Verteilte Systeme
Grundwissen

Ich hätte da mal eine Frage:

Was ist das, ein Verteiltes System?

Mögliche Definitionen

Naive Antwort: Ein verteiltes System besteht aus …

- Einer beliebigen Anzahl von Computern,
- Die durch ein Netzwerk miteinander verbunden sind,
- Und mit einer Software zur Koordination ausgestattet sind (Netzwerkdienste des Betriebssystems)

Zum Beispiel: Netzwerkdrucker oder Dateifreigaben

Mögliche Definitionen

Definition von Leslie Lamport:

Ein verteiltes System ist ein System, mit dem ich nicht arbeiten kann, weil irgend ein Rechner abgestürzt ist, von dem ich nicht einmal weiß, dass es ihn überhaupt gibt.

Definition von Andrew S. Tanenbaum:

Ein verteiltes System ist eine Kollektion unabhängiger Computer, die dem Anwender als Einzelcomputer erscheinen.

Mögliche Definitionen

Tanenbaums Definition implziert …

- Mehrere miteinander verbundene Computer
- Eine gemeinsame Ressourcennutzung (z.B. Hardware)
- Ein einheitliche Sicht auf das Gesamtsystem

Klausurrelevante Definition:

Ein verteiltes System ist ein System, in dem sich Hardware- und Softwarekomponenten auf vernetzten Computern befinden können und diese nur über den Austausch von Nachrichten miteinander kommunizieren, um ihre Aktionen zu koordinieren.

Was bedeutet das?

Ein verteiltes System besteht aus …

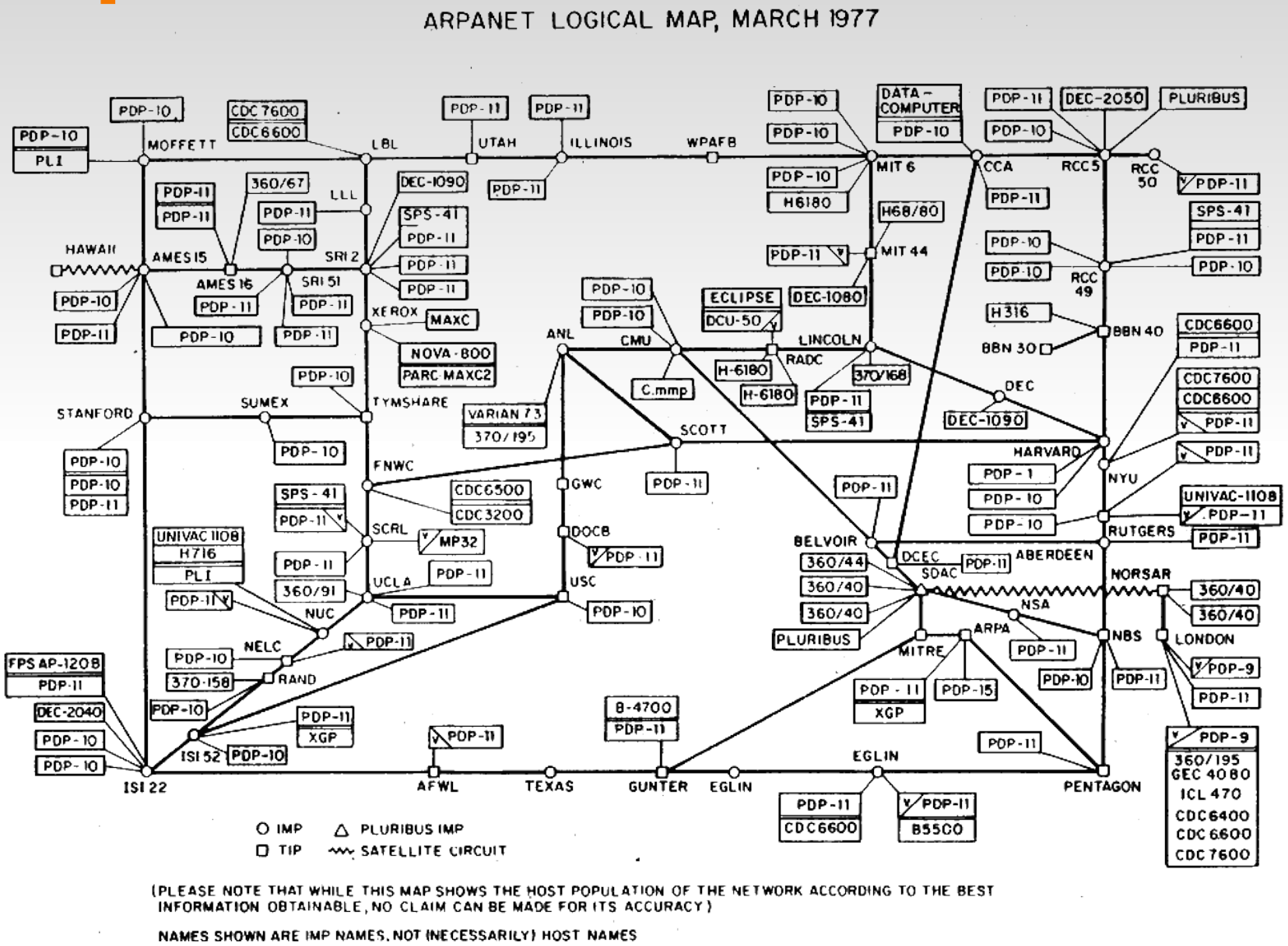
- Mindestens einem Computer und
- Mindestens zwei parallel laufenden Anwendungen, die eine gemeinsame Aufgabe erfüllen

Die Koordination erfolgt …

- Durch den Austausch einfacher Nachrichten
- Jedoch nicht durch direkte Methodenaufrufe etc.

Es entstehen also dieselben Synchronisationsprobleme wie bei der nebenläufigen Programmierung!

Beispiel: ARPANET bzw. Internet



Quelle: ARPANET Completion Report (1977), nach 10 Jahren ARPANET

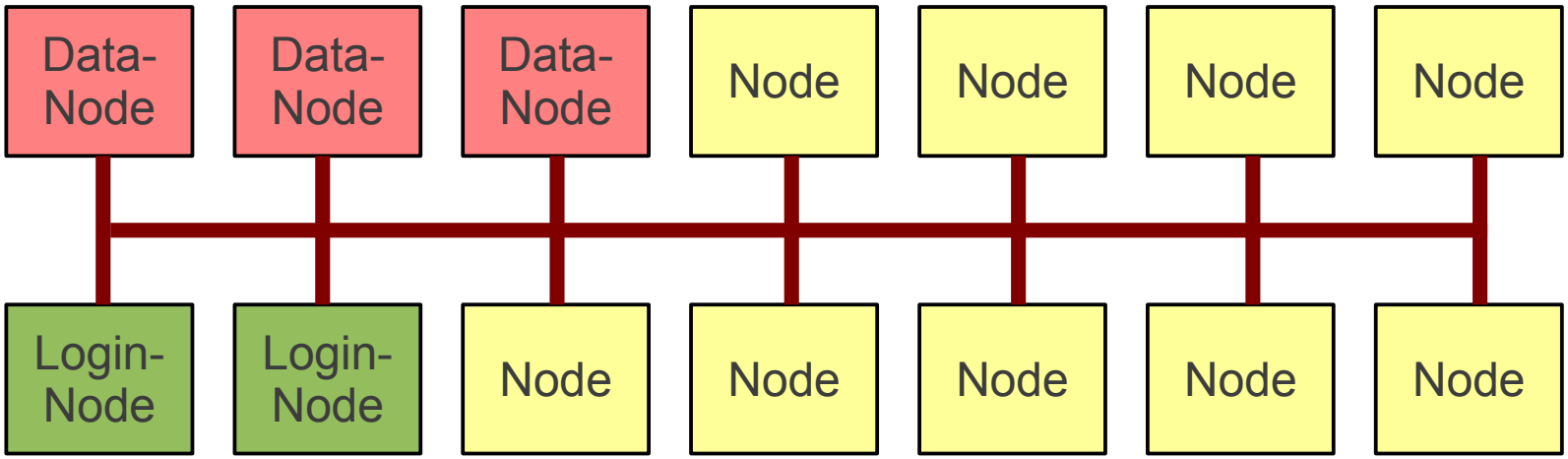
Beispiel: Grid Computing

Aus vielen Einzelcomputern wird ein Großcomputer

Einplanung von Jobs über spezielle Loginknoten

Verteilung der Rechenlast durch die Kontrollsoftware

Gemeinsamer Datenbestand für alle Knoten



Probleme verteilter Systeme

- Viele gleichzeitige (parallele) Aktivitäten
- Exakte globale Zeit nicht erfahrbar/vorhanden
- Keine konsistente Sicht des Gesamtzustandes
- Kooperation durch Kommunikation
- *Ursache* und *Wirkung* zeitlich getrennt

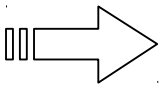
> Räumliche Separation, autonome Komponenten

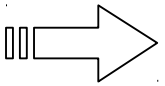
> Heterogenität

> Dynamik, Offenheit

> Komplexität

> Sicherheit



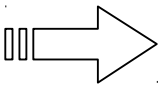


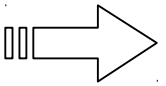
> Probleme sequentieller Systeme

> Nebenläufigkeit

> Nichtdeterminismus

> Zustandsverteilung





> Synchronisation schwieriger

> Programmierung komplexer

> Testen aufwendiger

Definition: Verteilte Anwendung

Eine verteilte Anwendung …

- Läuft auf einem verteilten System und
- Besteht aus parallel ablaufenden Komponenten, die untereinander und mit dem Anwender kommunizieren

Beispiele verteilter Anwendungen:

- Einfache Internetanwendungen wie HTTP oder IMAP
- Verteilte Informationssysteme, z.B. Flugbuchungssysteme
- Geografische Simulationen auf Grid Computern
- Groupware-Installationen in weltweit tätigen Unternehmen

Das Marketing verspricht

Gemeinsame Ressourcen
Alle Ressourcen werden optimal ausgenutzt

Offenheit
Das System lässt sich einfach erweitern

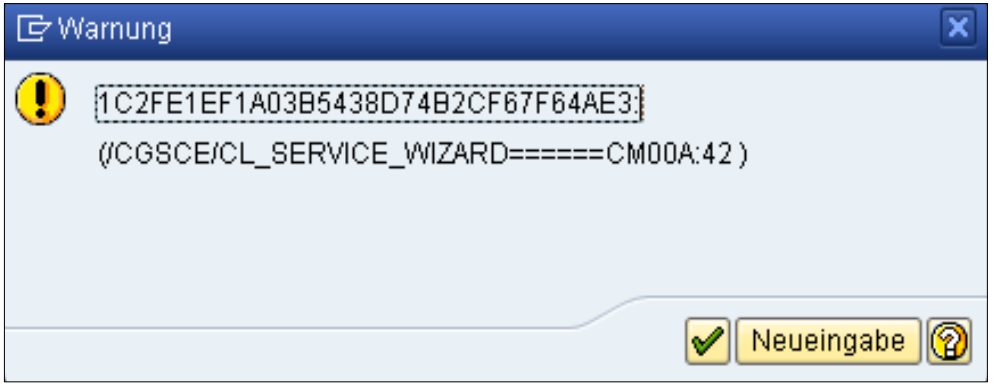
Skalierbarkeit
Erweiterungen führen nicht zu Leistungsverlust

Nebenläufigkeit
Bessere Ressourcennutzung dank nebenläufiger Prozesse

Sicherheit
Keine Spionage, Verlust und Verfälschung von Daten

Fehlertoleranz
Fehler werden automatisch erkannt und kompensiert

Transparenz
Komplexe Details des Systems bleiben verborgen



Offenheit und Skalierbarkeit

Ergänzung des Systems um weitere Funktionen

Erweiterung des Systems bei steigender Beanspruchung

- Wenn zum Beispiel aus 5 Benutzern 5.000 Benutzer werden
- Oder wenn der Speicherbedarf von 1 GiB auf 120 GiB wächst
- Erschöpfung von Ressourcen verhindern (Vgl. IPv4/IPv6)
- Leistungsengpässen vorbeugen (z.B. durch Lastverteilung)

Lineare Erweiterbarkeit mit O(n) wird angestrebt

- Beliebige Komponenten können einfach hinzugefügt werden
- Kein Overhead und keine Obergrenzen für Erweiterungen

Sicherheitsaspekte

Vertraulichkeit
Kein Zugriff auf die Daten für unbefugte Dritte

Datenintegrität
Verhindert die Verfälschung von Daten, beispielsweise während der Übertragung

Authentizität
Zweifelsfreie Feststellung der Absenderidentität. Jemand drittes kann sich nicht für den Absender ausgeben

Verfügbarkeit
Entfernte Ressourcen sollen immer verfügbar sein und nicht durch DoS-Attacken gestört werden können

Mobiler Code
Mobiler Code darf das System nicht beschädigen, das System darf mobilen Code nicht beschädigen

Beispiel: Sicherheit

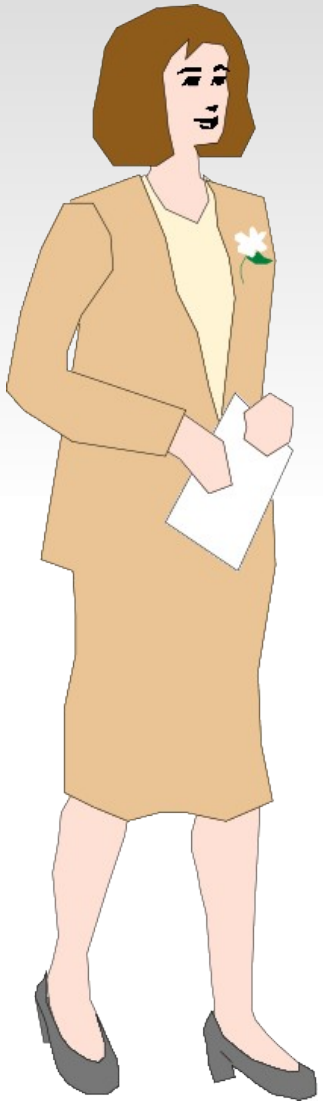
Symmetrische Verschlüsselungsverfahren:

- Beruhen auf einem einfachen Verschlüsselungscode
- Der Code muss allen Beteiligten bekannt sein
- Übertragung des Codes selbst ist eine häufige Sicherheitslücke

Asymmetrische Verschlüsselungsverfahren:

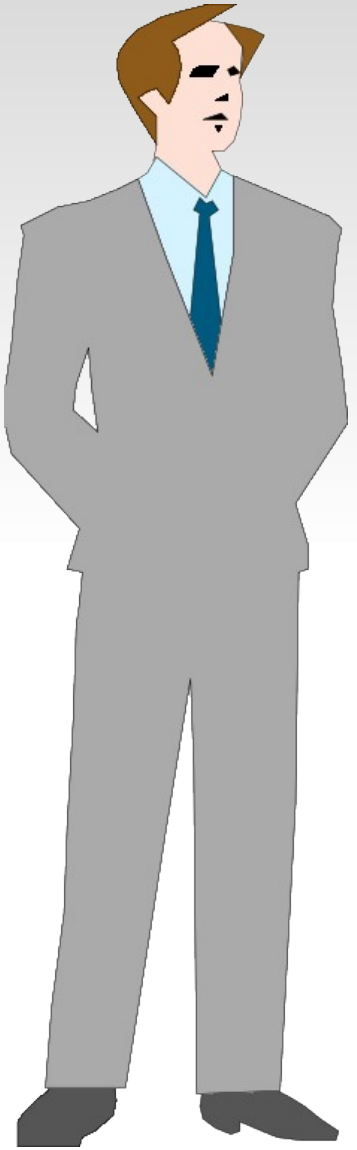
- Beruhen auf zwei komplementären Verschlüsselungscodes
- Verschlüsselung ist mit beiden Codes möglich, Entschlüsselung jedoch nur mit dem jeweils anderen Code
- Ein Schlüssel wird als **Public Key** öffentlich gemacht, der andere Schlüssel verbleibt als **Private Key** beim Besitzer

Beispiel: Sicherheit



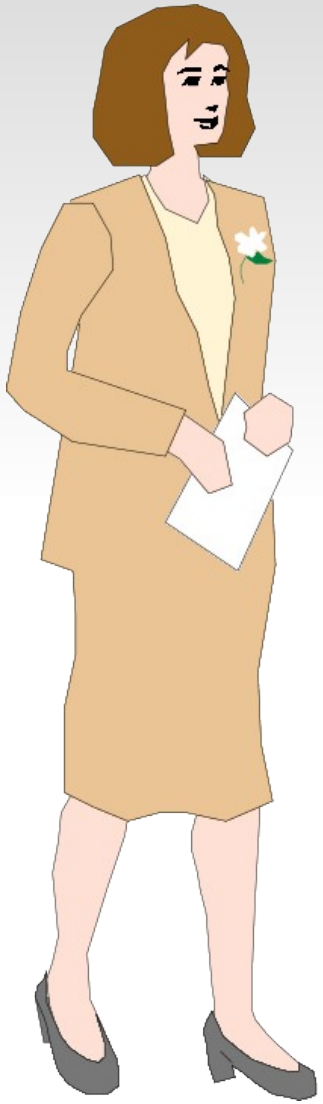
Alice

Gesicherte Kommunikation zwischen Alice und Bob



Bob

Beispiel: Sicherheit

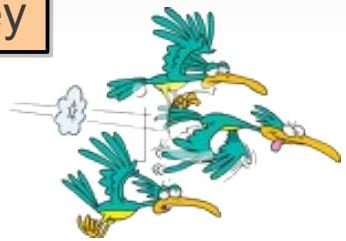


Alice

Zertifikat mit Alices Public Key →

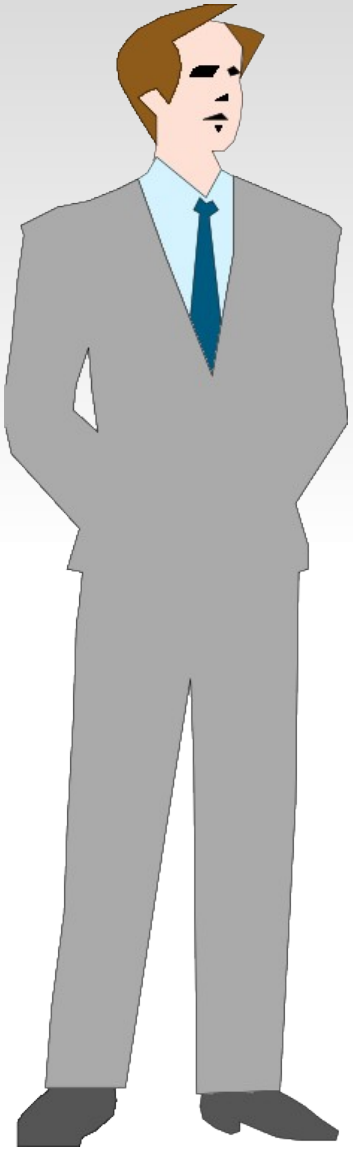
← Zertifikat mit Bobs Public Key

Unverschlüsselte Nachricht, signiert mit Alices Private Key →



← Verschlüsselte Nachricht, verschlüsselt mit Alices Public Key

Verschlüsselte Nachricht, verschlüsselt mit Bobs Public Key, Signiert mit Alices Private Key →



Bob

Fehlertoleranz

Fehler erkennen
Zum Beispiel durch Prüfsummen oder aktive Tests

Fehler maskieren
Fehler verbergen oder abschwächen, zum Beispiel durch wiederholtes Senden

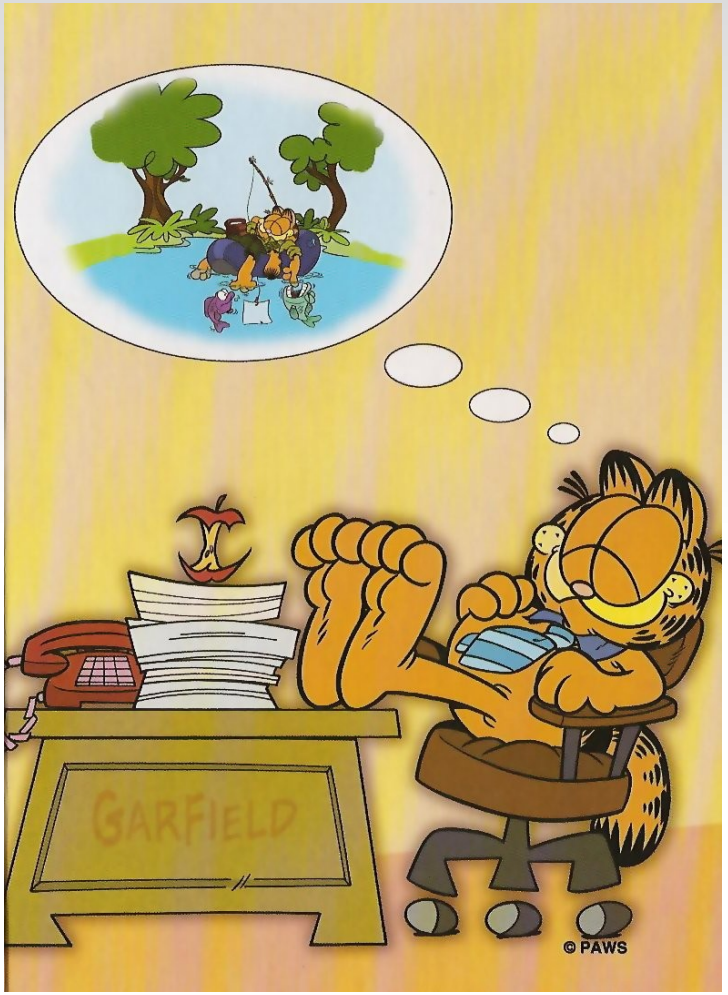
Unkritische Fehler tolerieren

Fehler beheben
Rückkehr in einen sicheren Zustand (Recovery)

Fehler vermeiden
Zum Beispiel durch redundante Datenhaltung oder Hot Standby

Transparenz

In der Wirtschaft: Komplizierte aber unlogische Zusammenhänge können nachgelesen werden.



In der Informatik: „Interessiert mich nicht!“

Transparenz

Ein verteiltes System gilt als transparent wenn …

- Seine Komplexität vor dem Anwender verborgen bleibt,
- Die Verteilung vor dem Anwendungsprogrammierer ebenfalls verborgen bleibt
- Und es somit als Ganzes wahrgenommen werden kann

Acht Arten der Transparenz gemäß ISO:

- Zugriffstransparenz
- Positionstransparenz
- Mobilitätstransparenz
- Nebenläufigkeitstransparenz
- Replikationstransparenz
- Fehlertransparenz
- Leistungstransparenz
- Skalierbarkeitstransparenz

Transparenz verteilter Systeme

Zugriffstransparenz
Zugriff auf lokale und entfernte Ressourcen mit identischen Operationen

Positionstransparenz
Zugriff auf Ressourcen unabhängig von ihrer Position

Mobilitätstransparenz
Erlaubt das Verschieben von Ressourcen als Folge der Positionstransparenz

Nebenläufigkeitstransparenz
Parallele Aktivitäten dürfen sich nicht gegenseitig stören

Replikationstransparenz
Verwendung von redundanten Ressourcen ohne besondere Berücksichtigung durch Anwender oder Entwickler

Fehlertransparenz
Kleinere Fehler dürfen das System nicht beeinträchtigen

Leistungstransparenz
Ermöglicht die Rekonfiguration des Systems zur Leistungssteigerung

Saklierbarkeitstransparenz
Vergrößerung des Systems, ohne dass Anpassungen Struktur und Algorithmen notwendig werden

Architekturmodelle

Ein Architekturmodell beschreibt …

- Die Rolle der Anwendungskomponenten und
- Die Beziehung der Komponenten zueinander

Eigentlich nur drei Arten von Komponenten

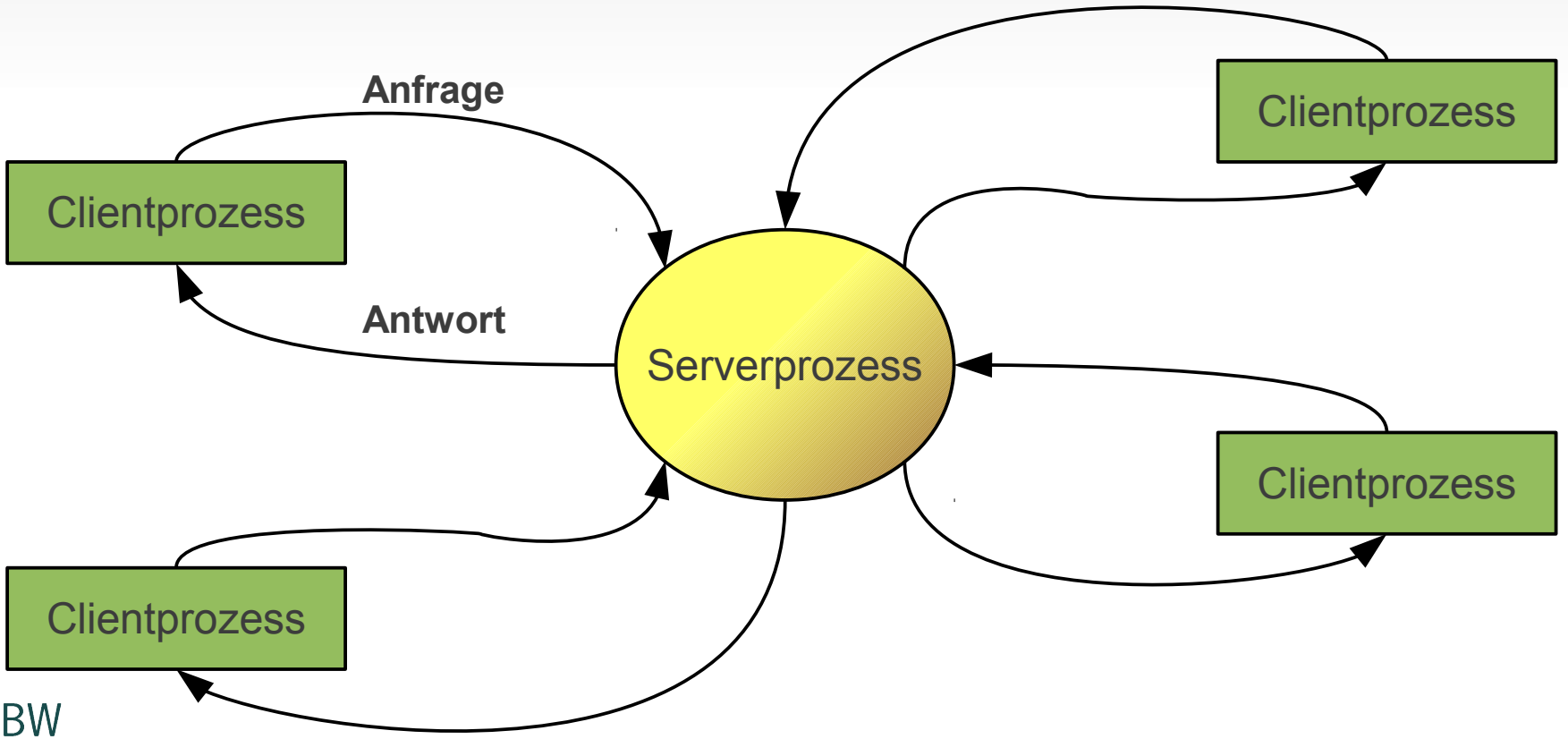
- **Clientprozesse:** Leben nur für die Dauer einer Sitzung. Sie werden vom Anwender gestartet und stoßen die Kommunikation an
- **Serverprozesse:** Beantworten die Anfragen mehrerer Clients und leben daher länger als eine Sitzung
- **Peerprozesse:** Kurzlebige, vom Anwender kontrollierte Prozesse, die sowohl Client als auch Server sein können

Model: Client/Server

Kurzlebige Clientprozesse initiieren die Kommunikation

Ein langlebiger Serverprozess beantwortet die Anfragen

Trifft auf die meisten, verteilten Anwendungen zu

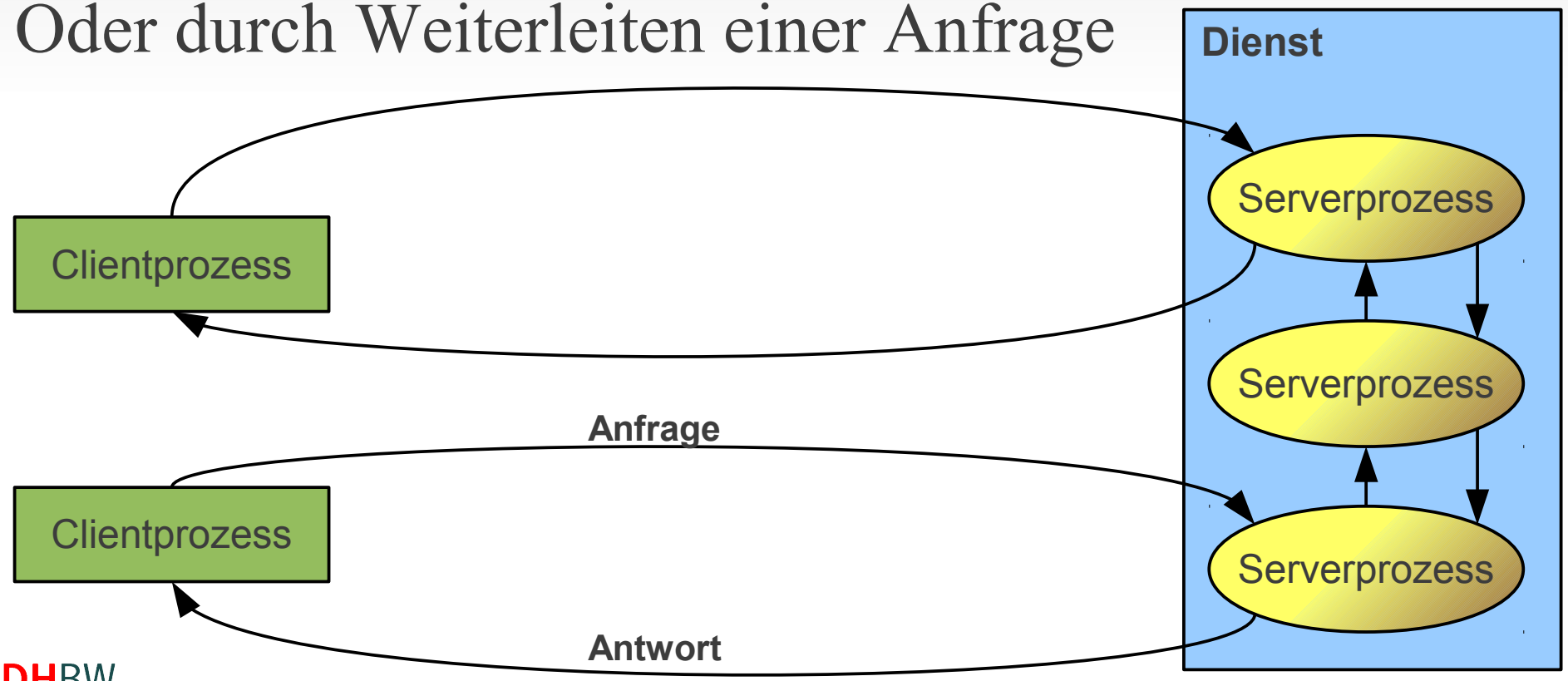


Model: Kooperierende Server

Transparente Bearbeitung der Anfragen durch einen Serververbund, ohne dass der Client dies weiß

Zum Beispiel bei Lastverteilung durch Replikation

Oder durch Weiterleiten einer Anfrage

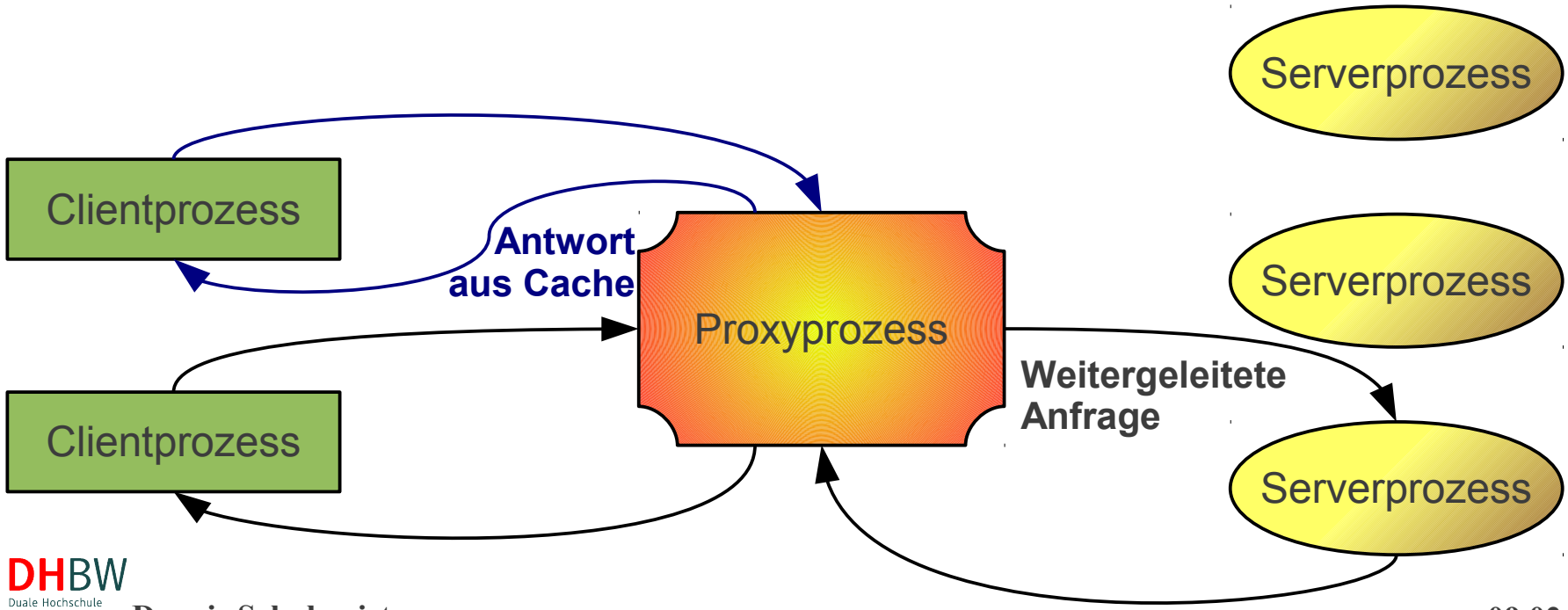


Model: Proxyserver

Anfragen werden nicht direkt an den Server sondern an einen zwischengeschalteten Proxyserver gestellt

Antworten kommen vom Proxy oder vom Zielserver

Zum Beispiel: Caching von Zwischenergebnissen, Filtern unerwünschter Inhalte, Zugangsbeschränkungen



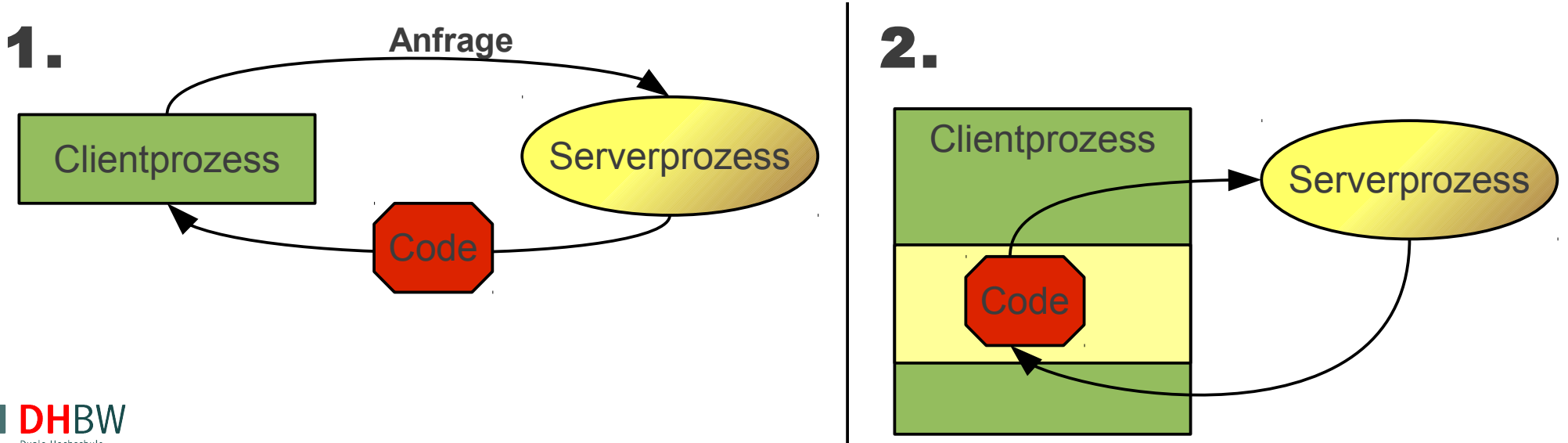
Model: Mobiler Code

Ein Server liefert mobilen Code an den Client aus

Der Client führt den Code in einer Sandbox aus

Der Code nimmt weiteren Kontakt zum Server auf

Zum Beispiel: Java Applets, Adobe Flash, AJAX, …

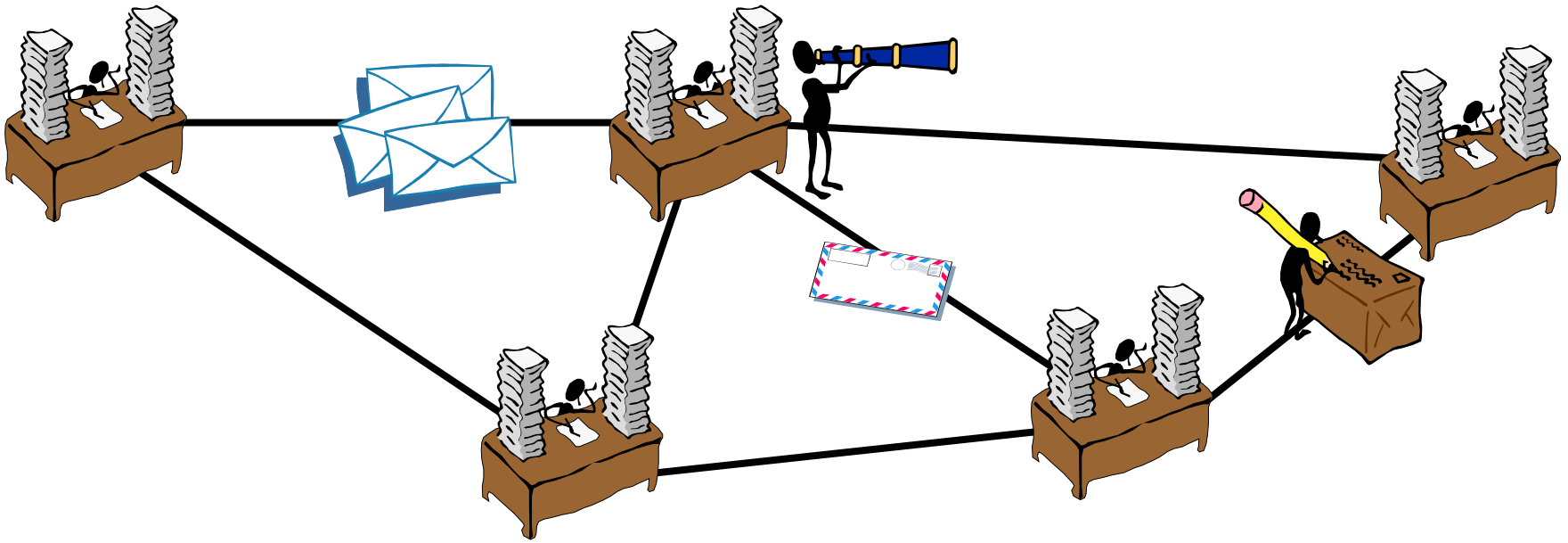


Model: Peer-To-Peer

Jeder Prozess ist sowohl Client als auch Server

Kein Teilnehmer kennt alle beteiligten Knoten

Peers kommen und gehen in zufälligen Intervallen

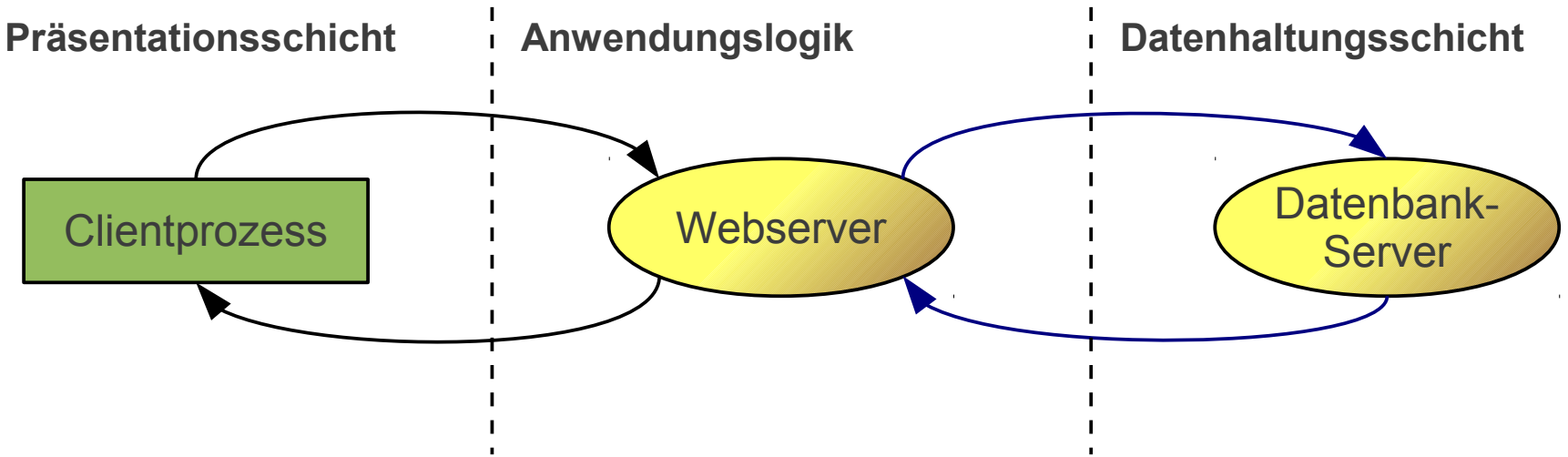


Mehrschichtenmodelle

Verallgemeinerung des Client/Server-Modells

Aufteilung einer Anwendung in mehrere Schichten

Der Server greift auf einen anderen Server zurück, um eine Anfrage des Clients bearbeiten zu können



Mehrschichtenmodelle

Beliebige Teilung der Verantwortlichkeiten möglich

- **Zwei Schichten:** Client/Server
- **Drei Schichten:** Präsentation, Anwendung, Datenbank
- **Vier Schichten:** Aufteilung der Anwendungsschicht
- Sonstige n-Tier Modelle

Anwendungsschicht kann sehr umfangreich werden

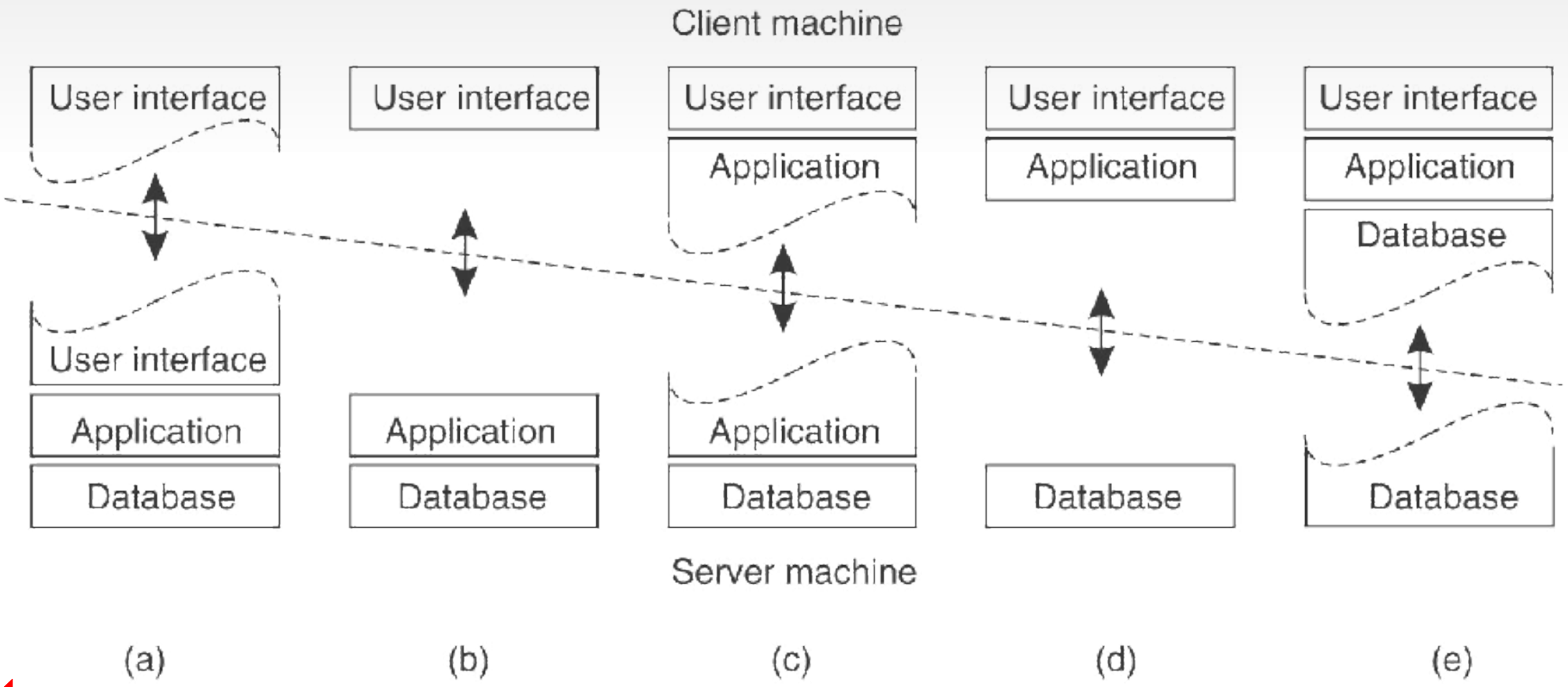
Clients sind heute häufig einfache Webbrowser

Früher waren jedoch spezielle Clientprogramme nur für die Darstellung sehr beleibt, z.B. SAPGUI

Mehrschichtenmodelle

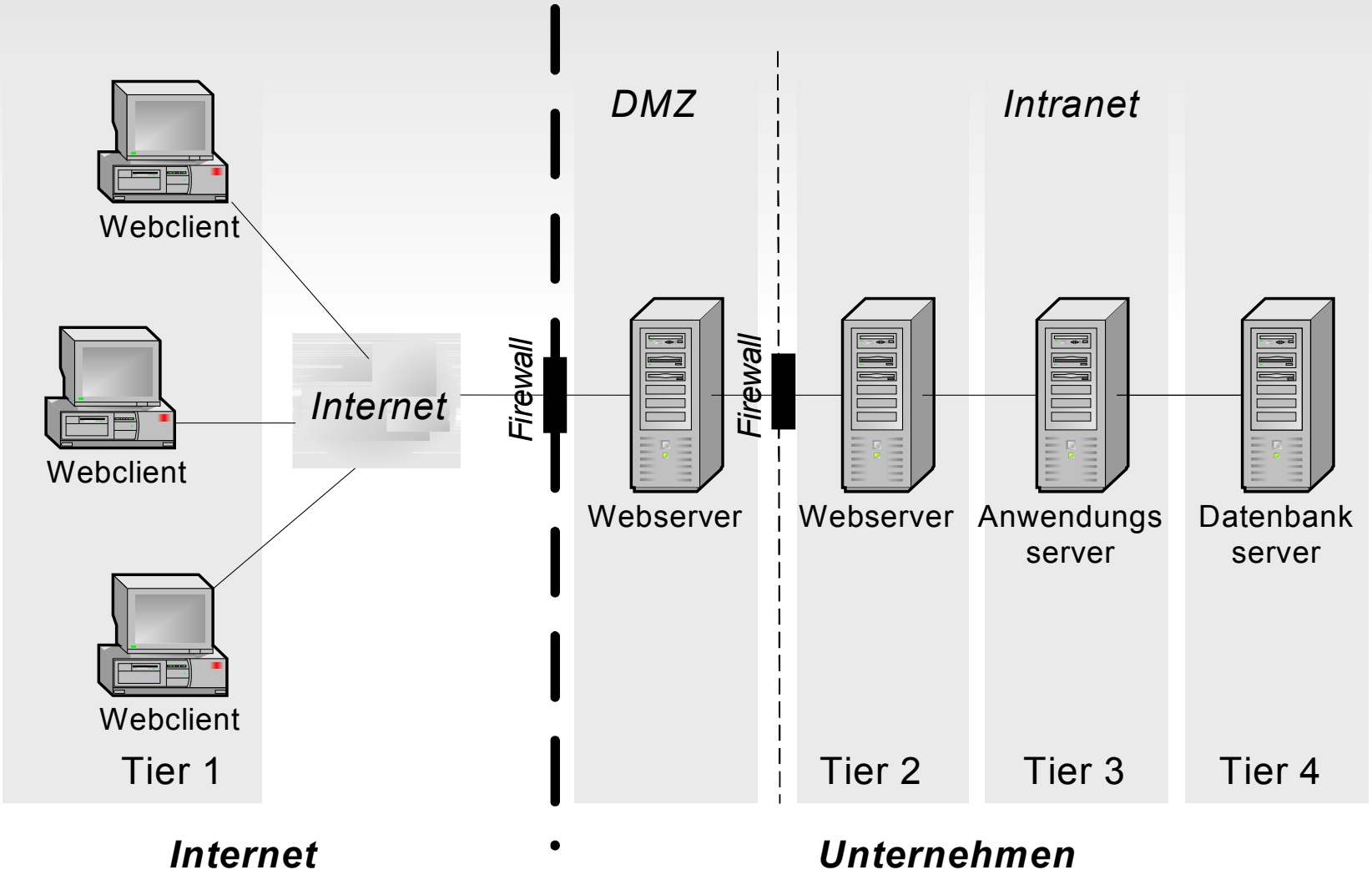
Historische Entwicklung von Client und Server

Von rechts nach links: **Ultra Fat Client** → **Ultra Thin Client**



Mehrschichtenmodelle

Zugriff auf Unternehmensdaten über das Internet



Programmierparadigmen verteilter Anwendungen

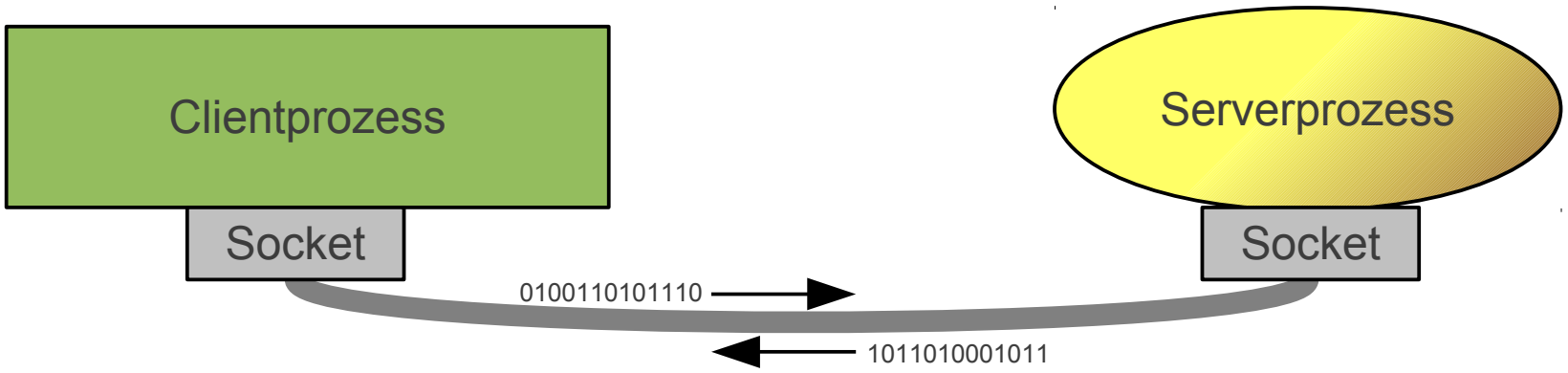
Streams und Sockets

Fester Bestandteil der Java Klassenbibliothek

- Klassen **`ServerSocket`** und **`Socket`**
- Klassen **`Input/OutputStream`**, **`Reader`** und **`Writer`**

Sehr gute Low Level-Optimierungsmöglichkeiten

Allerdings auch sehr hoher Programmieraufwand



Middleware

Befindet sich zwischen Anwendung und System

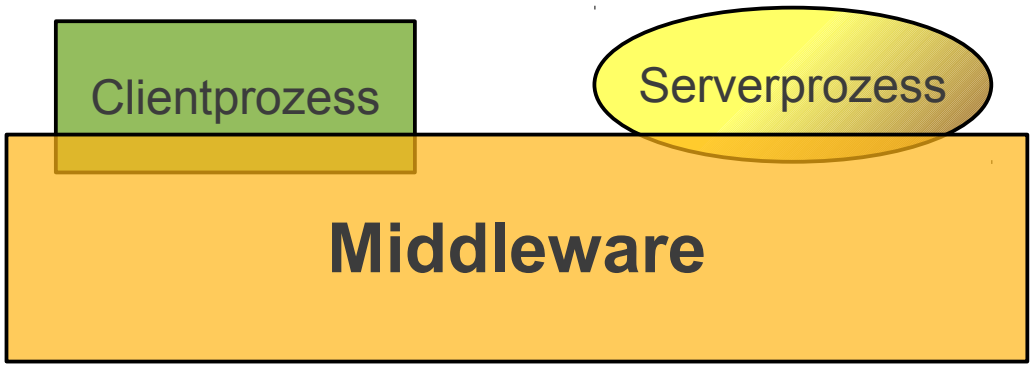
Abstrahiert das Netzwerk gegenüber der Anwendung

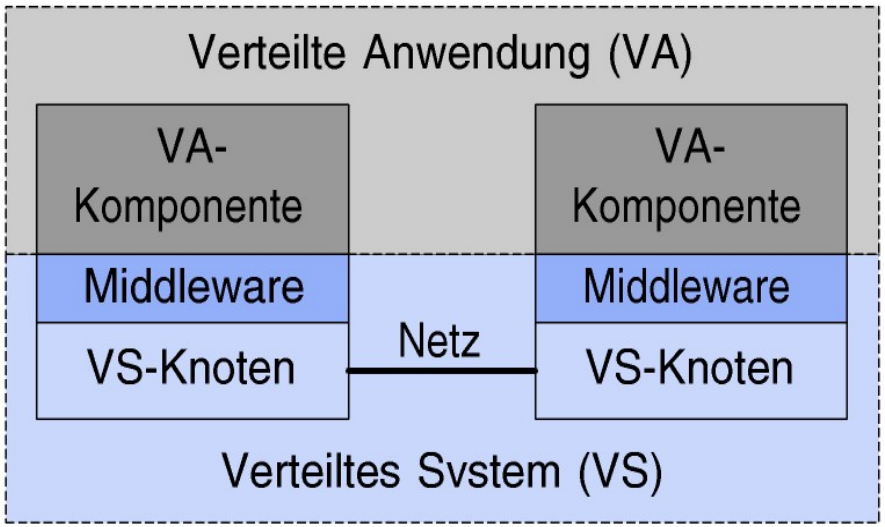
Somit keine direkte Netzwerk-programmierung notwendig

Kaum Optimierungspotential

Zusatzdienste je nach Produkt:

- Namens/Verzeichnisdienst
- Authentifizierung
- Transaktionsdienst
- Datenhaltung
- …





Relevante Middlewareprodukte

CORBA

Herstellerunabhängiger Standard, der für viele Betriebssysteme und Programmiersprachen zur Verfügung steht.

Sehr großer Funktionsumfang, dafür kompliziert in der Programmierung. War vorwiegend in den 1990ern beliebt.

Sun/Oracle RMI

Im Lieferumfang des JDK enthalten. An Java gebunden.

Webservices

Verschiedene Standards basierend auf HTTP und XML, die mit jeder Sprache genutzt werden können. Beschränken sich auf den Austausch einfacher XML-Nachrichten.

Applikationsserver

Bieten den größten Funktionsumfang, sind jedoch oft an einen Hersteller und eine Programmiersprache gebunden. Beliebt bei großen Java und .NET-Anwendungen.

Klassifizierung von Middleware

Kommunikationsorientierte Middleware

- Setzt direkt auf den Netzwerkdiensten des Systems auf
- Vereinheitlicht das Datenformat verschiedener Systeme
- Stellt eine einheitliche Kommunikationsinfrastruktur bereit
- Bietet verschiedene Paradigmen zur Programmierung an

Anwendungsorientierte Middleware

- Erweitert die kommunikationsorientierte Middleware
- Beinhaltet eine **Laufzeitumgebung** für echte **Komponenten**
- Stellt der Anwendung verschiedene Zusatzdienste bereit

Aufgaben der Middleware

Bereitstellen der Kommunikationsinfrastruktur

- Lokalisierung und Identifikation aller Systemknoten
- Bereitstellung eines einheitlichen Datenformats
- Definition eines gemeinsamen Netzwerkprotokolls
- Behandlung technischer Kommunikationsfehler
- Unterstützung bei der Thread- und Prozessverwaltung

Typische Programmierparadigmen

- **Prozedural & Synchron:** Entfernter Prozeduraufruf, Webservices
- **Objektorientiert & Synchron:** Entfernte Methodenaufrufe
- **Asynchron:** Nachrichtenaustausch mit Warteschlangen

Einheitliche Datenformate

Problem: Heterogene Hardware, Betriebssysteme und Sprachen

- Unterschiedliche Zahlendarstellung (Little Endian vs. Big Endian)
- Verschiedene Zeichensätze (EBCDIC, Latin-1, UTF-8, …)
- Andere Darstellung von Daten und Objekten im Hauptspeicher

Zwei Arten der Behandlung durch die Middleware:

- Verwendung eines einheitlichen, externen Datenformats
- Mitsenden aller relevanten Typinformationen im Datenstrom

Beispiele für Externe Datenformate: XML, Corba CDR, …

Beibehaltung der Typinformationen: Java Objektserialisierung

Einheitliche Datenformate

THERE'S BEEN A LOT OF CONFUSION OVER 1024 vs 1000, KBYTE vs KBIT, AND THE CAPITALIZATION FOR EACH.

HERE, AT LAST, IS A SINGLE, DEFINITIVE STANDARD:

SYMBOL	NAME	SIZE	NOTES
kB	KILOBYTE	1024 BYTES OR 1000 BYTES	1000 BYTES DURING LEAP YEARS, 1024 OTHERWISE
KB	KELLY-BOOTLE STANDARD UNIT	1012 BYTES	COMPROMISE BETWEEN 1000 AND 1024 BYTES
KiB	IMAGINARY KILOBYTE	$1024\sqrt{-1}$ BYTES	USED IN QUANTUM COMPUTING
kb	INTEL KILOBYTE	1023.937528 BYTES	CALCULATED ON PENTIUM F.P.U.
Kb	DRIVEMAKER'S KILOBYTE	CURRENTLY 908 BYTES	SHRINKS BY 4 BYTES EACH YEAR FOR MARKETING REASONS
KBa	BAKER'S KILOBYTE	1152 BYTES	9 BITS TO THE BYTE SINCE YOU'RE SUCH A GOOD CUSTOMER

Entfernte Aufrufe

Vereinfachung des Client/Server-Modells

Je nach Programmiersprache des Servers

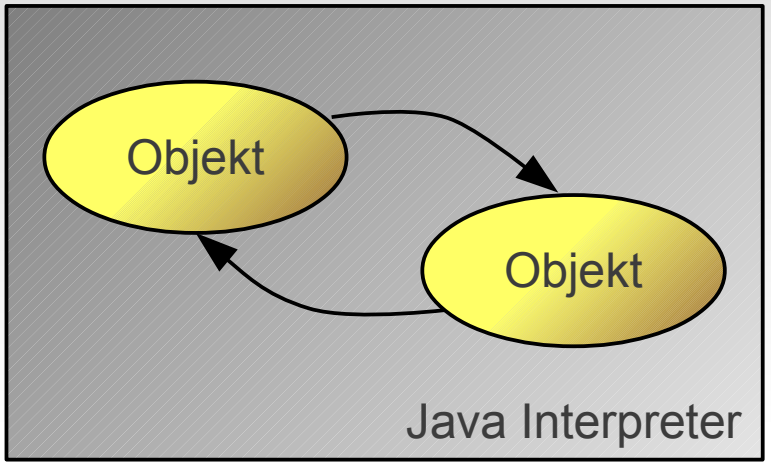
- **Remote Function Call** oder
- **Remote Method Invocation**

Der Client ruft einfach Funktionen oder Methoden auf, die auf einem anderen Rechner laufen

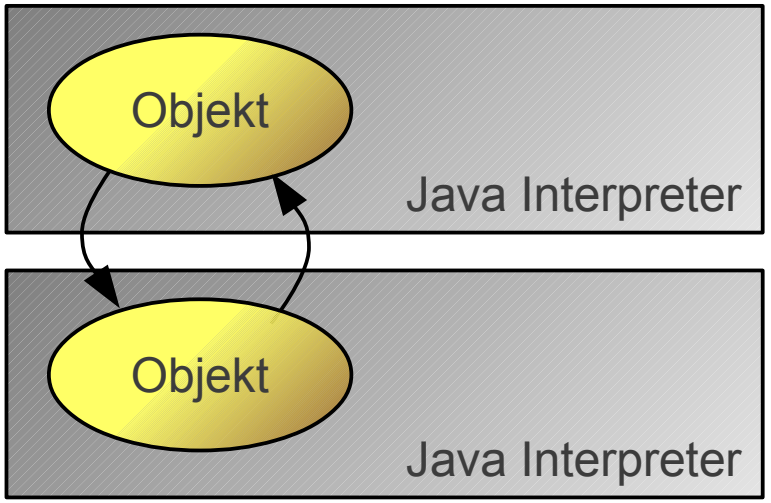
Die Datenübertragung erfolgt durch die Middleware, ohne Zusatzprogrammierung für Client oder Server

Dabei wartet der Client auf die Antwort des Servers (Synchroner Aufruf)

Normaler Methodenaufruf



Entfernter Methodenaufruf



Ablauf entfernter Aufrufe

Client

Server

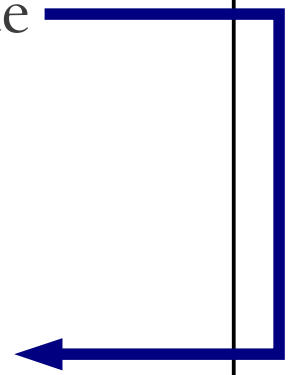
Serverprogramm starten

Serverprogramm meldet die entfernt aufrufbaren Funktionen/Methoden bei der Middleware an

Clientprogramm starten

Clientprogramm fragt die Middle-ware, auf welchem Server eine Funktion/Methode bereitsteht

Aufruf der Funktion/Methode



Client Wartet ...

Funktion/Methode wird ausgeführt

Clientprogramm läuft weiter

Asynchroner Nachrichtenaustausch

Asynchroner Aufruf, ähnlich wie E-Mailversand

- Sender übergibt eine Nachricht an die Middleware
- Anschließend wartet der Sender nicht auf eine Antwort
- Die Middleware ordnet die Nachricht in eine Warteschlange ein
- Empfänger rufen die Nachricht aus der Warteschlange ab
- Zwei Arten von Warteschlangen: Mit nur einem Empfänger oder mit mehreren Empfängern ähnlich einer Mailingliste

Zwei Arten der Nachrichtenzustellung

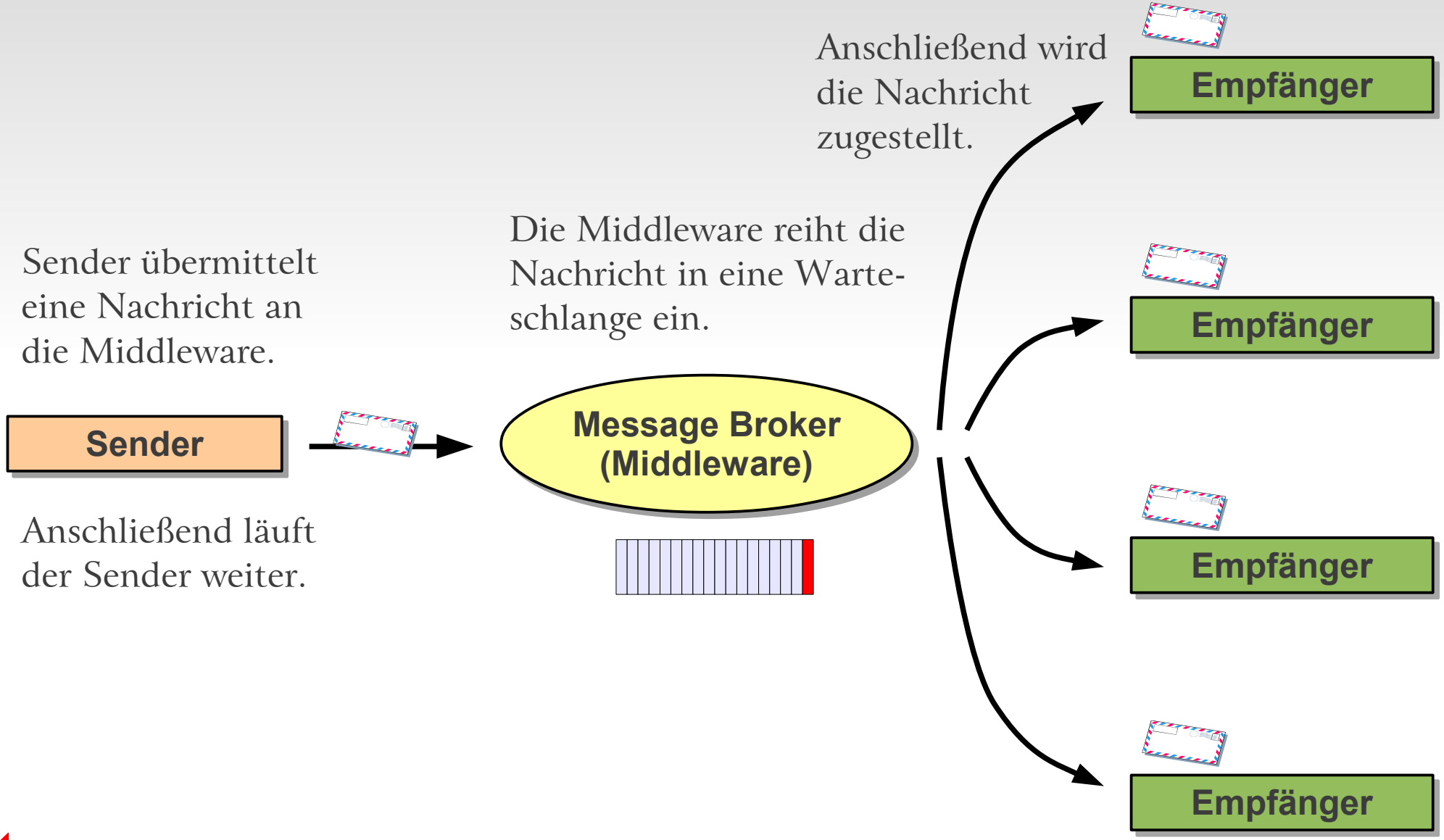
- Empfänger prüfen regelmäßig ihre Warteschlangen (Pull)
- Middleware ruft regelmäßig die Empfänger auf (Push)

Beispiel: Nachrichtenversand

Sender übermittelt eine Nachricht an die Middleware.

Die Middleware reiht die Nachricht in eine Warte-schlange ein.

Anschließend wird die Nachricht zugestellt.



Anschließend läuft der Sender weiter.

Laufzeitumgebung

Kommunikationsorientierte Middleware

- Bietet kaum mehr als Kommunikationsfunktionen
- Lässt sich einfach in bestehende Programme einbinden
- Keine Unterstützung beim Schreiben neuer Programme
- Besteht meist aus einer einfachen Klassenbibliothek

Anwendungsorientierte Middleware

- Bietet eine Laufzeitumgebung für Softwarekomponenten
- Die meiste Anwendungslogik steckt in der Middleware
- Nur die fehlende Fachlogik muss selbst entwickelt werden

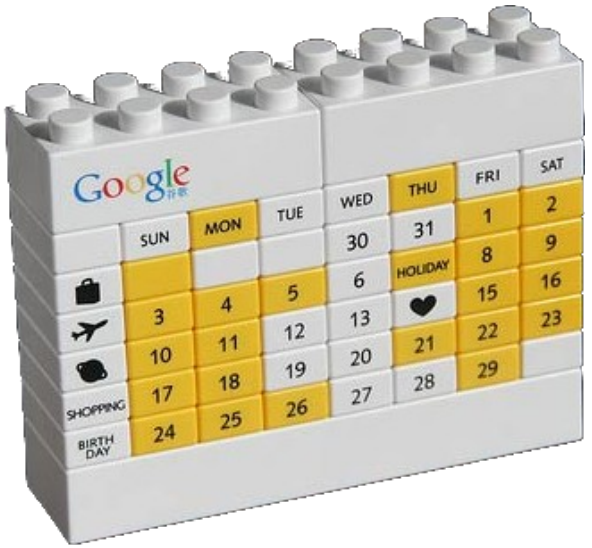
Laufzeitumgebung

Jede Softwarekomponente

- Kapselt eine spezielle Funktion der Anwendung und
- Ist darum eine einfache Klasse,
- Die eine vorgegebene Schnittstelle umsetzt,
- Damit sie automatisch erzeugt und zerstört werden kann

Die Laufzeitumgebung

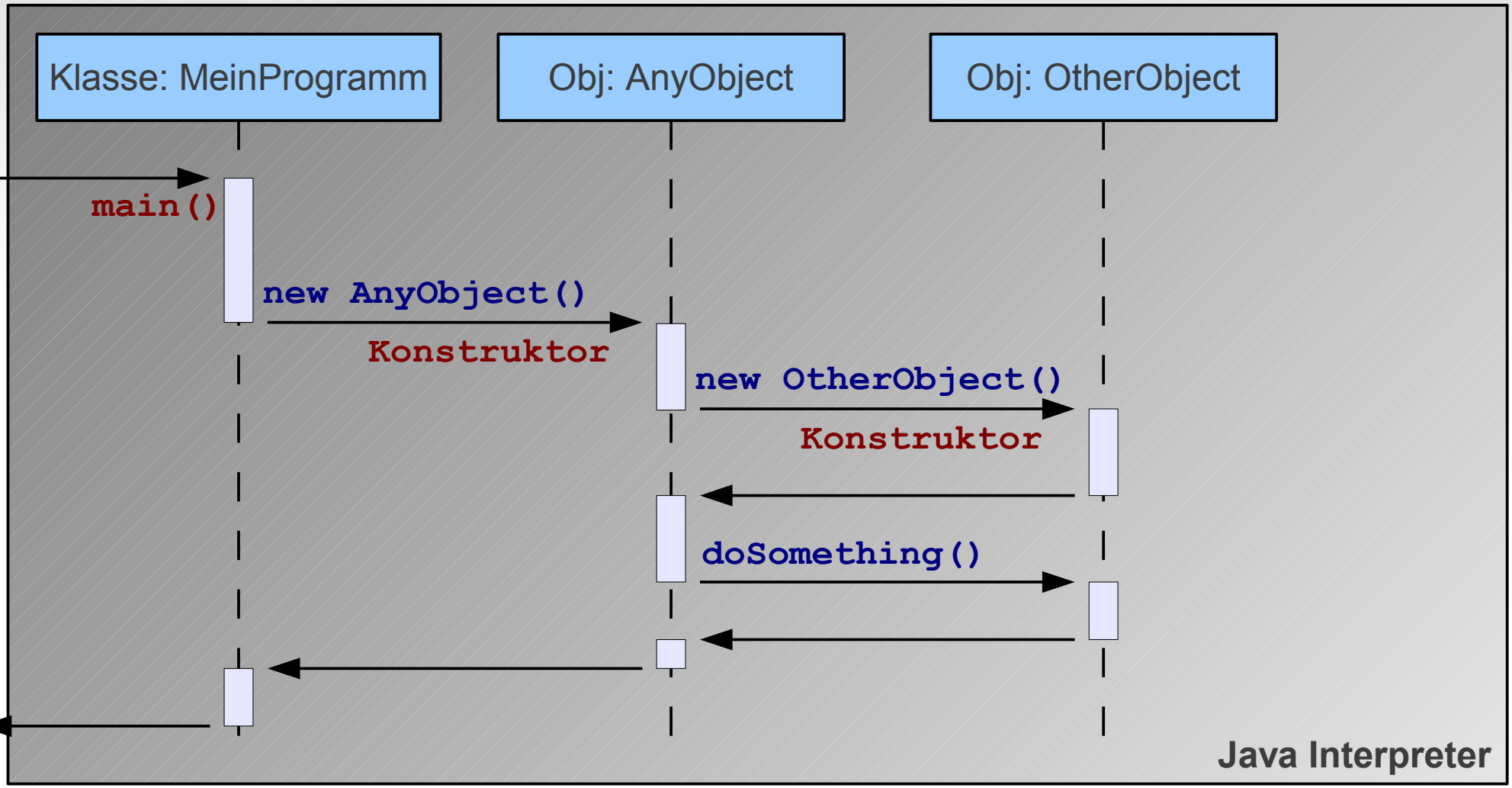
- Häufig auch **Container** genannt,
- Ist ein bereits fertiges Programm,
- Das durch die Komponenten erweitert wird
- Und Komponentenobjekte erzeugt/zerstört



Beispiel: Normale Anwendung

Anwender startet das Programm:

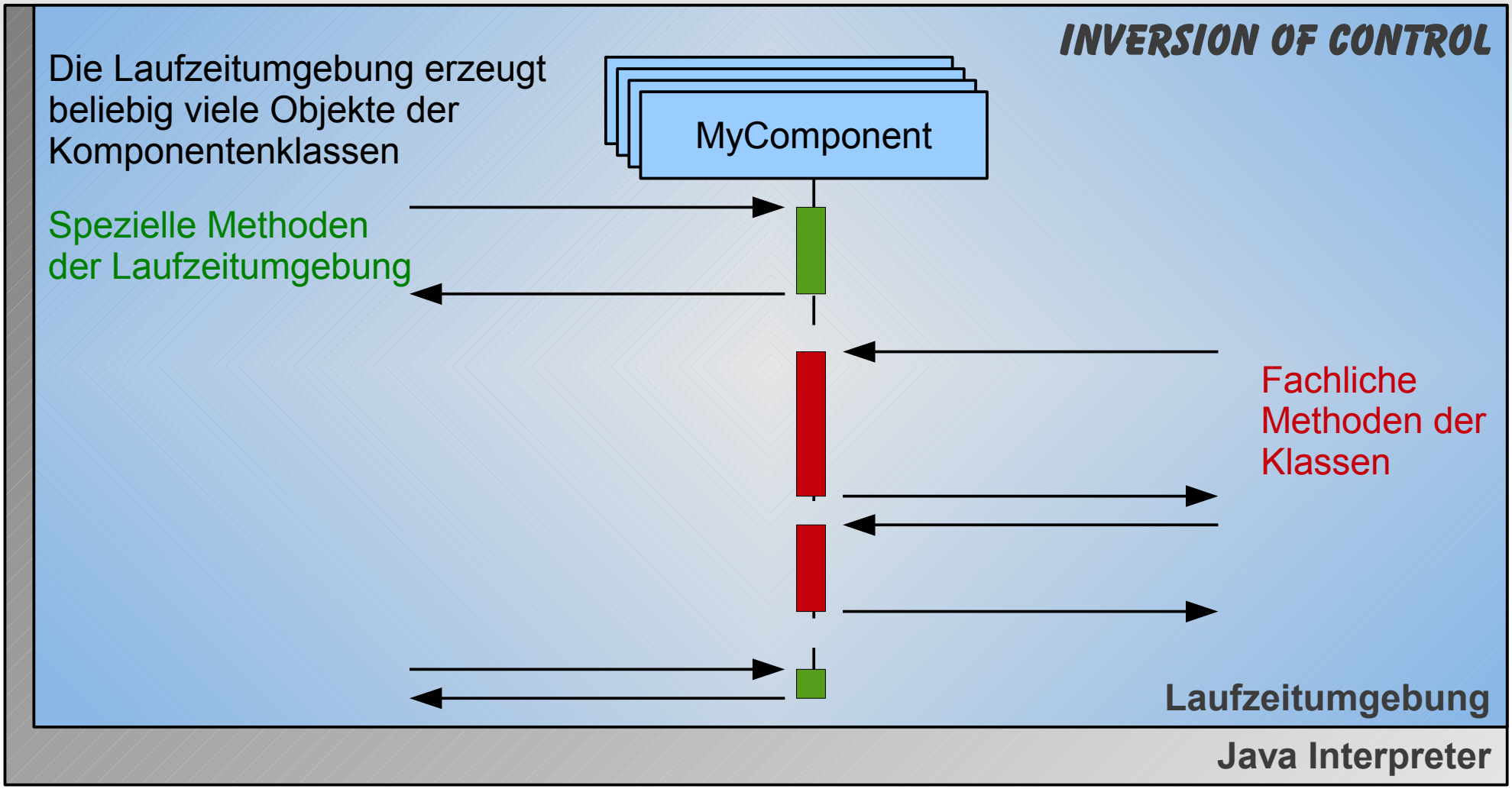
```
dennis@localhost:~$ java MeinProgramm
```



Beispiel: Softwarekomponenten

Anwender startet die Laufzeitumgebung

```
dennis@localhost:~$ jboss/bin/run.sh
```



Fünf sichere Anzeichen für Softwarekomponenten

Sie starten Ihr Programm nicht mit `java Klassenname`

Stattdessen müssen Sie ein völlig anderes Programm aufrufen

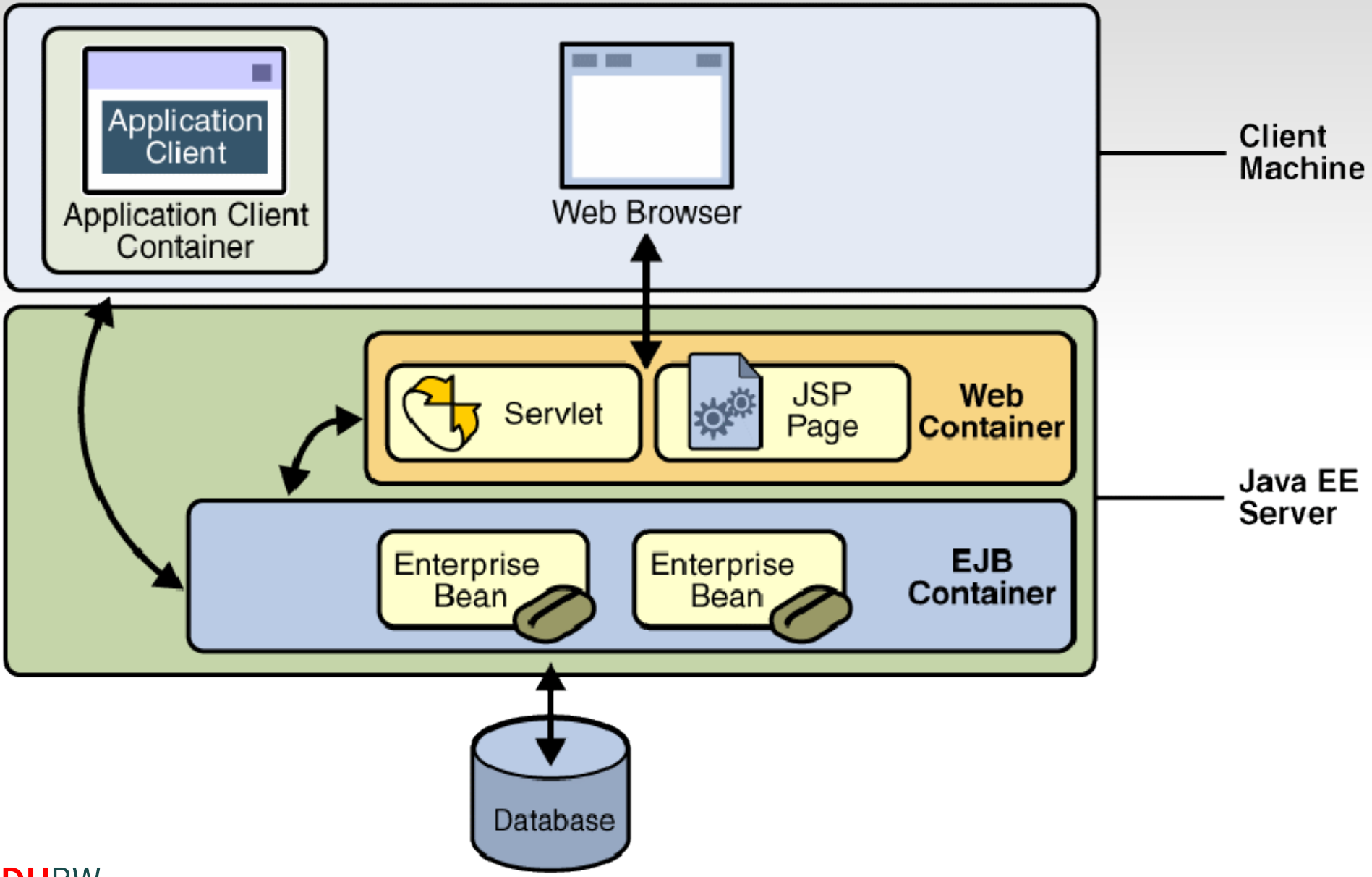
Ihre Klassen implementieren mindestens zwei Schnittstellen:

- Eine mit den fachlichen Methoden der Klasse und
- Eine mit speziellen Methoden der Laufzeitumgebung

Sie wissen gar nicht, dass Sie gerade ein Programm schreiben

Denn ihr Programm besitzt gar keine `main()`-Methode

Java Enterprise Edition



Bestandteile von Java EE

Webprogrammierung
Servlets
Java Server Pages
Java Server Faces

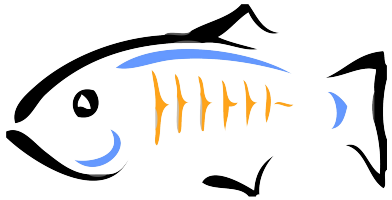
Entfernter Prozeduraufruf
JAX-RPC
JAX-WS

Entfernte Methodenaufrufe
Enterprise Java Beans

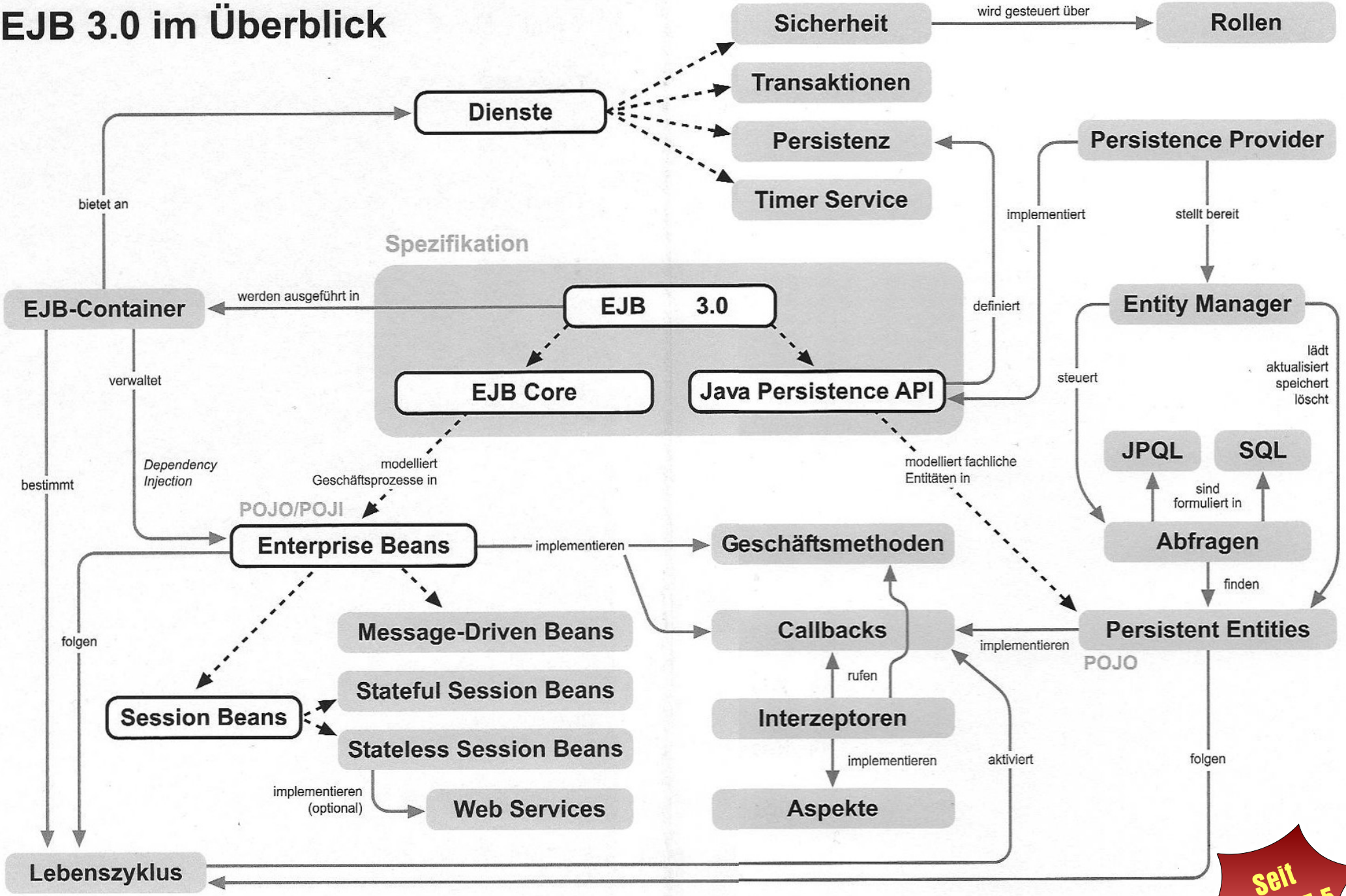
Nachrichtenaustausch
Java Messaging Services

Persistenzdienst
Java Database Connection
Java Persistence API
Java Transaction API

Verzeichnisdienst
Java Naming and Directory Interface



EJB 3.0 im Überblick



Welche Technik für wen?

Einfache Client/Server-Anwendungen
Spezialanwendungen mit hohem Optimierungsbedarf
Anwendungen mit geringen Abhängigkeiten zu externen Bibliotheken

Streams und Sockets

Kommunikationsorientierte Middleware

Webservices

Applikationsserver

Komplexe Informationssysteme
Betriebswirtschaftliche Großanwendungen
Anwendungen mit geringem Optimierungsbedarf