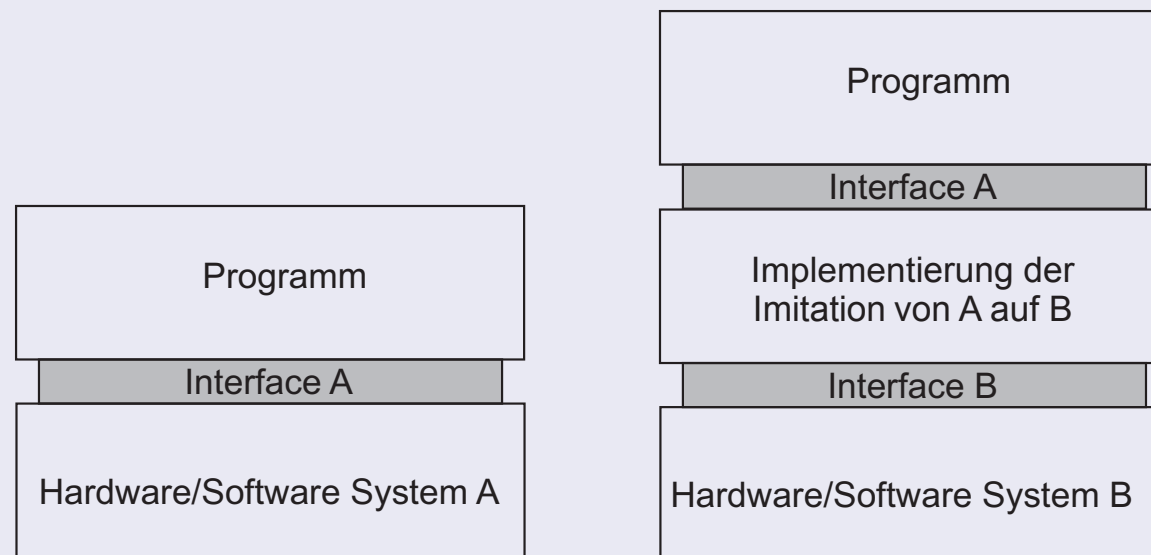
Virtualisierung

Beobachtung

Virtualisierung ist wichtig:

- Hardware **ändert sich schneller** als Software
- Einfache **Portierbarkeit** und Code Migration
- **Isolation** ausgefallener oder angegriffener Komponenten

Prinzip: Interfaces imitieren



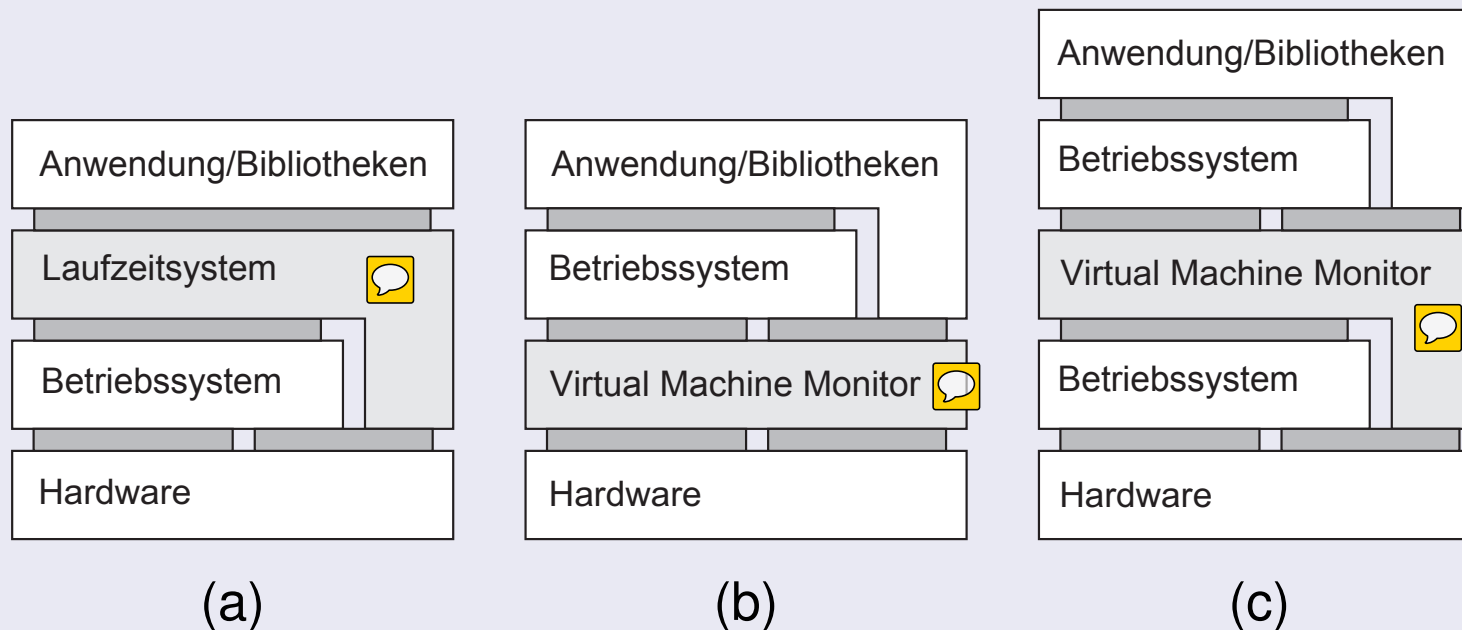
Nachahmung von Interfaces

Vier Arten von Interfaces auf drei Ebenen

- 1 **Befehlssatzarchitektur**: Menge von Maschineninstruktionen mit zwei Untermengen:
 - Privilegierte Instruktionen: Ausführung nur durch BS
 - Generelle Instruktionen: Ausführung durch jedes Programm
- 2 **Systemaufrufe** angeboten vom BS.
- 3 **Bibliotheksaufrufe** (**Application Programming Interface (API)**)

Arten der Virtualisierung

(a) Prozess VM, (b) Nativer VMM, (c) Hosted VMM



Unterschiede

- (a) Separate Instruktionen, Interpreter/Emulator läuft auf BS.
- (b) Low-level Instruktionen, zusammen mit minimalem BS.
- (c) Low-level Instruktionen, delegiert meiste Arbeit an volles BS.

VMs und Cloud computing

Drei Typen von Cloud Services

- **Infrastructure-as-a-Service** betrifft Basisinfrastruktur
- **Platform-as-a-Service** betrifft Dienste der Systemebene
- **Software-as-a-Service** betrifft echte Anwendungen

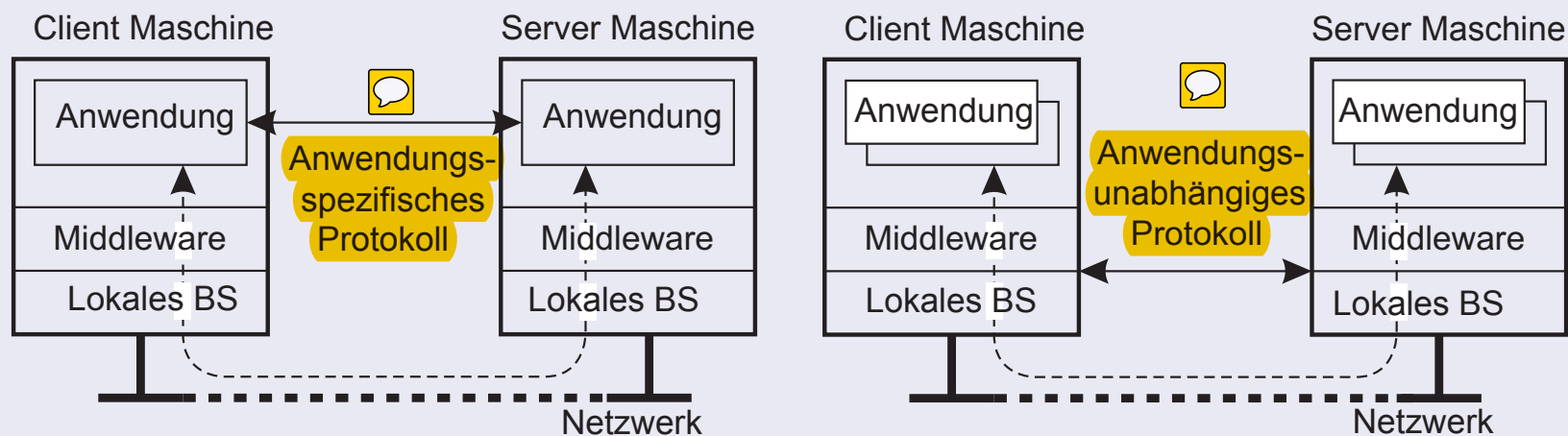
IaaS

Anstatt einen physischen Computer zu vermieten, vermietet ein Cloud-Anbieter eine VM (oder VMM), die ggf. eine physische Maschine mit anderen Kunden teilt \Rightarrow fast vollständige Isolierung zwischen Kunden (**Leistungsisolierung** ist aber nicht immer möglich).



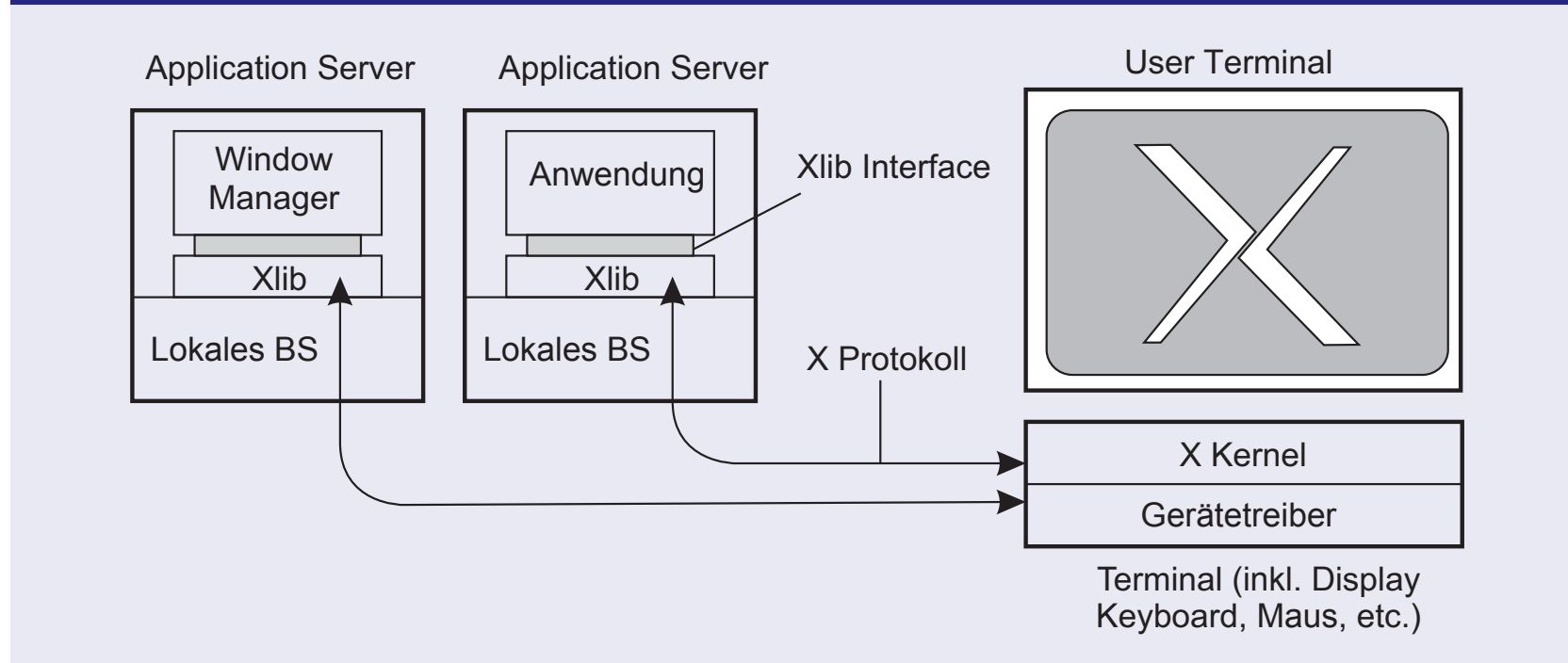
Client-Server Interaktion

Unterscheide Lösungen auf Anwendungs- und Middleware-Ebene




Beispiel: X Window System

Grundlegender Aufbau



X Client und Server

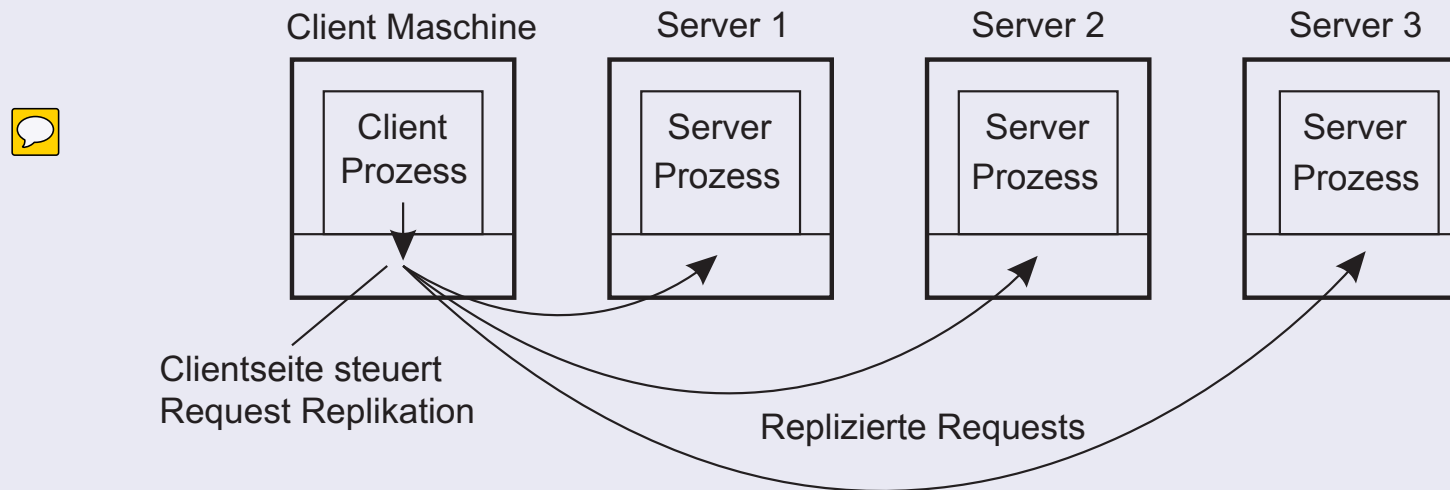
Anwendung ist **Client** des X-Kernel. Letzterer läuft als **Server** auf Maschine des Anwenders. 



Clientseitige Software

Generell zugeschnitten auf Verteilungstransparenz

- **Zugriffstransparenz:** clientseitige RPC-“Stubs”
- **Orts-/Migrationstransparenz:** Client überwacht aktuellen Ort
- **Replikationstransparenz:** mehrfache Aufrufe durch Client-Stub.



- **Fehlertransparenz:** kann oft nur beim Client platziert werden (Versuch, Server- und Kommunikationsausfälle zu verbergen).

Server: Genereller Aufbau

Basismodell

Prozess, der einen **Service** für Clients implementiert.

Er wartet auf die eingehende **Anfrage** von einem Client und stellt anschließend sicher, dass sie **bearbeitet** wird.

Danach wartet er auf die nächste Anfrage.

Nebenläufige Server

Zwei grundlegende Typen

- **Iterative Server**: Server bearbeitet einen Request nach dem anderen.
- **Nebenläufige Server**: nutzt **Dispatcher**, der eingehende Requests annimmt und an separate Threads/Prozesse weiterreicht.

Beobachtung

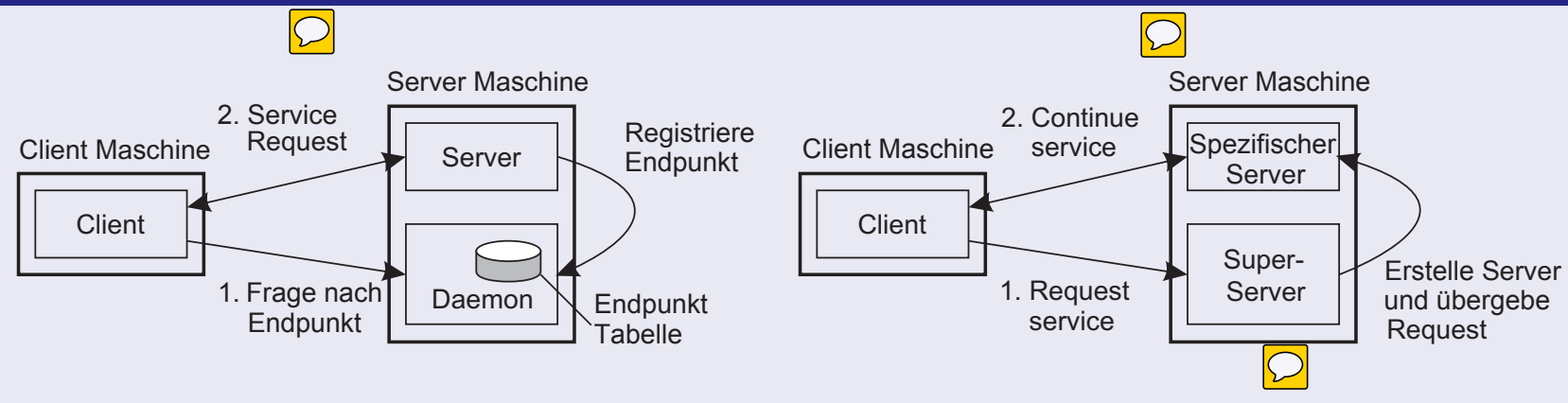
Nebenläufige Server sind die Norm: Sie können problemlos mehrere Requests verarbeiten, auch bei blockierenden Operationen (auf Festplatten oder andere Server).

Server kontaktieren

Beobachtung: die meisten Server haben einen spezifischen Port

ftp-data	20	File Transfer [Default Data]
ftp	21	File Transfer [Control]
telnet	23	Telnet
smtp	25	Simple Mail Transfer
www	80	Web (HTTP)

Ports dynamisch zuweisen



Out-of-Band Kommunikation

Problem

Ist es möglich einen Server zu **unterbrechen**, nachdem er eine Dienstanfrage angenommen hat?

Lösung 1: Nutze separaten Port für dringende Daten

- Server hat einen separaten Thread/Prozess für dringende Nachrichten
- Dringende Nachricht kommt an \Rightarrow **assozierten Request anhalten**
- Anmerkung: erfordert **Betriebssystemunterstützung für prioritätsbasiertes Scheduling**

Lösung 2: Nutze Mechanismen der Transportschicht:

- Beispiel: TCP erlaubt dringende Nachrichten in der gleichen Verbindung
- Dringende Nachrichten vom Betriebssystem signalisieren lassen

Server und Zustand

Zustandsbehaftete Server

Verfolgt den Status seiner Clients nach, z.B. ein **Dateiserver**:

- Speichert, dass eine Datei geöffnet wurde, so dass im voraus gelesen werden kann
- Weiß, welche Daten ein Client gecached hat und erlaubt es Clients, lokale Kopien geteilter Daten zu halten

Beobachtung

- Bei Verlust des Zustands wäre die Funktion des Systems beeinträchtigt ⇒ erfordert **Wiederherstellung**
- Die **Performanz zustandsbehafteter Server kann extrem hoch sein**, wenn Clients lokale Kopien halten dürfen. Wie sich zeigt, ist **Verlässlichkeit meist kein großes Problem**.

Server und Zustand

Zustandslose Server

Es werden keine **detaillierten** Informationen über den Status von Clients nach Bearbeitung einer Anfrage gehalten, z.B. bei Dateiserver:

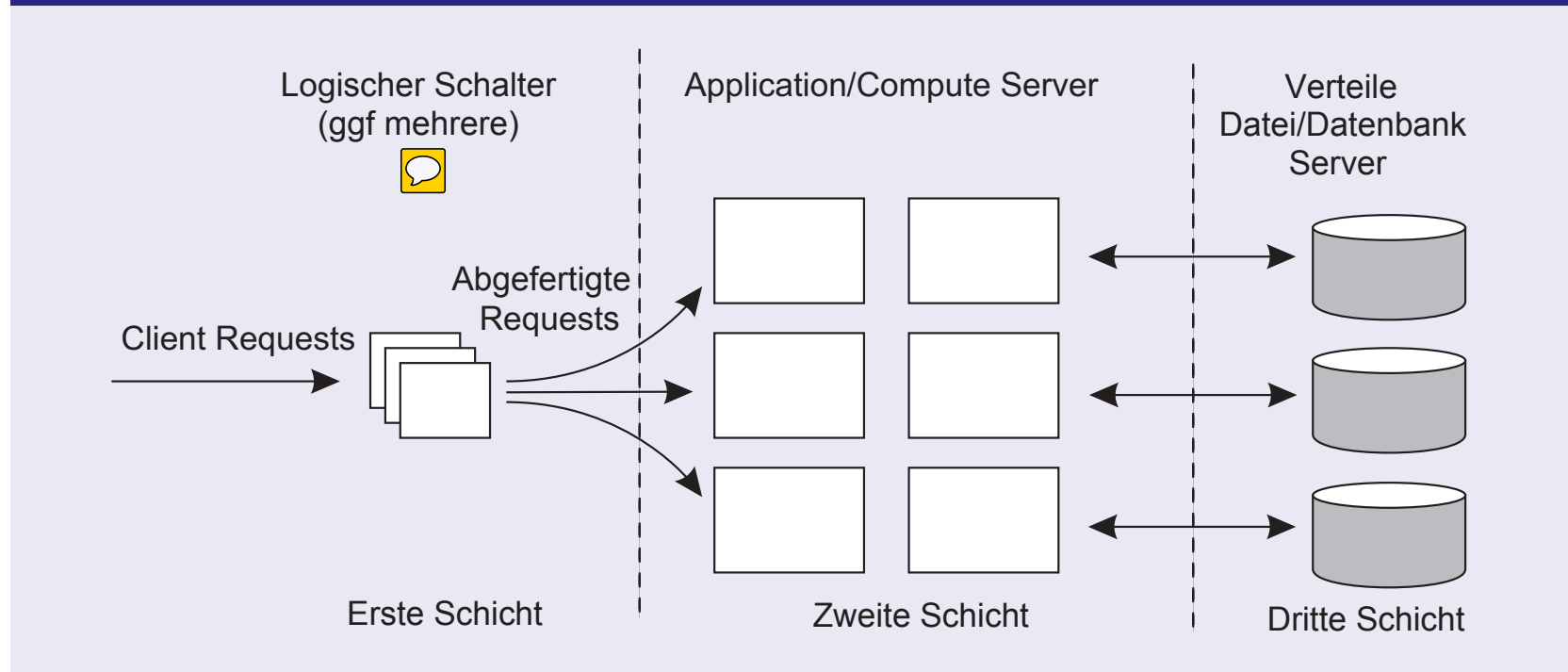
- Nicht speichern ob eine Datei geöffnet wurde (einfach nach Zugriff wieder schließen)
- Nicht versprechen den Client Cache zu invalidieren
- Clients nicht nachverfolgen

Konsequenzen

- Clients und Server sind **unabhängig**
- **Inkonsistenz des Zustands** z.B. durch Ausfälle **werden reduziert**
- Ggf. **weniger Performanz**, z.B. weil Client-Verhalten nicht mehr antizipiert werden kann (Dateiblöcke im voraus lesen etc.)

Dreischicht-Modell

Üblicher Aufbau



Wesentliches Element

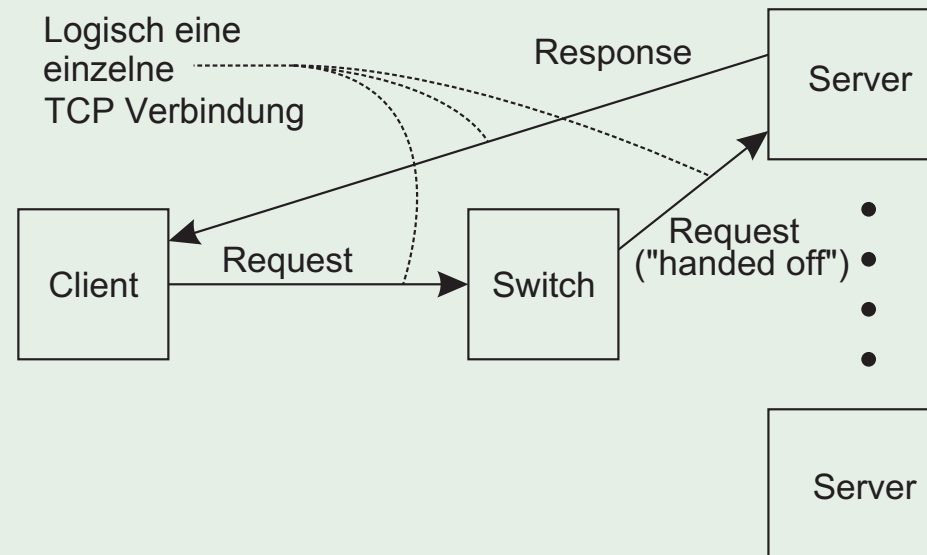
Die erste Ebene ist immer für die Weiterleitung von Requests an einen passenden Server Verantwortlich ([Request Dispatching](#)).

Request Verarbeitung

Beobachtung

Wenn die erste Ebene alle Kommunikation von/zum Cluster verarbeitet, entsteht ggf. ein **Engpass**.

Mögliche Lösung: TCP-Handoff

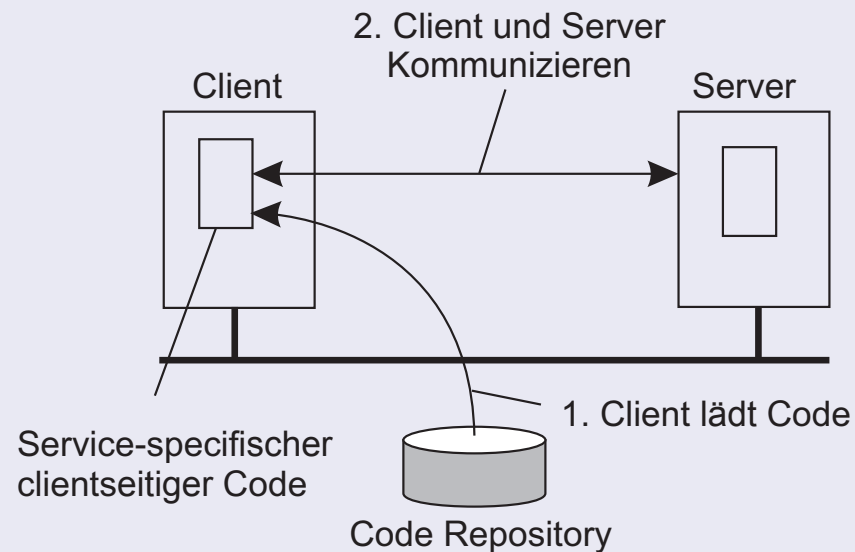


Code Migration: Gründe

Lastverteilung

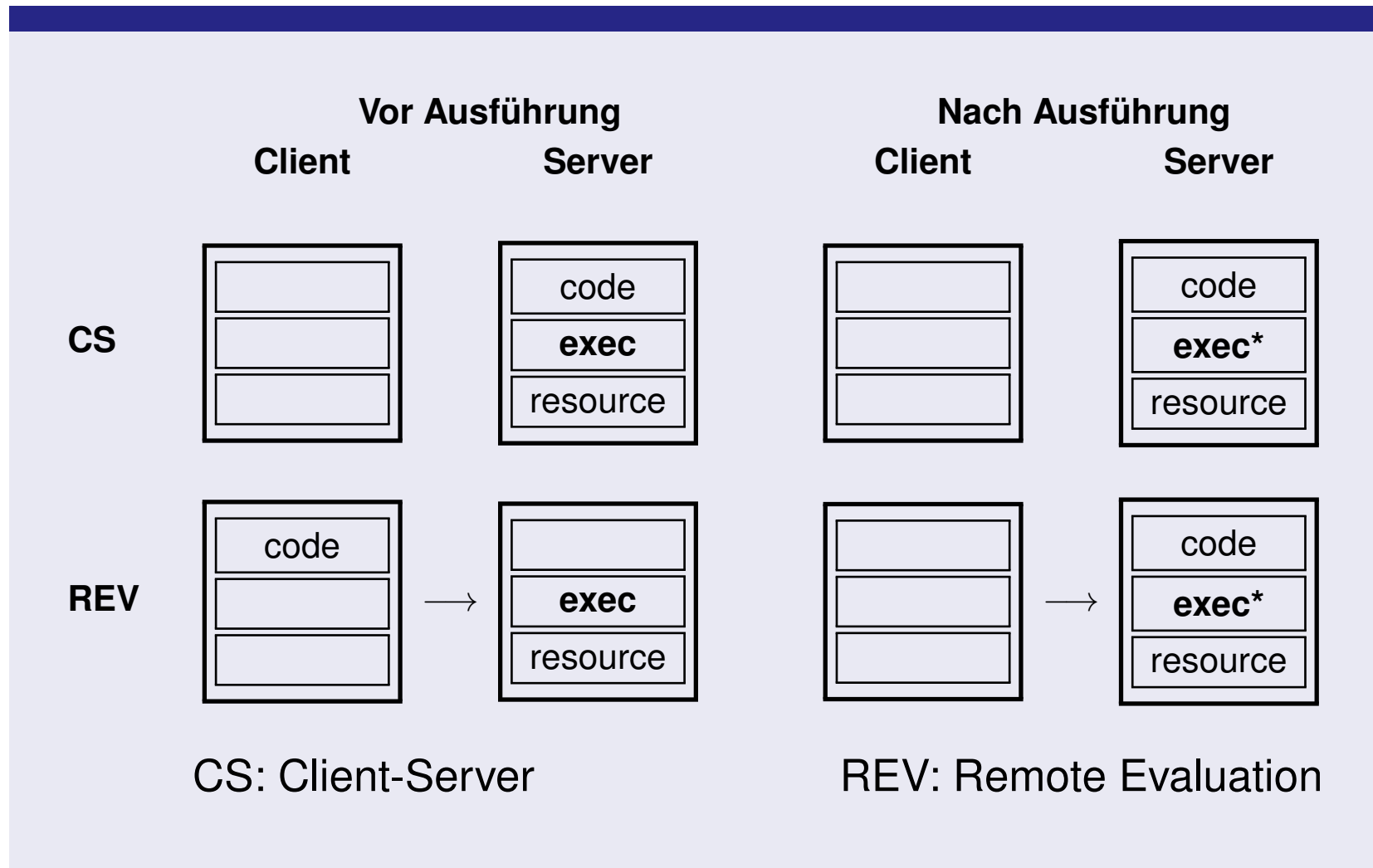
- Sicherstellen, dass Server in einem Rechenzentrum **ausreichend** ausgelastet sind (z.B. um Energieverschwendung zu vermeiden)
- Minimierung von Kommunikation durch Verschiebung von Berechnungen nahe zu den Daten (z.B. Mobile Computing).

Flexibilität: Code nach Bedarf zum Client verschieben

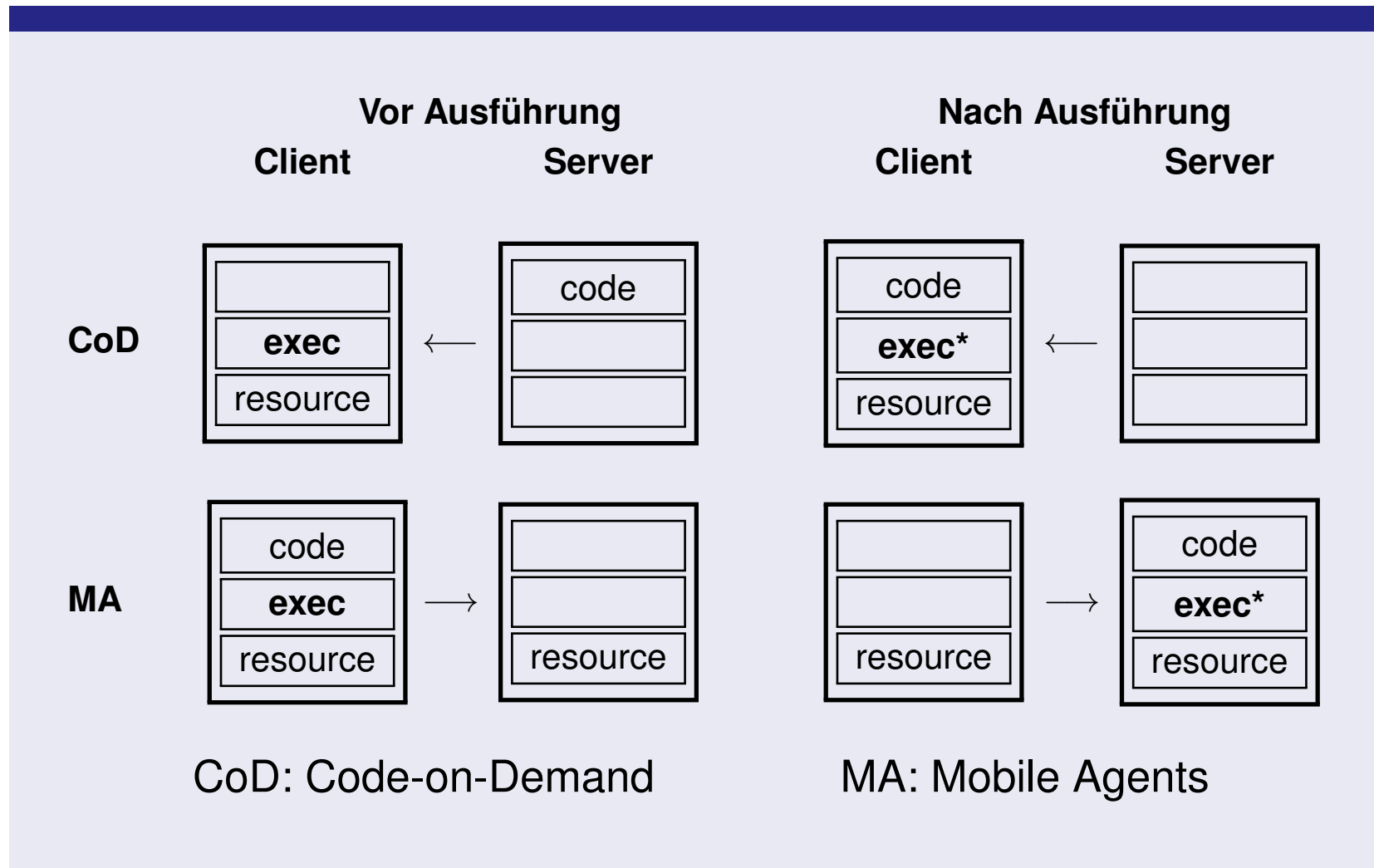


Vermeidet vorinstallierte Software und erhöht die Dynamik der Konfiguration.

Modelle für Code Migration



Modelle für Code Migration



Starke und schwache Mobilität

Objekt Komponenten

- **Code-Segment**: enthält den eigentlichen Code
- **Daten-Segment**: enthält den Zustand
- **Ausführungszustand**: enthält Kontext des Threads, der den Code ausführt

Schwache Mobilität: verschiebe nur Code- und Daten-Segment (Neustart)

- Relativ einfach, speziell bei portablem Code
- Unterscheide **“Code Shipping”** (push) von **“Code Fetching”** (pull)

Starke Mobilität: verschiebe Komponente, inkl. Ausführungszustand

- **Migration**: verschiebe ganzes Objekt von einer Maschine zur anderen
- **Cloning**: starte einen Klon und versetze ihn in gleichen Ausführungszustand

Migration in heterogenen Systemen

Hauptproblem

- Der Zielknoten ist ggf. **nicht geeignet**, um migrierten Code auszuführen
- Die Definition von Prozess-/Thread-/Prozessorkontext ist **stark von lokaler Hardware sowie vom Betriebs- und Laufzeitsystem abhängig**.

Einzigste Lösung: abstrakte Maschine implementiert auf unterschiedlichen Plattformen

- Interpretierte Sprachen mit eigener VM (z.B. Java)
- Virtuelle Maschine (wie zuvor besprochen)