

Verteilte Systeme 2

Prinzipien und Paradigmen

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

Kapitel 02: Architekturen

Version: 16. 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

Architektur und Architekturstil

- **Architektur**: Entwurf einer Problemlösung in Bezug auf gegebene Bedingungen/Einschränkungen. 💬
- **Architekturstil**: Generelle Prinzipien, die die Gestaltung von Architekturen beeinflussen. 💬



- **Architektur**: Louvre
- **Architektur-Stil**: Barock




- **Architektur**: Villa Savoye
- **Architektur-Stil**: Moderne

Architekturstile

Grundidee

Ein Stil drückt sich aus durch

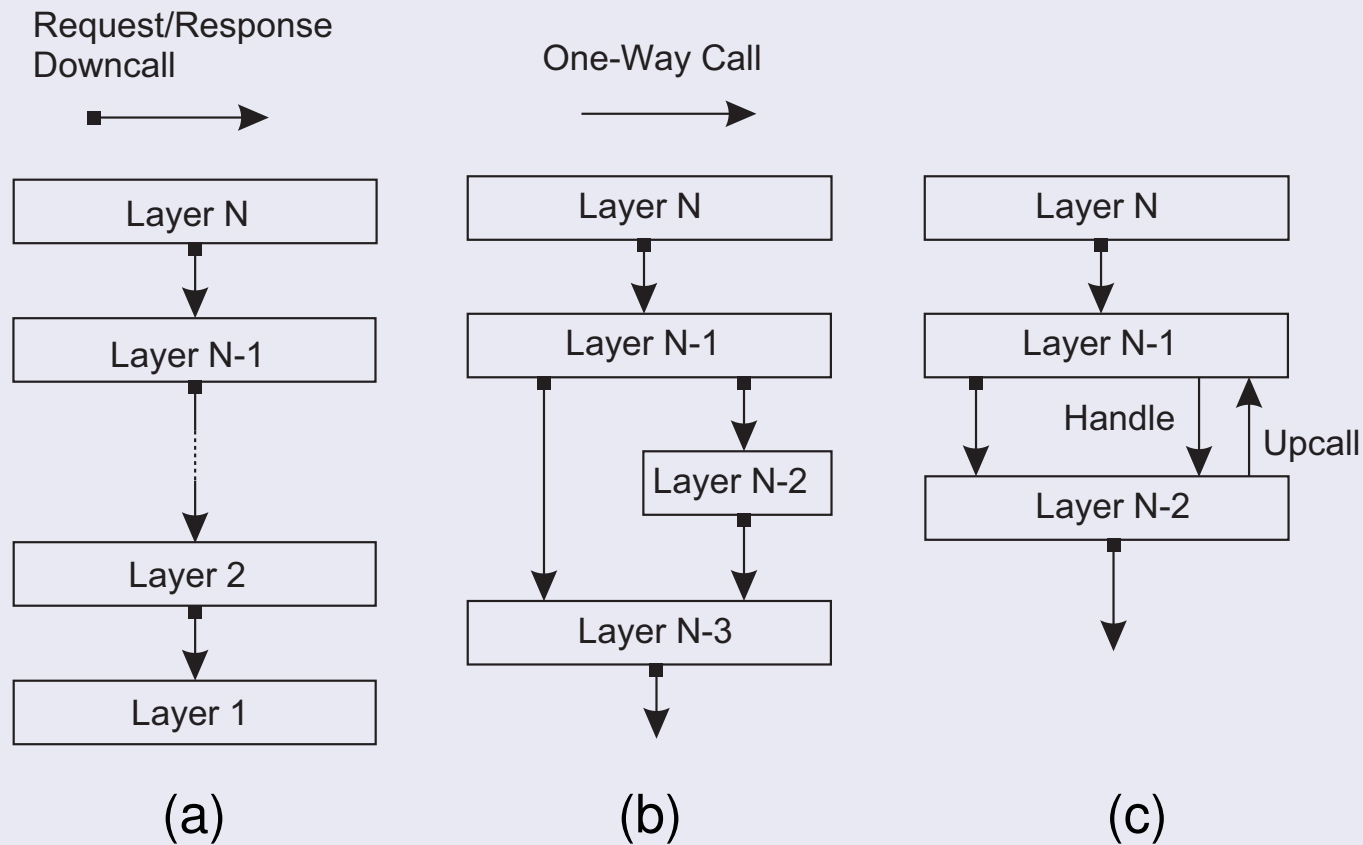
- *Komponenten* mit wohldefinierten Schnittstellen
- die Art, wie Komponenten *verbunden* sind
- die *Daten*, die die Komponenten austauschen 
- die Art, wie Komponenten und Konnektoren gemeinsam zu einem System *konfiguriert* werden.

Konnektor

Mechanismus zur Vermittlung von Kommunikation, Koordination oder Kooperation zwischen Komponenten. **Beispiel:** Systemfunktionen für (entfernten) Prozeduraufruf, Messaging oder Streaming.

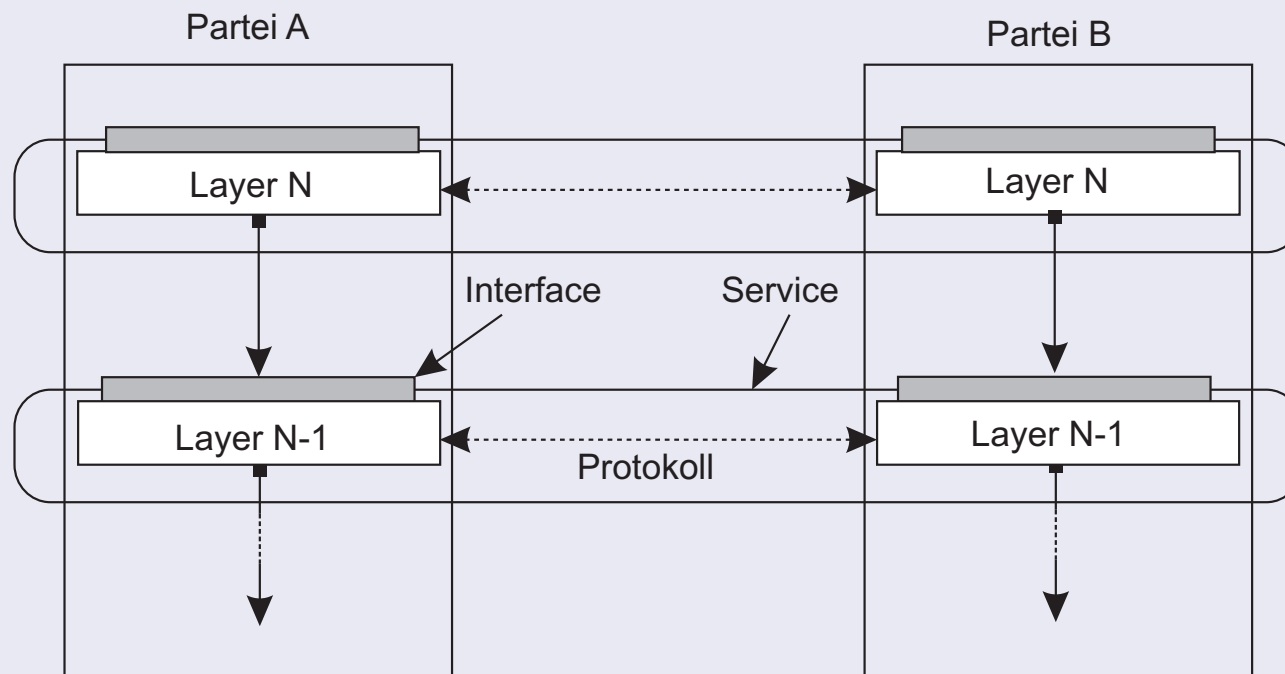
Geschichtete Architektur

Varianten der Organisation von Schichten (Layer)



Beispiel: Kommunikationsprotokolle

Protokoll, Service, Interface



Zwei-Parteien Kommunikation

Server

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 (conn, addr) = s.accept() # returns new socket and addr. client
4 while True:               # forever
5     data = conn.recv(1024) # receive data from client
6     if not data: break     # stop if client stopped
7     conn.send(str(data)+"*") # return sent data plus an "*"
8 conn.close()              # close the connection
```

Client

```
1 from socket import *
2 s = socket(AF_INET, SOCK_STREAM)
3 s.connect((HOST, PORT)) # connect to server (block until accepted)
4 s.send('Hello, world') # send some data
5 data = s.recv(1024)    # receive the response
6 print(data)            # print the result
7 s.close()              # close the connection
```

Schichtung von Anwendungen

Traditionelle dreischichtige Sicht

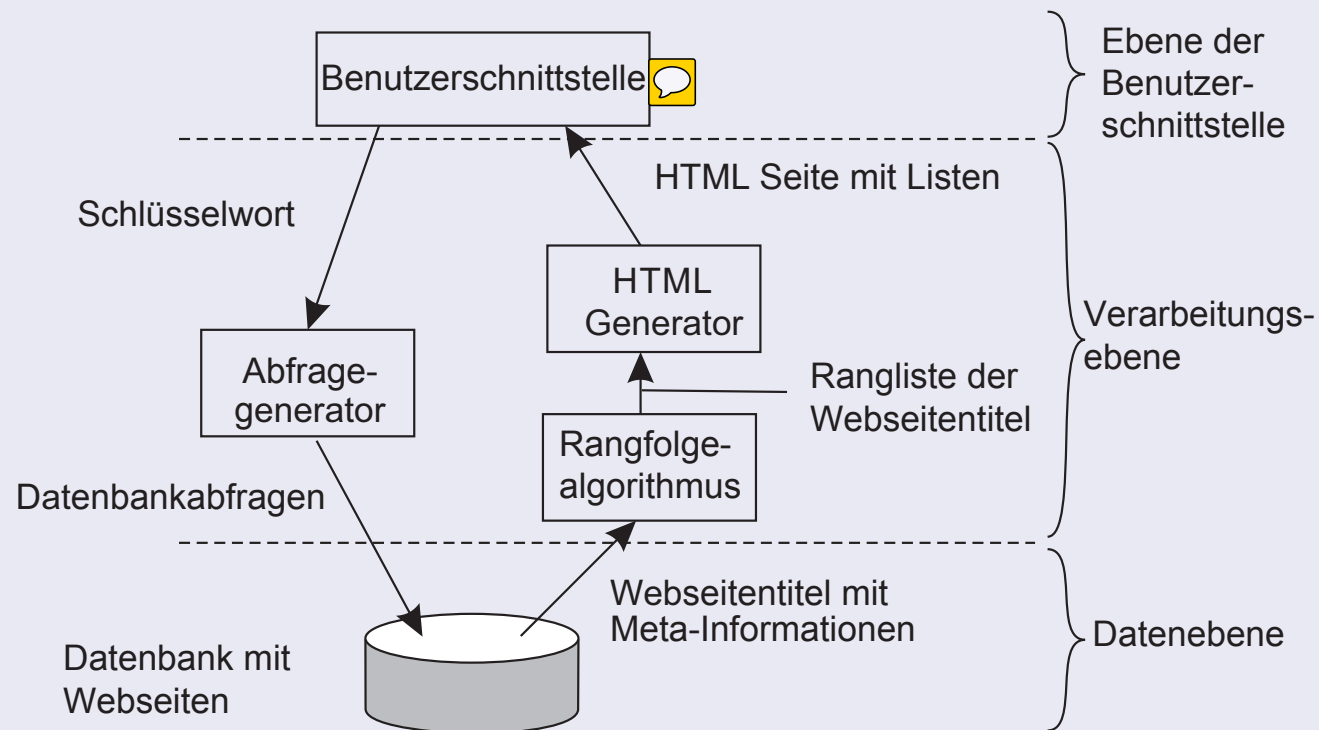
- Die **Ebene der Benutzerschnittstelle** enthält Komponenten zur Benutzerinteraktion.
- Die **Verarbeitungsebene** enthält die Funktionen einer Anwendung ohne spezifische Daten.
- Die **Datenebene** enthält die Daten, die Nutzer über Anwendungskomponenten manipulieren möchten.

Beobachtung

Diese Schichtung findet sich in vielen verteilten Informationssystemen, die traditionelle Datenbanken und entsprechende Anwendungen nutzen.

Schichtung von Anwendungen

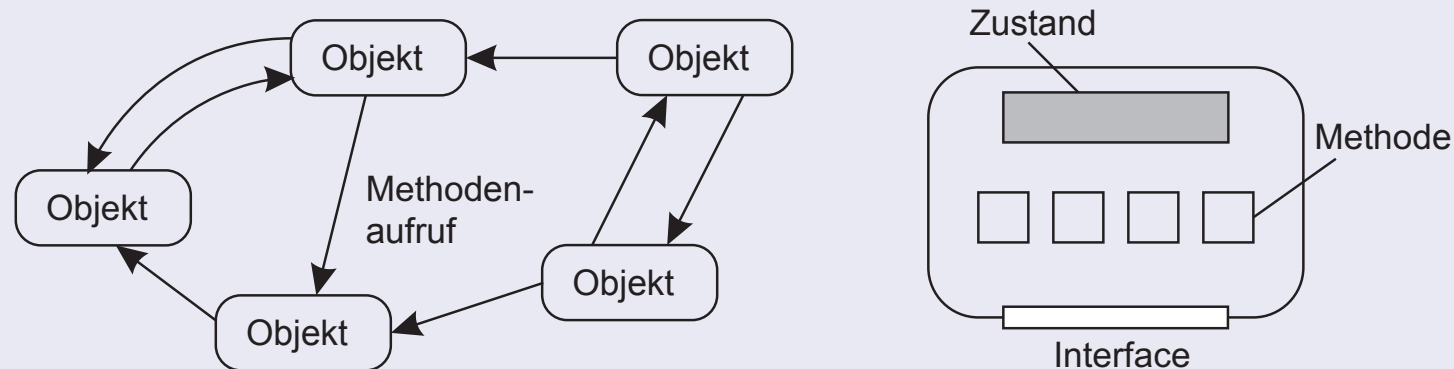
Beispiel: einfache Suchmaschine



Objektbasierter Stil

Im Kern

Komponenten sind Objekte, verbunden durch Prozeduraufrufe. Sie liegen ggf. auf verschiedenen Maschinen und Aufrufe können über das Netzwerk gehen. 💬




Kapselung

Man sagt, Objekte **kapseln Daten** und erlauben **Methoden auf diesen Daten** ohne ihre interne Implementierung offenzulegen.

REST-Architekturstil

Im Kern

Verteiltes System als Sammlung von Ressourcen, die einzeln von Service-Komponenten verwaltet werden. Ressourcen können von Anwendungen hinzugefügt, entfernt, abgerufen und geändert werden.

- 1 Ressourcen werden über gemeinsames Namensschema identifiziert 
- 2 Alle Dienste bieten die gleiche Schnittstelle
- 3 Nachrichten an oder von einem Dienst sind selbstbeschreibend
- 4 Nach Ausführung einer Operation bei einem Dienst vergisst die Komponente alles über den Aufrufer


Basic operations

Operation	Description
PUT	Erzeuge neue Ressource
GET	Rufe Zustand von Ressource (Repräsentation) ab
DELETE	Entferne Ressource
POST	Ändere Ressource durch Transfer von neuem Zustand

Beispiel: Amazon Simple Storage Service (S3)

Im Kern

Objects (d.h., Dateien) platziert in **Buckets** (d.h., Ordner). Buckets können keine Buckets enthalten. Operationen auf `ObjectName` im Bucket `BucketName` erfordern folgenden Bezeichner:

`http://BucketName.s3.amazonaws.com/ObjectName` 

Typische operationen

Alle Operationen basieren auf HTTP Requests:

- Erzeuge Bucket/Object: `PUT`, mit URI
- Objekte aufzählen: `GET` auf Bucket Namen
- Objekt lesen: `GET` auf URI

Über Interfaces

Problem

Entwickler mögen REST-Ansätze, weil die Schnittstelle zu einem Service einfach ist. Der Haken: viel **Parametrisierung**

Amazon S3 SOAP Interface

Bucket Operationen	Object Operationen
ListAllMyBuckets	PutObjectInline
CreateBucket	PutObject
DeleteBucket	CopyObject
ListBucket	GetObject
GetBucketAccessControlPolicy	GetObjectExtended
SetBucketAccessControlPolicy	DeleteObject
GetBucketLoggingStatus	GetObjectAccessControlPolicy
SetBucketLoggingStatus	SetObjectAccessControlPolicy

Über Interfaces

Vereinfachung

Angenommen Interface `bucket` hat Operation `create` mit String Parameter (z.B. `mybucket` erzeugt Bucket "mybucket").

SOAP

```
import bucket  
bucket.create("mybucket")
```

REST

```
PUT "http://mybucket.s3.amazonsws.com/"
```

Folgerung



Gibt es eine?



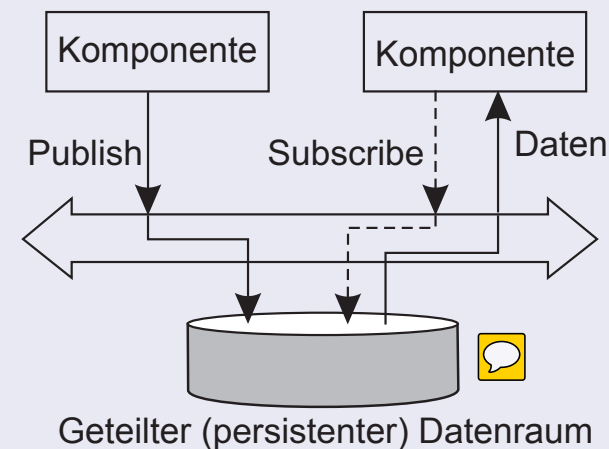
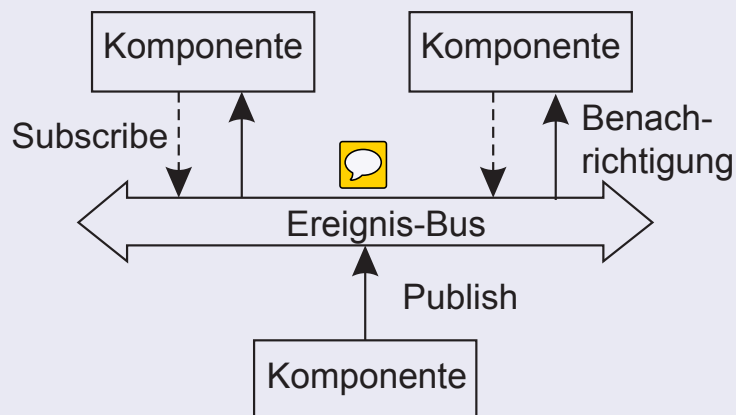
Koordination

Temporale und Referenzielle Kopplung

	Temporal gekoppelt	Temporal entkoppelt
Referenziell gekoppelt	Direkt	Mailbox
Referenziell entkoppelt	Ereignis-basiert	Geteilter Datenraum




Ereignis-basiert und geteilter Datenraum



Beispiel: Linda Tupelraum

Drei einfache Operationen

-  $\text{in}(t)$: entnehme Tupel, das Template t entspricht
- $\text{rd}(t)$: entnehme Tupel-Kopie, entsprechend Template t
- $\text{out}(t)$: füge Tupel t zum Tupelraum hinzu

Mehr Details

- Mehrfacher Aufruf von $\text{out}(t)$ speichert **zwei** Kopien von Tupel t
 \Rightarrow Ein Tupelraum ist als **Multimenge** realisiert.
- in und rd sind **blockierende** Operationen: Aufrufer blockiert bis passendes Tupel gefunden wurde, oder verfügbar wird.

Beispiel: Linda Tupelraum

Bob



```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("bob","distsys","I am studying chap 2"))
4 blog._out(("bob","distsys","The linda example's pretty simple"))
5 blog._out(("bob","gtcn","Cool book!"))
```

Alice

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 blog._out(("alice","gtcn","This graph theory stuff is not easy"))
4 blog._out(("alice","distsys","I like systems more than graphs"))
```

Chuck

```
1 blog = linda.universe._rd(("MicroBlog",linda.TupleSpace))[1]
2
3 t1 = blog._rd(("bob","distsys",str))
4 t2 = blog._rd(("alice","gtcn",str))
5 t3 = blog._rd(("bob","gtcn",str))
```

Altsysteme für Middleware

Problem

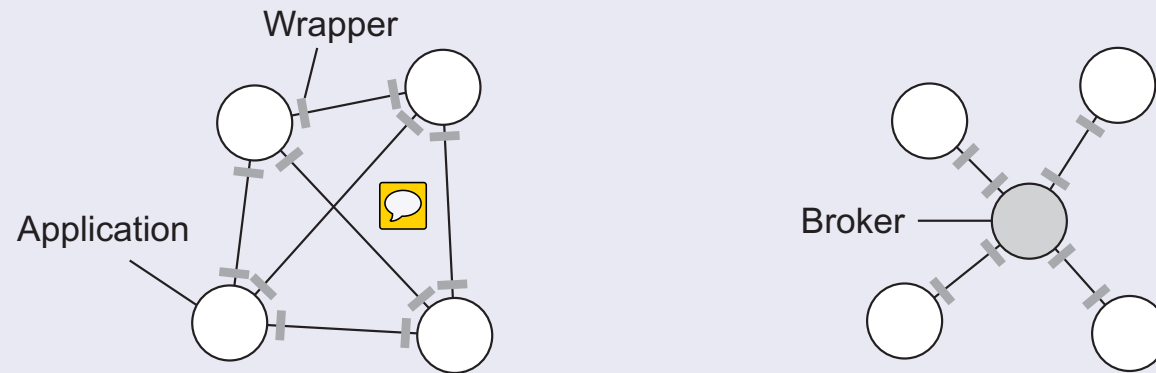
Interfaces einer alten (Legacy) Komponente meist nicht für alle Anwendungen geeignet.

Lösung

Wrapper oder **Adapter** bieten ein Interface passend zur Client Anwendung. Funktionen werden passend zur alten Komponente transformiert.

Wrapper organisieren

Zwei Lösungen: 1-zu-1 oder per Broker




Komplexität mit N Anwendungen

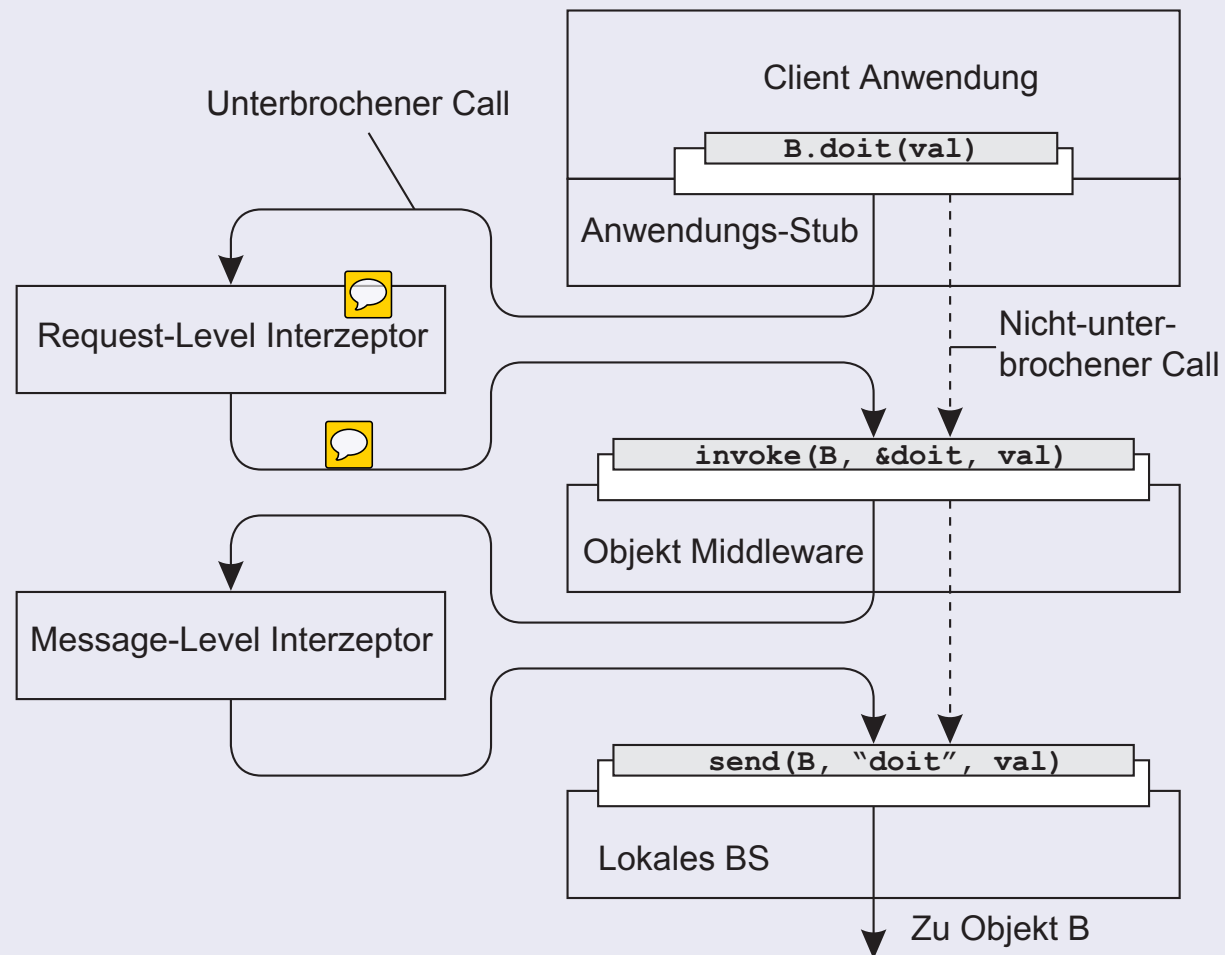
- **1-zu-1**: erfordert $N \times (N - 1) = \mathcal{O}(N^2)$ Wrapper
- **Broker**: erfordert $2N = \mathcal{O}(N)$ Wrapper

Entwicklung anpassbarer Middleware

Problem

Middleware enthält meistens Funktionen, die auf die **breite Masse** von Anwendungen Zielen \Rightarrow manchmal möchte man das Verhalten für spezifische Anwendungen anpassen. 

Unterbrechen (Interzeption) des Kontrollflusses



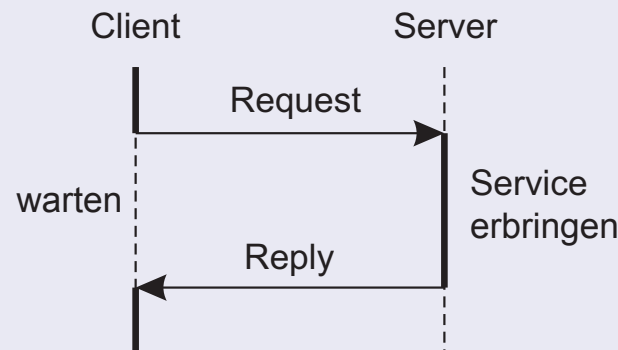


Zentralisierte System Architekturen

Grundlegendes Client–Server Modell

Charakteristiken:

- Es gibt Prozesse die Services anbieten (**Server**)
- Es gibt Prozesse die Services nutzen (**Clients**)
- Clients und Server können auf verschiedenen Maschinen laufen
- Clients folgen Request/Reply Modell bei Nutzung von Services



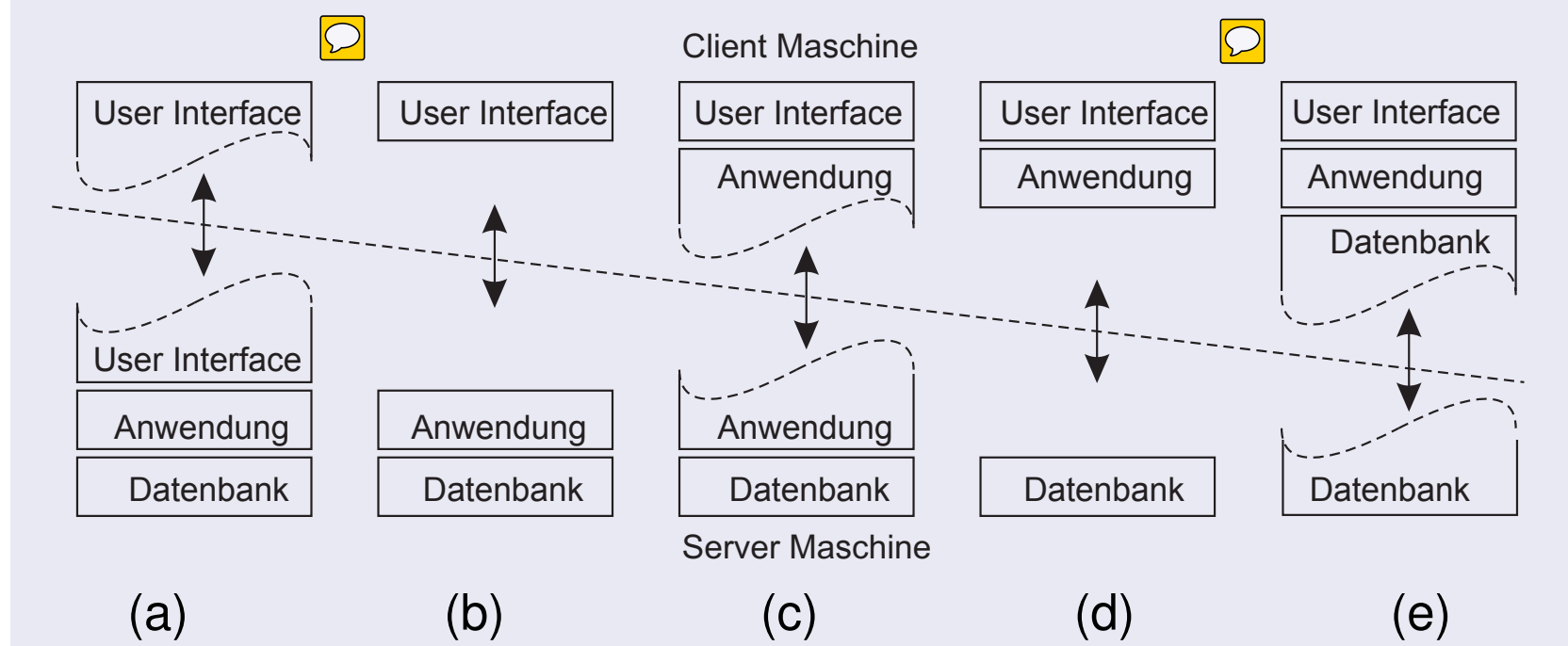


Mehrschichtige zentralisierte System Architekturen

Einige traditionelle Konfigurationen

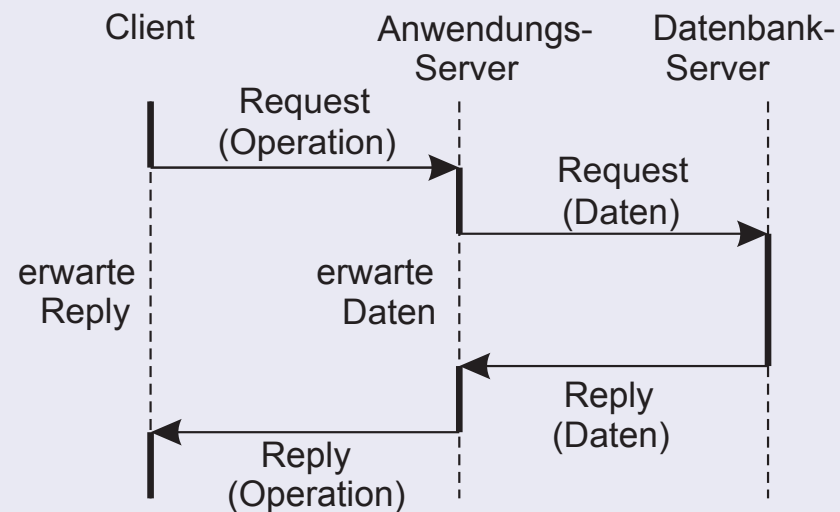
- **Einschichtig:** Terminal/Mainframe Konfiguration
- **Zweischichtig:** Client/Einzelserver Konfiguration
- **Dreischichtig:** Jede Schicht auf separater Maschine

Traditionelle zweischichtige Konfigurationen



Gleichzeitig Client und Server sein

Dreischichtige Architekturen



Übung 2: Mehrschichtige-Architekturen

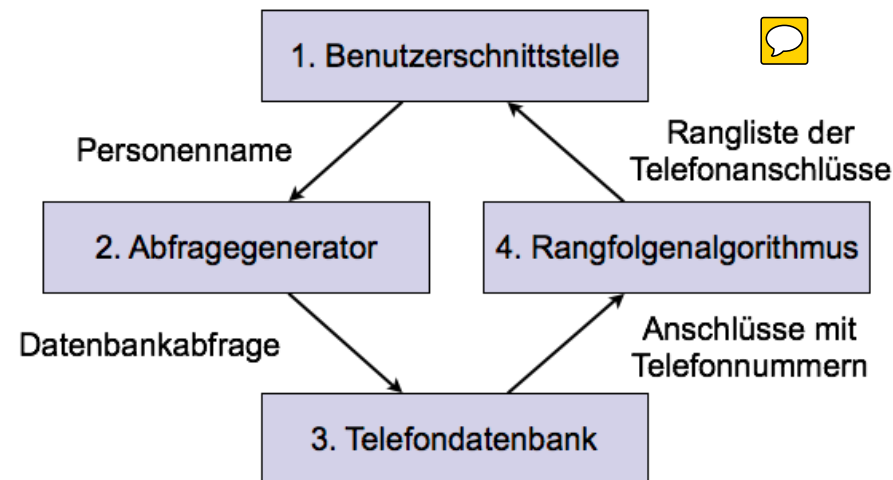
a) Schichtung von Anwendungen

Beschreiben Sie kurz die **drei Anwendungsebenen**, die der Schichtung hierarchischer Client-Server-Systeme zugrunde liegen.



b) Geschichteter Architekturstil

Ordnen Sie die Komponenten der gezeigten Telefonauskunft den zuvor genannten Anwendungsebenen zu.



Übung 2: Mehrschichtige-Architekturen

c) Verteilte Systemarchitektur

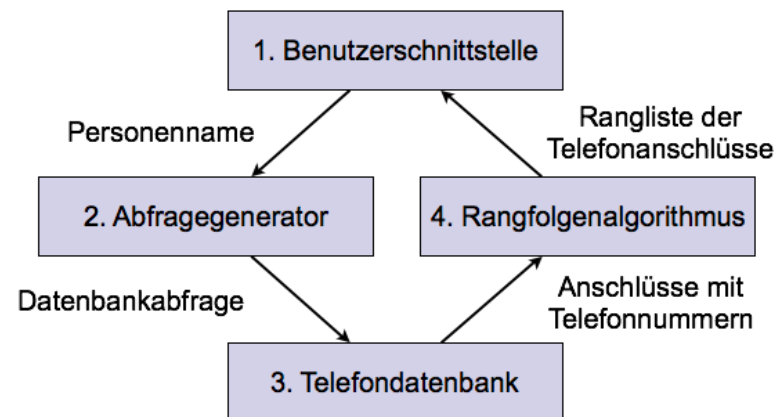
Die Telefonauskunft soll nun als verteiltes System realisiert werden. Dazu müssen die Komponenten der drei Anwendungsebenen auf verschiedene Rechner verteilt werden. Es sollen zwei Varianten getestet werden:

- V1 Zwei-Schicht-Architektur mit Fat Client
- V2 Drei-Schicht-Architektur mit Thin Client

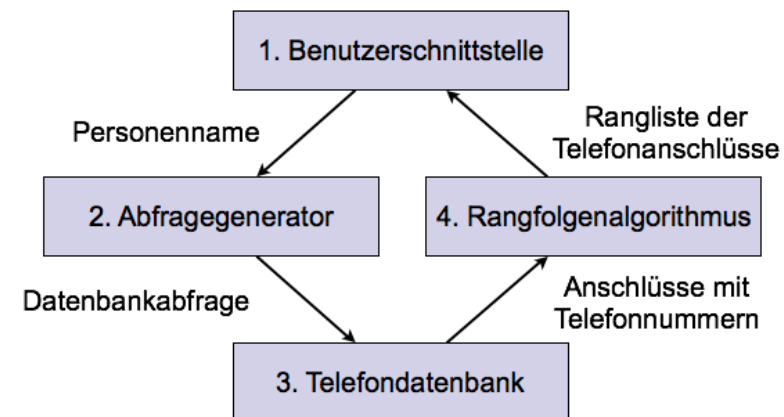
Die Komponenten können in Prozesse auf folgenden Geräten verteilt werden:

R1 Smartphone, R2 Workstation, R3 Webserver, R4 Datenbankserver

Beschreiben Sie für beide Varianten (V1, V2) der verteilten Systemarchitektur, auf welchen Geräten (R1-R4) welche Komponenten (1.-4.) jeweils laufen sollen.



V1



V2

Alternative Organisation

Vertikale Verteilung

Verteilte Anwendungen werden in drei logische Schichten unterteilt. Komponenten jeder Schicht laufen auf anderen Servern (Maschinen).

Horizontal Verteilung

Client oder Server wird physisch in logisch äquivalente Teile zerlegt, von denen jeder mit eigenem Anteil (Partition) des vollständigen Datensatzes arbeitet.

Peer-to-Peer Architekturen

Prozesse sind alle gleich: Funktionen, die ausgeführt werden müssen, werden von jedem Prozess repräsentiert \Rightarrow jeder Prozess agiert gleichzeitig als Client und Server (auch **Servant** genannt).

Strukturiertes Peer-to-Peer (P2P)

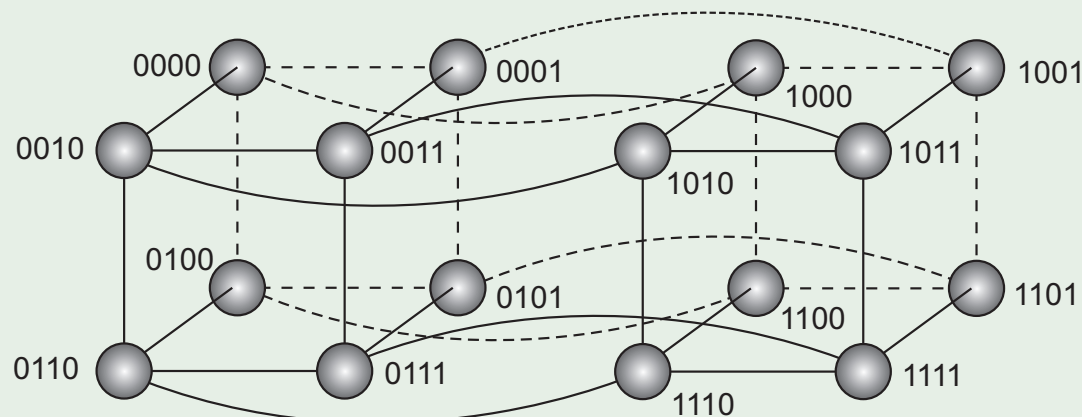
Im Kern

Nutze **Semantik-freien Index**: jedes Datenelement ist eindeutig assoziiert mit einem Schlüssel, der für Index genutzt wird. Praxis: nutze **Hash Funktion**:

$$\text{key}(\text{Datenelement}) = \text{hash}(\text{Wert des Datenelements})$$

P2P-System speichert nun (*Schlüssel*, *Wert*) Paare.

Einfaches Beispiel: Hypercube



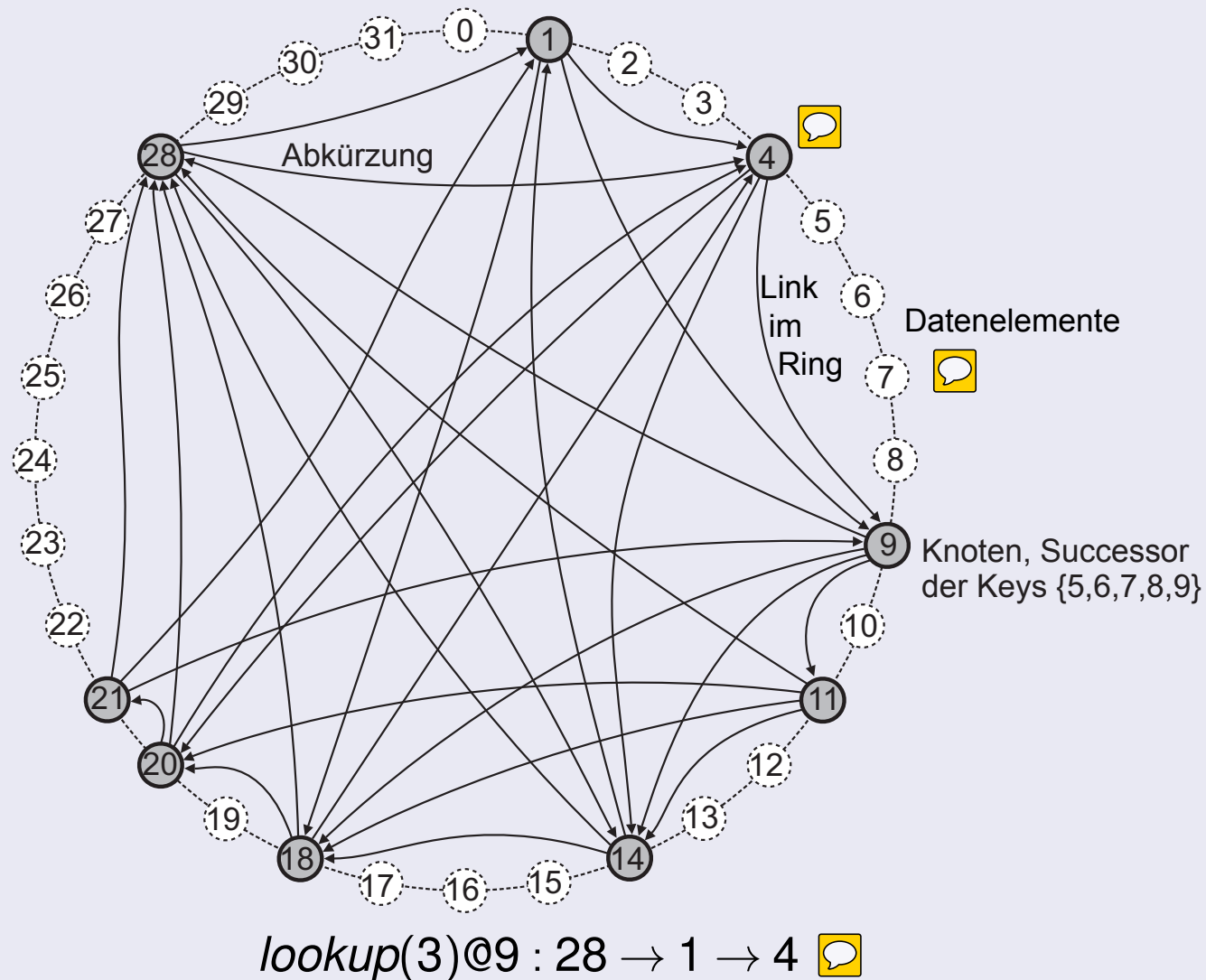
Nachschlagen von d mit **Schlüssel** $k \in \{0, 1, 2, \dots, 2^4 - 1\}$ durch **Routing** von Requests zum Knoten mit **Bezeichner** k .

Beispiel: Chord

Prinzip

- Knoten sind logisch in einem Ring organisiert. Jeder Knoten hat einen m -Bit **Bezeichner**.
- Jedes Datenelement wird zu einem m -Bit **Schlüssel** gehashed.
- Datenelement mit Schlüssel k wird im Knoten mit kleinstem Bezeichner $id \geq k$ gespeichert, genannt **Successor** von k .
- Der Ring wird mit verschiedenen **abkürzenden Verbindungen** zwischen anderen Knoten erweitert.

Beispiel: Chord Ring




Unstrukturiertes P2P

Im Kern

Jeder Knoten verwaltet eine Ad-hoc-Liste von Nachbarn. Das resultierende **Overlay** ähnelt einem **Zufallsgraphen**: Kante $\langle u, v \rangle$ existiert nur mit einer bestimmten Wahrscheinlichkeit $\mathbb{P}[\langle u, v \rangle]$.

Suchen

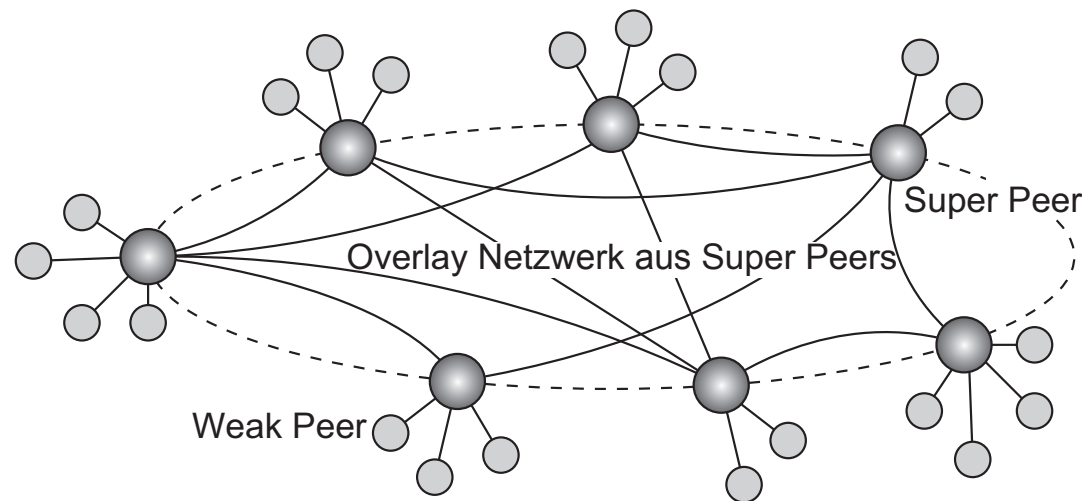
- **Flooding**: Ausgangsknoten u übergibt Anfrage für d allen Nachbarn. Anfrage wird ignoriert, wenn Empfangsknoten sie schon kennt. Sonst sucht v lokal nach d (rekursiv). Ggf. durch **Time-To-Live** begrenzt: maximale Anzahl von Sprüngen.
-  **Random Walk**: Ausgangsknoten u übergibt Anfrage für d an zufällig ausgewählten Nachbarn v . Wenn v nicht d hat, leitet es die Anfrage an zufällig gewählten Nachbarn weiter (und so fort).

Super-Peer Netzwerke

Im Kern

Es kann manchmal sinnvoll sein, die Symmetrie reiner P2P Netzwerke zu durchbrechen:

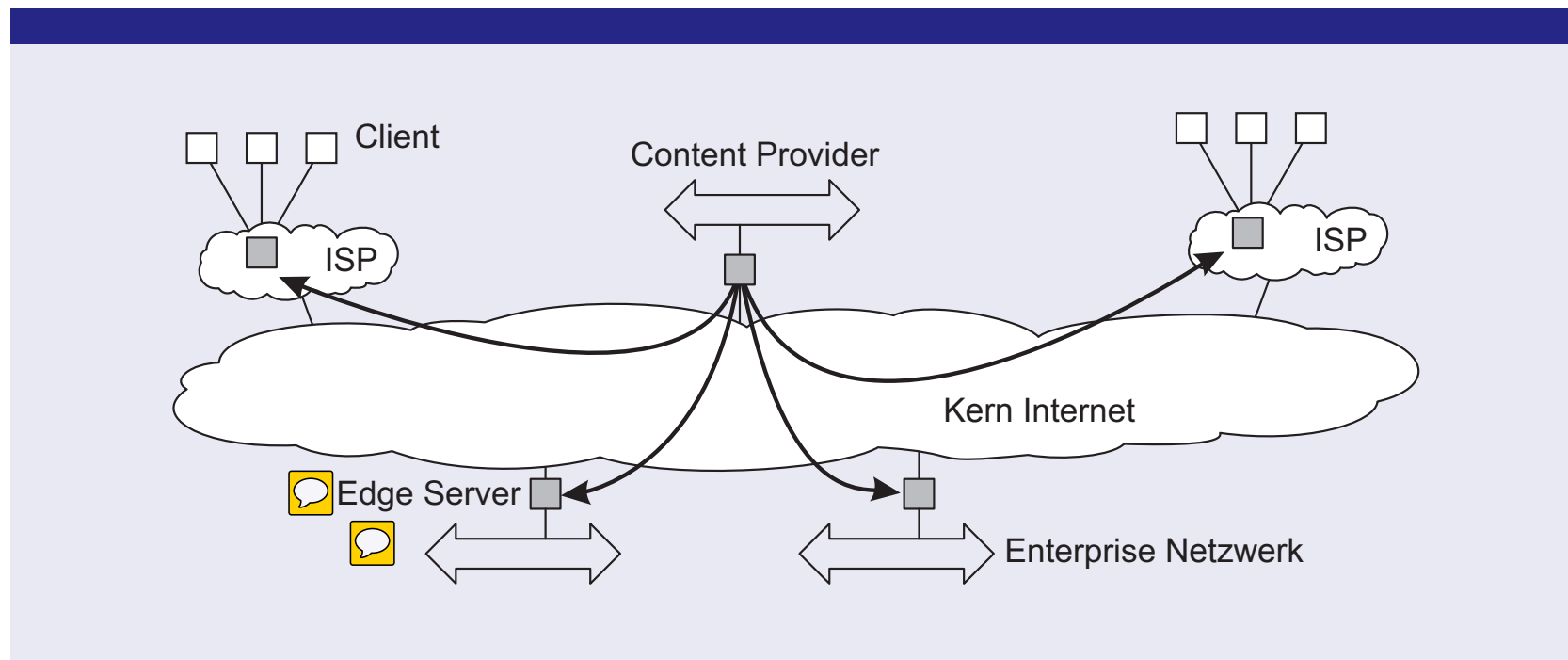
- Bei Suche in unstrukturierten P2P-Systemen bringen **Index Server** höhere Performanz.
- Entscheidung über Speicherort von Datenelementen können **Broker** oft effizienter treffen.



Edge-Server Architektur

Im Kern

Systeme, die im Internet eingesetzt werden, wo Server **am Rand** des Netzwerks platziert sind: an der Grenze zwischen Unternehmensnetzwerken und dem eigentlichen Internet.



Kollaboration bei BitTorrent

Prinzip: suche nach Datei F

- Lookup von Datei in Verzeichnis \Rightarrow führt zu **Torrent Datei**
- Torrent Datei enthält Referenz auf **Tracker**: Server mit akkurater Liste **aktiver** Knoten mit (Teilen von) F .
- P tritt **Schwarm** bei, kriegt Teil frei und handelt dann Kopie des Teils für den nächsten Teil mit Peer Q aus Schwarm.

