Verteilte Systeme
Betriebssysteme II

Kapitel 4.3: Client/Server Modell

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

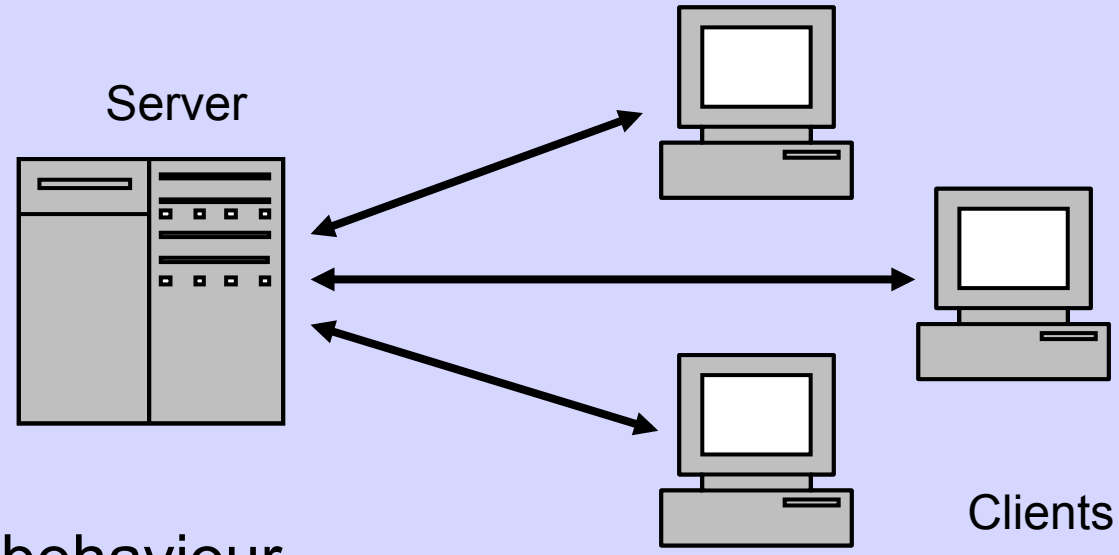
Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

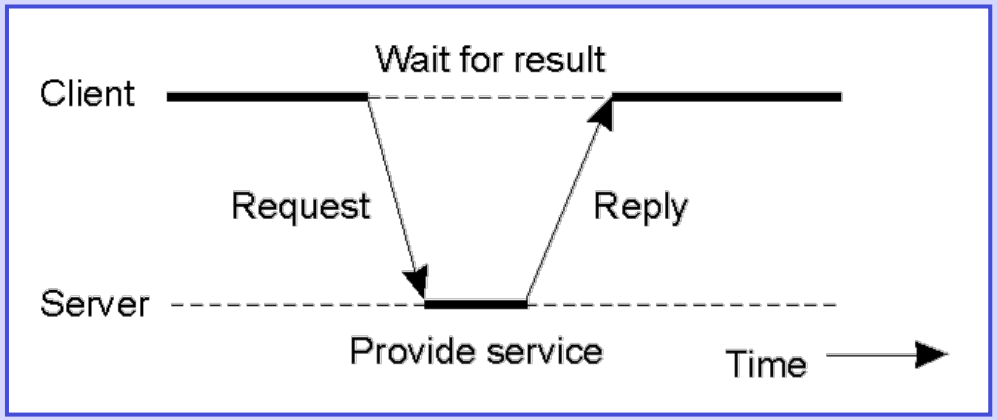
Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Client/Server-Modell



- request-reply behaviour



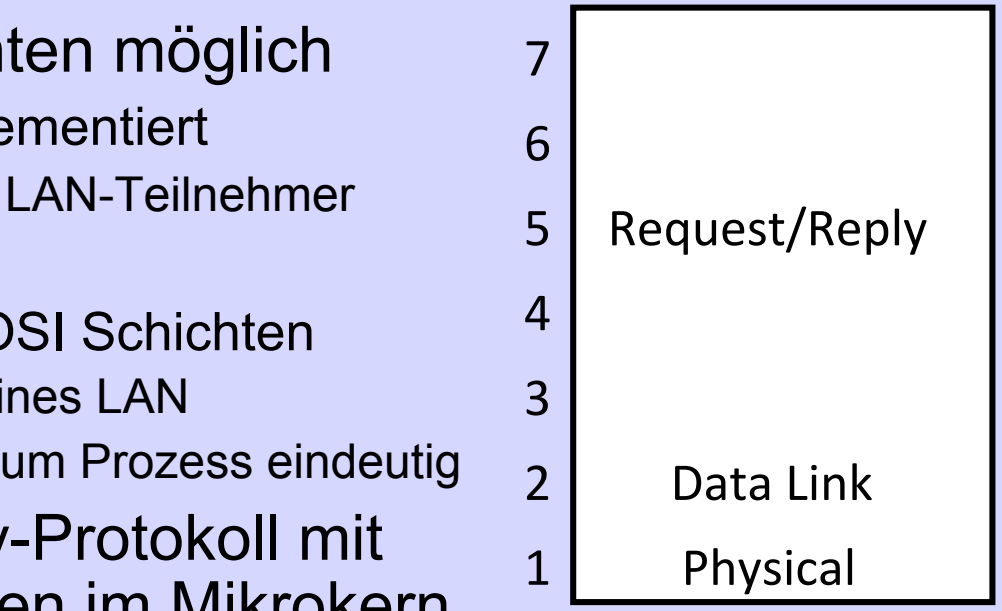
Quelle: „Distributed Systems“, Tanenbaum, van Steen, Abb.1-25

Client/Server-Modell

- Client/Server-Modell relevant für:
  - Dienste der höheren OSI Schichten (5-7)
  - verteilte Anwendungen
- Ablauf
  - Server (Dienstleistungsprogramm) stellt Funktion (Dienst) zur Verfügung
  - Client fordert bei Server Dienst an
  - *request/reply* Protokoll:
    - Weiterleiten von Anforderungen vom Client zum Server
    - Weiterleiten von Antworten vom Server zum Client
- ausreichend: verbindungsloser Datagramm Dienst

Client/Server-Modell

- verteiltes OS kann nach Client/Server-Prinzip strukturiert sein:
  - Menge von Servern läuft auf demselben Mikrokern und bietet Dienste an (Dateiserver, I/O Server . . .)
- Beschränkung auf 3 OSI-Schichten möglich
  - Schichten 1 und 2 durch LAN implementiert
    - Übermittlung von LAN-Paketen an LAN-Teilnehmer
  - Verbindungsaufbau i.A. nicht nötig
  - Bedingungen für Reduktion auf 3 OSI Schichten
    - OS läuft auf gleichen Maschinen eines LAN
    - Zuordnung von LAN Nachrichten zum Prozess eindeutig
- Implementierung Request/Reply-Protokoll mit zwei (blockierenden) OS-Befehlen im Mikrokern
  - `send(addr,&mess)`
  - `receive(addr,&buf)`

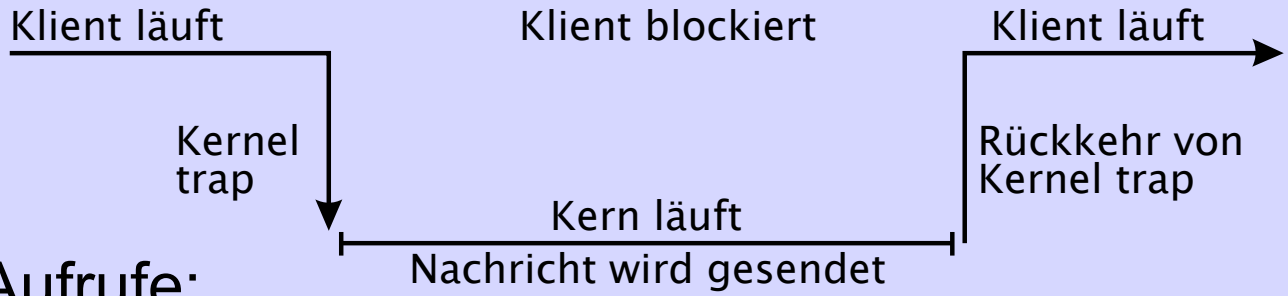


Client/Server-Modell – Adressierung

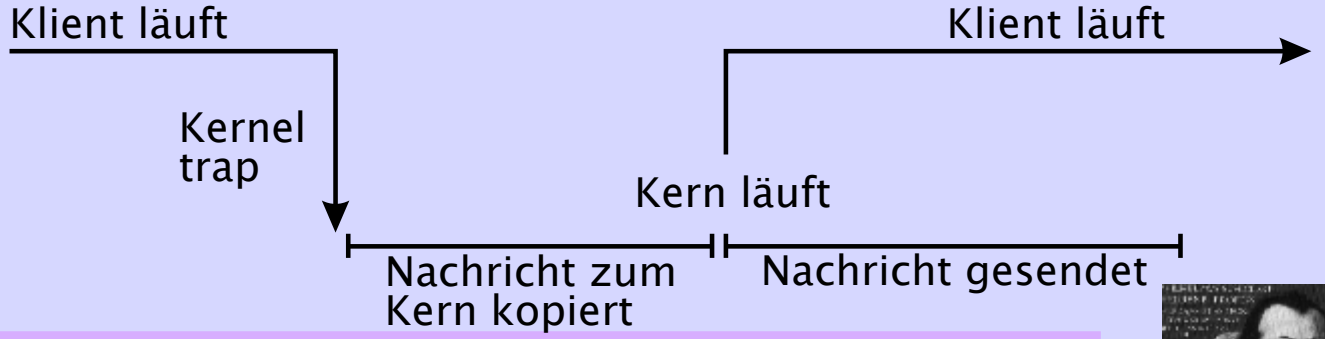
- Prozesse werden mit einem TSAP angesprochen
  - TSAP: `<machine,port>`
- BS-interne Abbildung (TSAP → Prozess-ID)
  - Vorteil: Kommunikation unabhängig von Prozess-ID
  - Problem: Nachrichten werden an fest codierte TSAPs geschickt
    - Server kann nicht auf andere Maschine verlagert werden
- Alternativ: anwendungsspezifisch eindeutiger Server-Name
  - z.B. vergeben durch zentrale Instanz (Skalierproblem!) oder als Zufallszahlen aus grossem (z.B. 64 bit) Adressraum
  - Abbildung (Namen → TSAP) durch *broadcast*
- Alternativ: *name server* an bekanntem TSAP
  - Skalierungsproblem!

Client/Server-Modell – (nicht) blockierende Aufrufe

- Standardversionen von send und receive sind *blockierend* (oder *synchron*):
  - Aufrufer blockiert, bis Nachricht verschickt (evtl. angekommen) bzw. empfangen



- nichtblockierende Aufrufe:
  - Aktion wird initiiert
  - Aufrufer erhält sofort Kontrolle zurück



Client/Server-Modell – (nicht) blockierende Aufrufe

- Vorteil nichtblockierende Lösung:
  - erhöhte Leistung durch Ausnutzen von Parallelität
  - Laufzeit von send/receive berechenbar
- Nachteil nichtblockierende Lösung:
  - Zusatzaufwand des Kopierens
    - Alternativ: Kern benachrichtigt Prozess nach dem Senden, dass Puffer frei ist
    - Nachteil: Kompliziertere Programmlogik
- Asynchrones Arbeiten auf Multiprozessoren mit Multithreading
  - einfache Lösung für send: blockierendes send als asynchroner Thread
  - Wichtiger: asynchrones receive
    - Wartezeit auf Nachricht nicht absehbar
    - Server will mehrere Ports bedienen → Reaktion auf nächste an irgendeinem Port ankommende Nachricht

Client/Server-Modell – (nicht) blockierende Aufrufe

- UNIX
  - send/receive blockierend
  - select: Auswahl eines Ports, an dem eine Nachricht anliegt
- Bei Multithreading:
  - für jeden Port separater receive-Thread
    - übernimmt auch Bearbeitung der Nachricht
    - oder meldet master-Prozess Nachrichteneingang
  - wartender receive-Thread hat keinen Kontext außer
    - Puffer
    - Zeiger auf die Empfangsroutine (message handler)
  - Alternativ: BS kann bei Eintreffen der Nachricht Thread spontan erzeugen (*pop-up thread*) und handler-Routine auf ihm starten
    - `receive(port,buffer,handler)`

Client/Server-Modell – (un)gepufferte Aufrufe

- *synchron* und *ungepuffert* → Rendezvous zwischen Sender und Empfänger
- *asynchron* und *gepuffert* → Kommunikation zeitlich entkoppelt (mailbox)
- asynchroner Aufruf
  - nur je ein Teil-Thread der Partner, der asynchron vom (Haupt-)Thread läuft, treffen sich genau
- gepufferter Aufruf
  - eingetroffene Nachricht wird vom BS zwischengespeichert, bis sie vom Empfänger abgerufen werden kann
- ungepuffertes `receive(addr,&buf)`
  - Empfänger wartet auf Nachricht von Adresse addr
  - Kommt Nachricht an, so wird sie in buf kopiert und receive() deblockiert
  - Kein receive()-Aufruf → Nachricht geht verloren
- asynchrones `receive()`
  - receive()-Thread verwaltet Nachricht, bis Haupt-Thread sie braucht
  - Empfänger richtet zu Programmbeginn Briefkasten (*mailbox*) ein
  - BS speichert eingehende Nachrichten (bis zu max. Anzahl)

Grundlegende Message Passing Primitive: Send und Receive

- **`send(void* sendbuf, int nelems, int dest)`**
  - **`sendbuf`**: Zeiger auf Puffer mit den zu sendenden Daten
  - **`nelem`**: Anzahl der zu sendenden Datenworte
  - **`dest`**: ID des Empfänger-Prozess
- **`receive(void* recvbuf, int nelems, int source)`**
  - **`recvbuf`**: Zeiger auf Puffer für die zu empfangenden Daten
  - **`nelem`**: Anzahl der zu empfangenden Datenworte
  - **`source`**: ID des Sender-Prozess

Blockierende vs. nicht-blockierende Message Passing Operationen

Prozess P0

```
a = 100;
send(&a,1,1);
a = 0;
```

Prozess P1

```
receive(&a,1,0);
printf("%d\n",a);
```

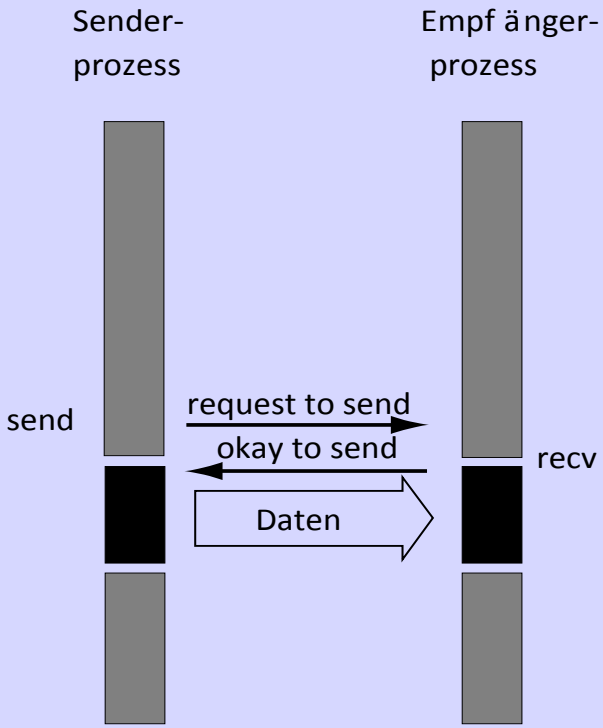
- Oft wird (mittels spezieller Hardware) die Nachrichtenübertragung parallel zum Programmablauf durchgeführt.
  - Übertragung von 0 statt 100 möglich!
- **Blockierende Message Passing Operationen**
  - Send Aufruf kehrt erst zurück, wenn der Sendepuffer modifiziert werden kann, ohne die Semantik zu verändern.
- **Nicht-blockierende Message Passing Operationen**
  - Send Aufruf kehrt sofort zurück; Korrektheit muss vom Programmierer sichergestellt werden.

Blockierende nicht-gepufferte Send/Receive Operationen

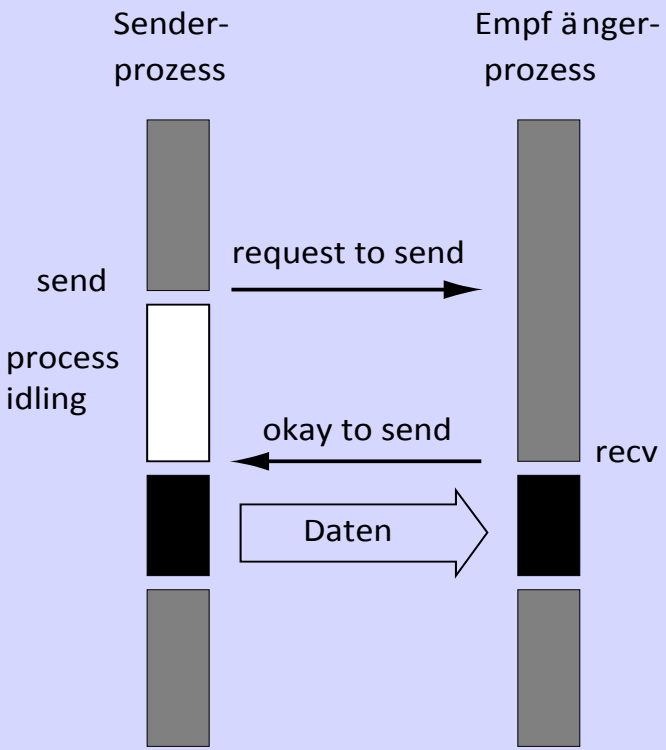
- Send-Aufruf kehrt erst zurück, wenn
  - der entsprechende Receive-Aufruf stattgefunden hat und
  - die Nachricht vollständig übertragen wurde.
- Implementierung mittels **Handshake-Protokoll**.
- Nachteile:
  - Falls keine enge Synchronisation möglich, wird der Sender- oder Empfängerprozess blockiert (→ **Process Idling**).
  - Deadlocks möglich:

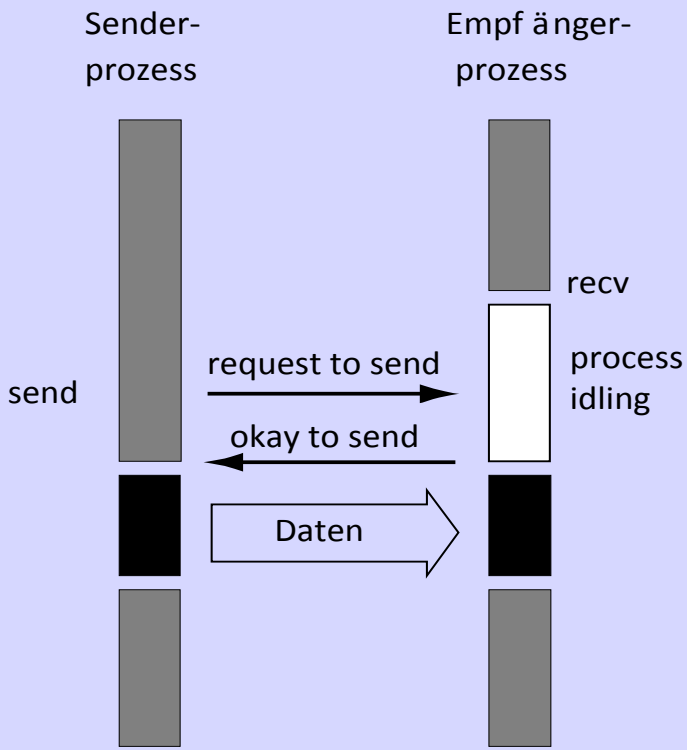
Prozess P0	**Prozess P1**
`send(&a,1,1);`	`send(&b,1,0);`
`receive(&b,1,1);`	`receive(&a,1,0);`

Blockierende nicht-gepufferte Send/Receive Operationen



Blockierende nicht-gepufferte Send/Receive Operationen



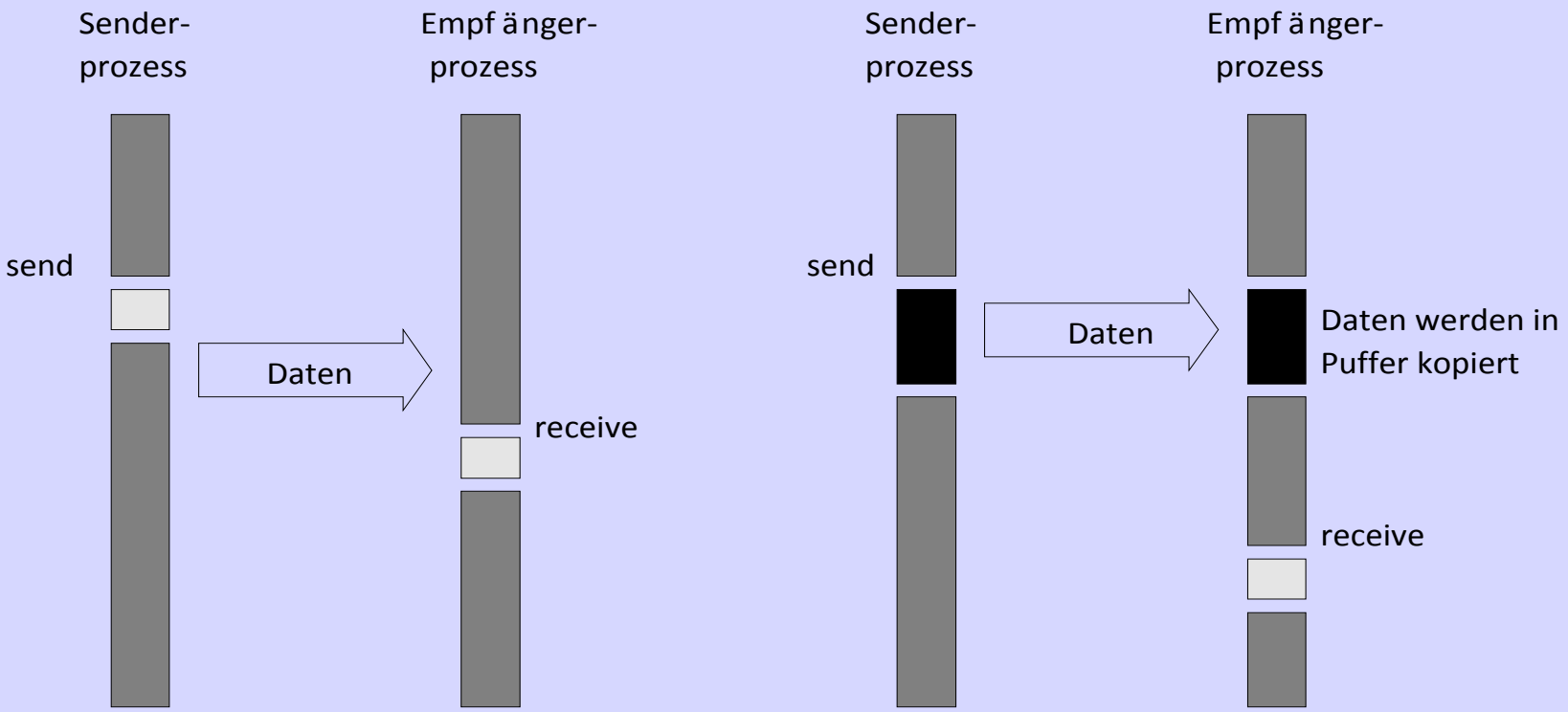


Blockierende gepufferte Send/Receive Operationen

- Sender und/oder Empfängerprozess verwenden interne Pufferspeicher für die Kommunikation.
  - Puffervariablen im Programm werden vom eigentlichen Nachrichtenaustausch entkoppelt.
- Nachteile:
  - Overhead für Puffermanagement (Kopieren der Daten, ...).
  - Bei Pufferüberlauf muss Senderprozess blockiert werden.
  - Deadlocks möglich (receive Operation kehrt erst zurück, wenn Daten im lokalen Puffer verfügbar sind):

Prozess P0	**Prozess P1**
`receive(&a,1,1);`	`receive(&b,1,0);`
`send(&b,1,1);`	`send(&a,1,0);`

Blockierende gepufferte Send/Receive Operationen



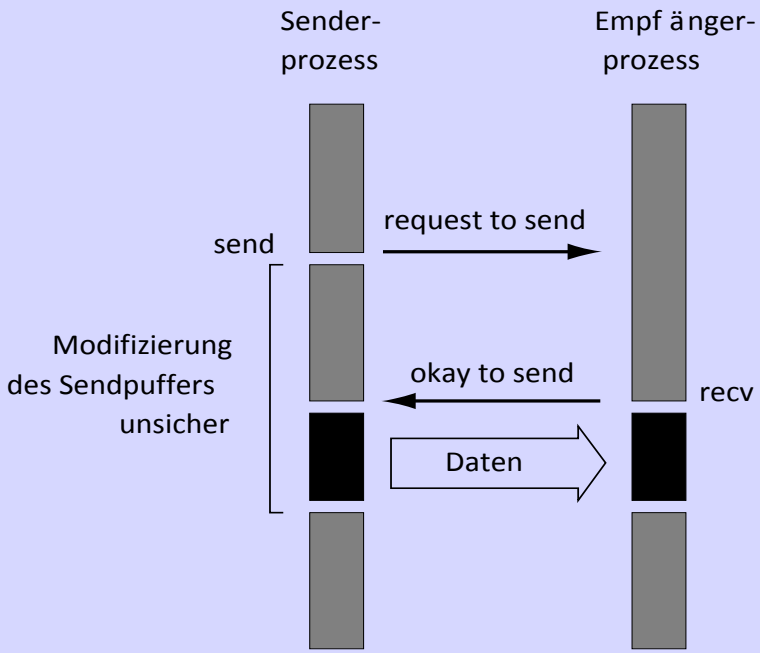
Mit Kommunikationshardware
Puffer bei Sender- und
Empfängerprozess

Ohne Kommunikationshardware
Puffer bei Empfängerprozess

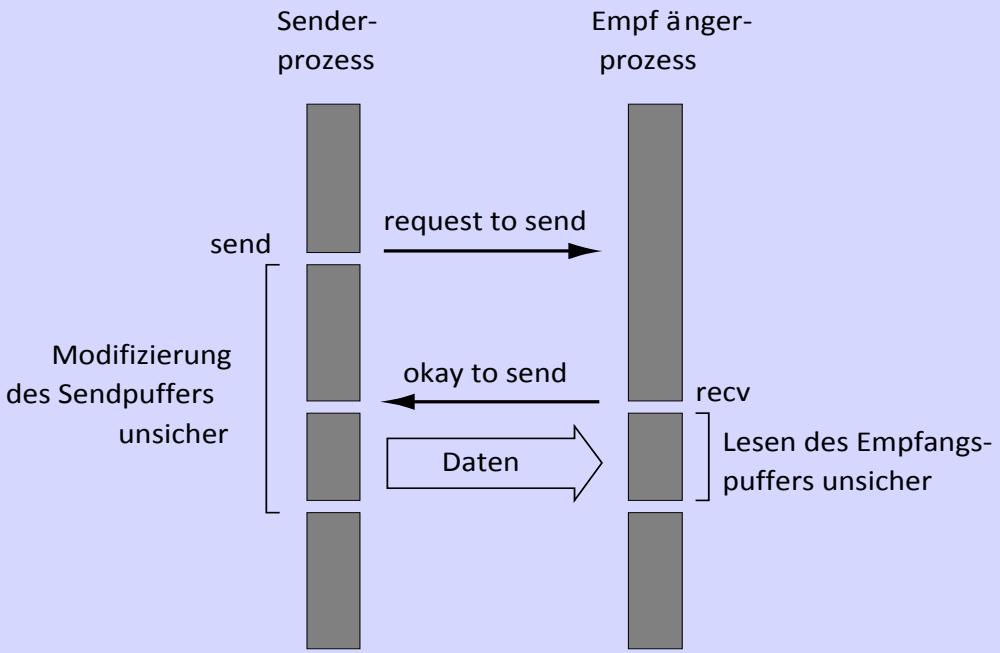
Nicht-blockierende Send/Receive Operationen

- Nicht-blockierende Send/Receive Aufrufe kehren zurück, bevor Puffervariablen sicher geändert werden können.
- Kein Overhead in Form von Process Idling oder Puffermanagement wie bei den blockierenden Operationen
- Programmierer muss sicherstellen, dass Puffervariablen nicht vor Beendigung der Kommunikationsoperation verändert werden.
- **Check-Status** Primitiv gibt Auskunft, ob Puffervariablen sicher überschrieben werden können.

Nicht-blockierende Send/Receive Operationen



Ohne Kommunikationshardware



Mit Kommunikationshardware

Vergleich

	blockierend	**Nicht-blockierend**
gepuffert	Send-Aufruf kehrt zurück, nachdem die Daten in den Kommunikationspuffer kopiert wurden.	Send(Receive)-Aufruf kehrt nach dem Start des Kopierens (DMA) sofort zurück. Sichere Modifikation der Daten nicht sofort möglich.
nicht-gepuffert	Send-Aufruf kehrt zurück, wenn ein entsprechender Receive-Aufruf ausgeführt wurde.	
	Korrektheit wird implizit sichergestellt	Programmierer muss Korrekt-heit explizit sicherstellen

Client/Server-Modell – (un)zuverlässige Aufrufe

- Vermeiden von Nachrichtenverlust durch Quittierung des BS
  - bei Ausbleiben der Quittung neues Senden der Nachricht
- Beispiel:
  1. Request (client→server)
  2. Ack (server→client)
  3. Reply (server→client)
  4. Ack (client →server)
- send() bleibt blockiert, bis Quittung erhalten
  → Overhead
- Verzicht auf erstes Ack, falls Reply früh genug gesendet wird
- Verzicht auf zweites Ack, falls Anfrage einfach wiederholt werden kann

Implementierung des Client-Server-Modells

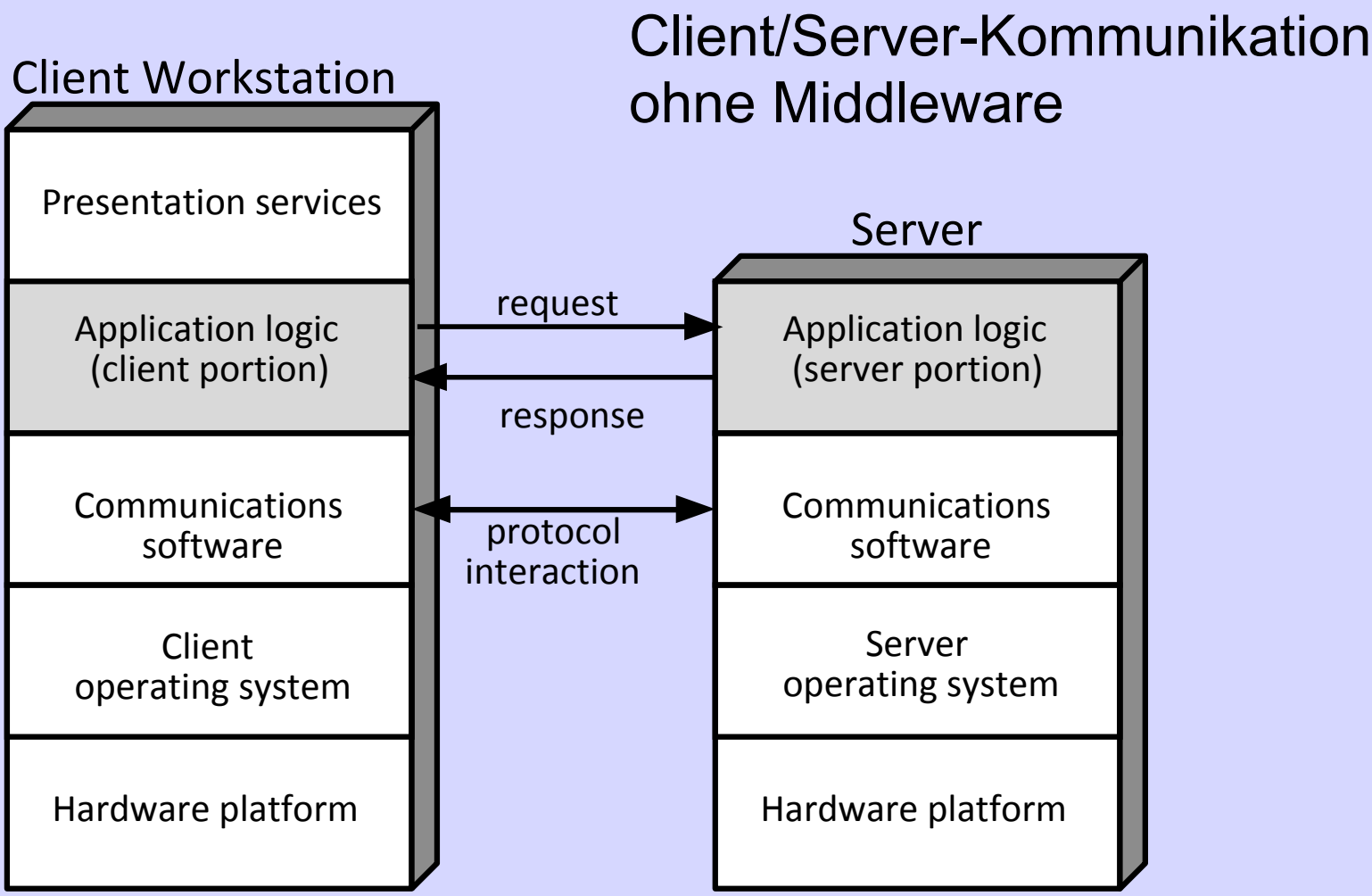
- Nachrichten müssen u.U. fragmentiert werden
  - Grund: Paketgröße der zugrundeliegenden Netzwerkschicht
- Client/Server-Protokoll kann eigene Pakete definieren

Implementierung des Client-Server-Modells

- Typische Nachrichten im Client/Server Modell:

Code	Paket-Typ	Richtung	Funktion
REQ	Request	Client→ Server	Enthält Anforderung (Auftrag)
REP	Reply	Server→Client	Enthält Bearbeitung des Auftrags
ACK	Acknowledge	Server→Client	Quittiere empfangenes Paket
AYA	Are you alive?	Client→ Server	Finde heraus, ob der Server funktioniert
IAA	I am alive	Server→Client	Der Server funktioniert
TA	Try again	Server→Client	Der Server ist z.Zt. überlastet
AU	Address unknown	Server→Client	Kein Prozess benutzt diese Adresse

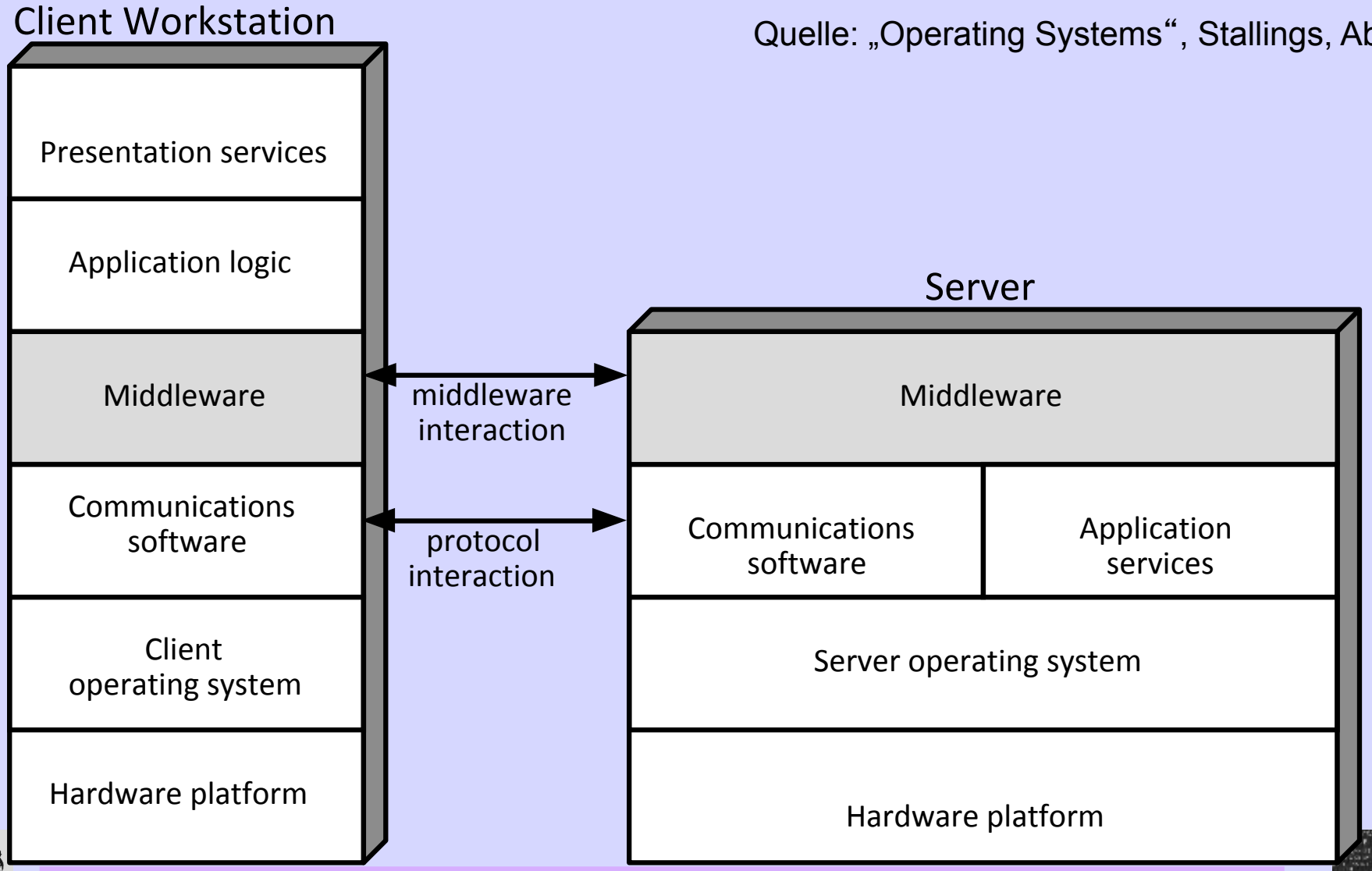
Die Rolle von Middleware

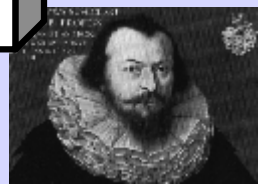


Quelle: „Operating Systems“, Stallings, Abb.13-7

Die Rolle von Middleware

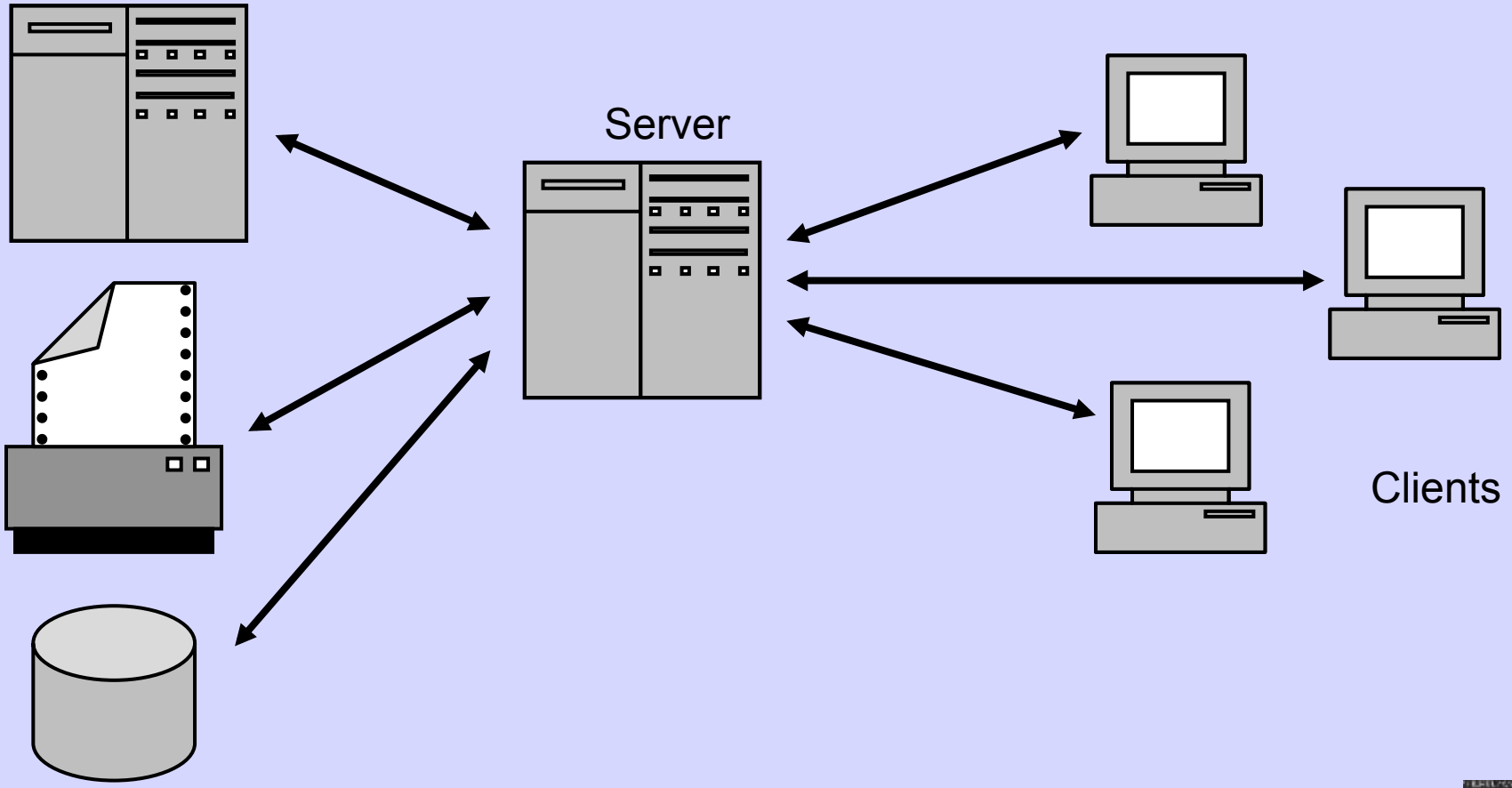
Quelle: „Operating Systems“, Stallings, Abb.13-12





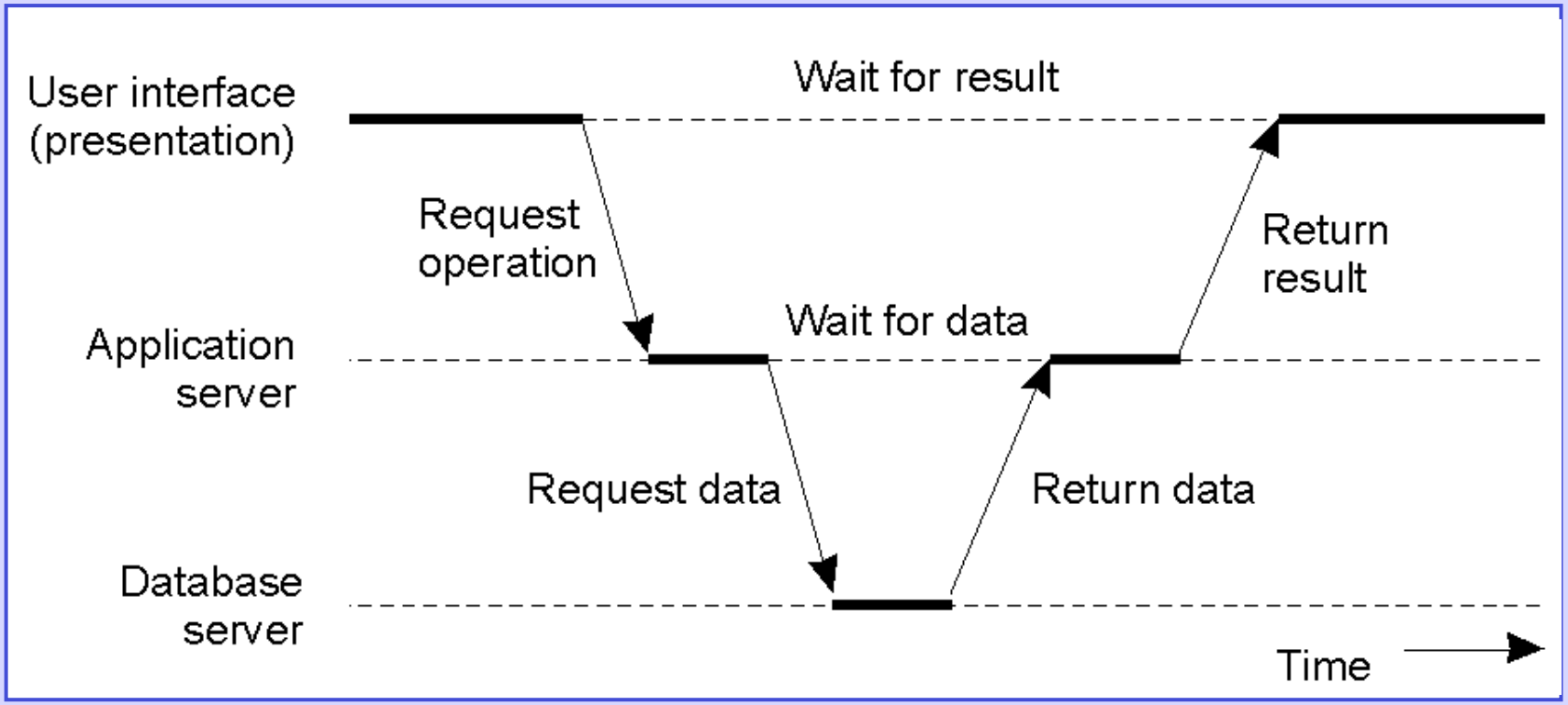
Three-Tier-Modell

- Server selbst ist auch Client



Three-Tier-Modell

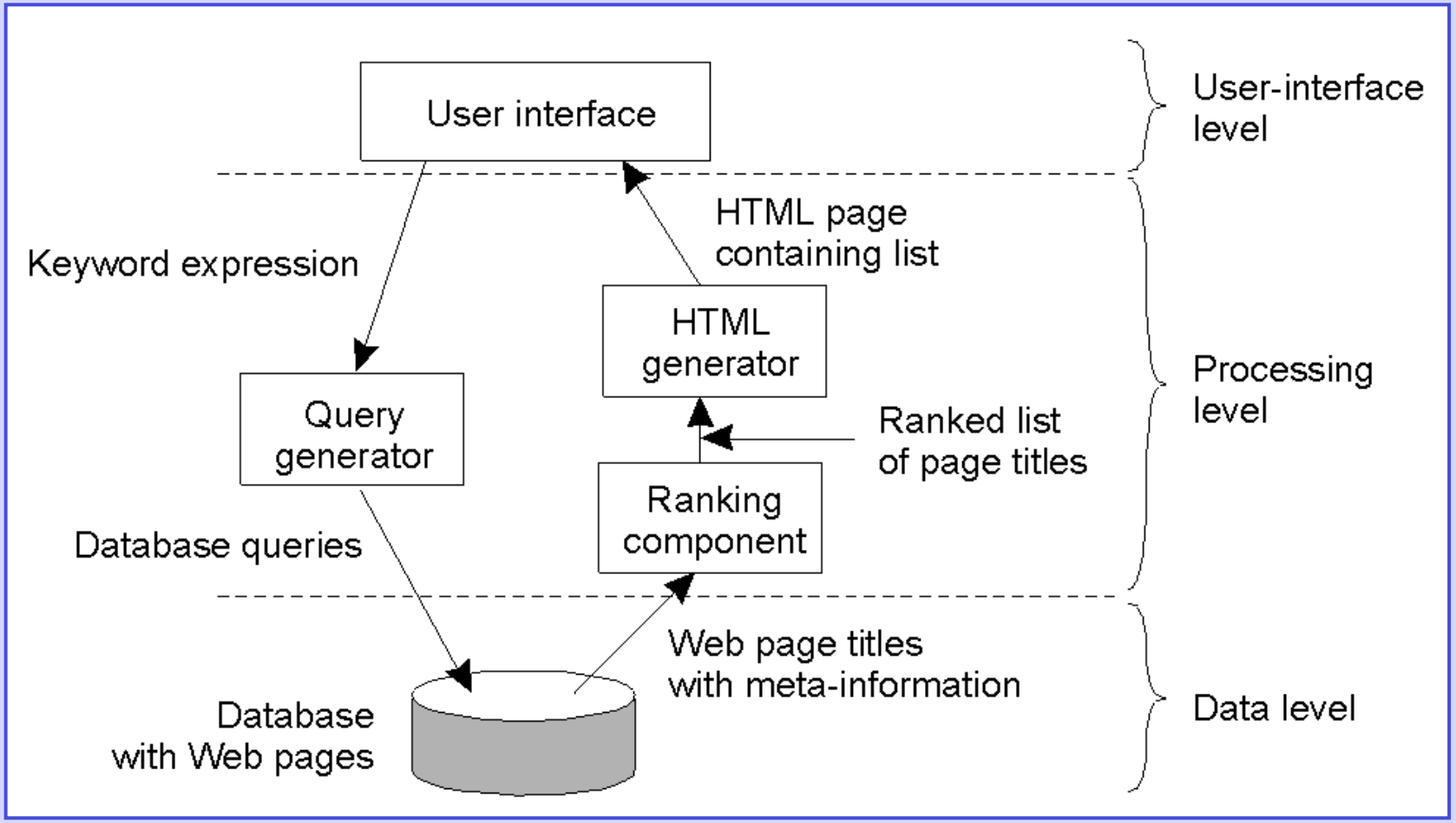
- erweitertes request-reply Verhalten



Quelle: „Distributed Systems“, Tanenbaum, van Steen, Abb.1-30

3+ -Schichten Client/Server-Modell

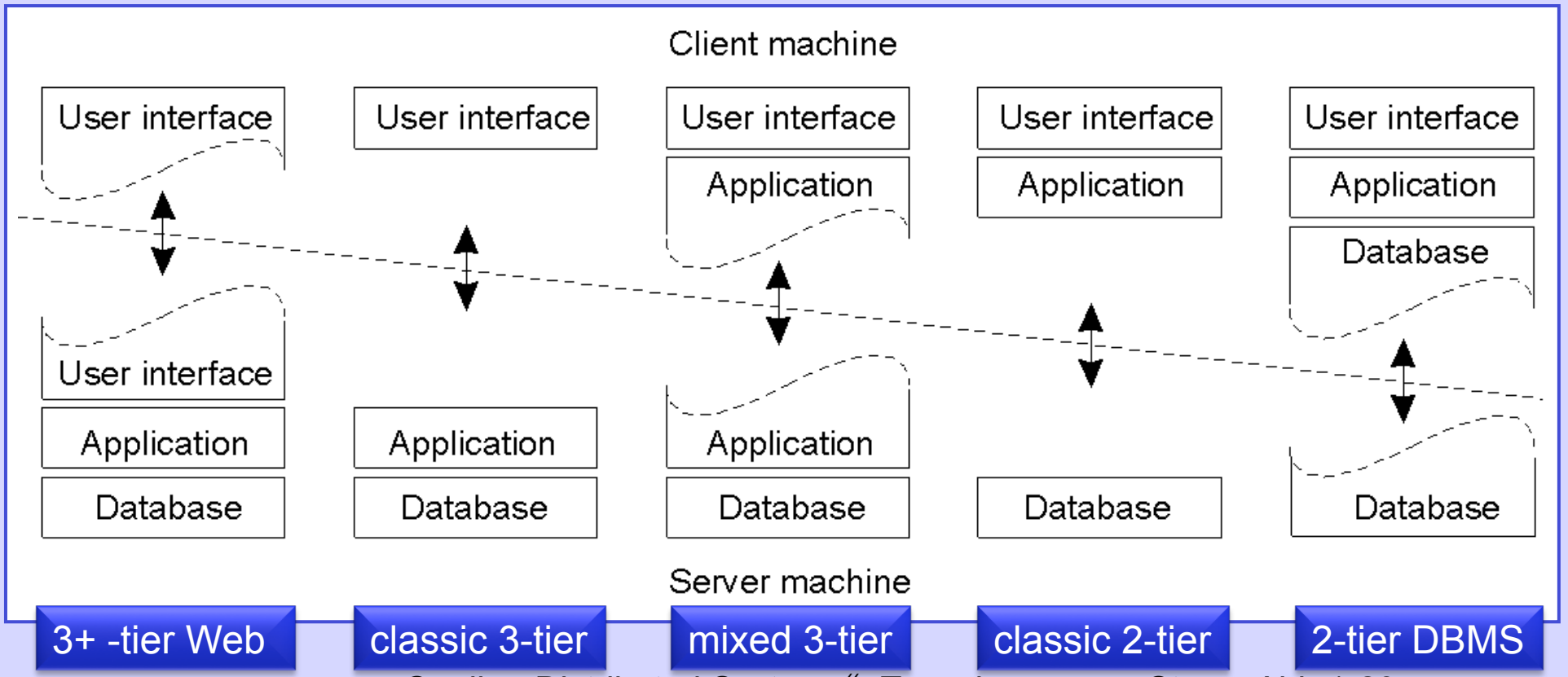
- Zwischenschichten in der Mitte, z.B. Web-Schicht



Quelle: „Distributed Systems“, Tanenbaum, van Steen, Abb.1-28

Client/Server-Modell: Aufgabenverteilung

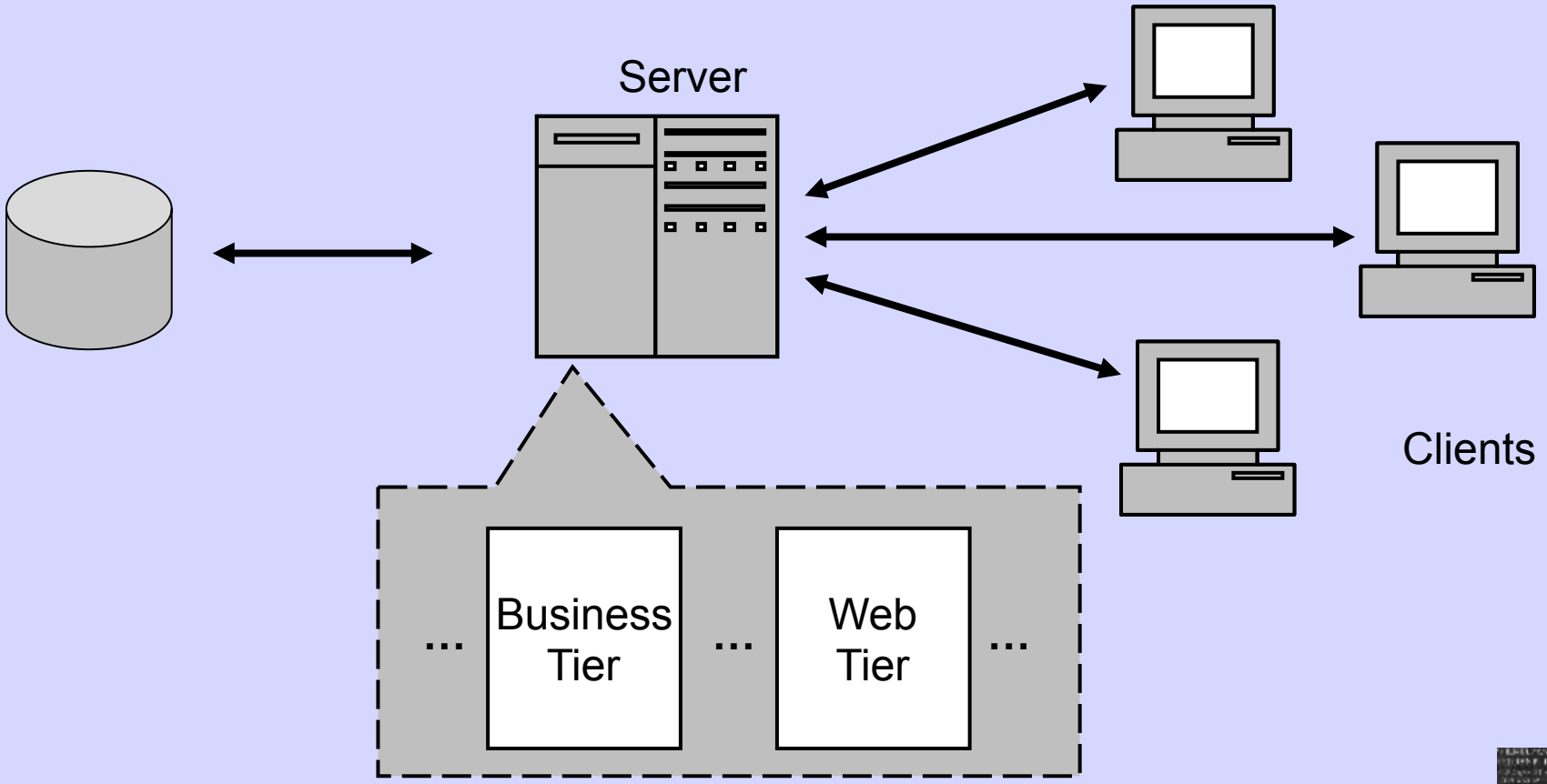
- Wie wird die Anwendung zwischen Server und Client verteilt?
  - Thin Client → Fat Client



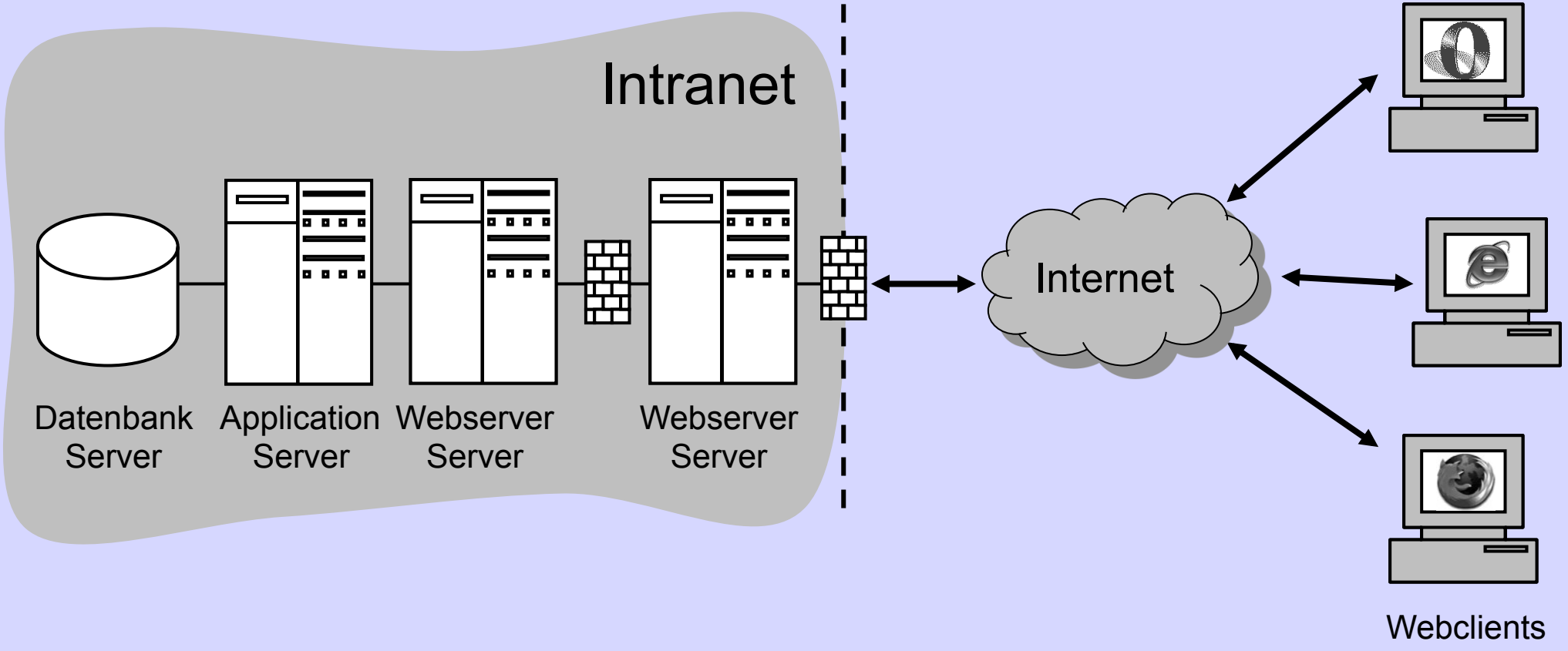
Quelle: „Distributed Systems“, Tanenbaum, van Steen, Abb.1-29

Multitier-Modell

- Einteilung in Komponenten gemäss der Anwendungslogik.

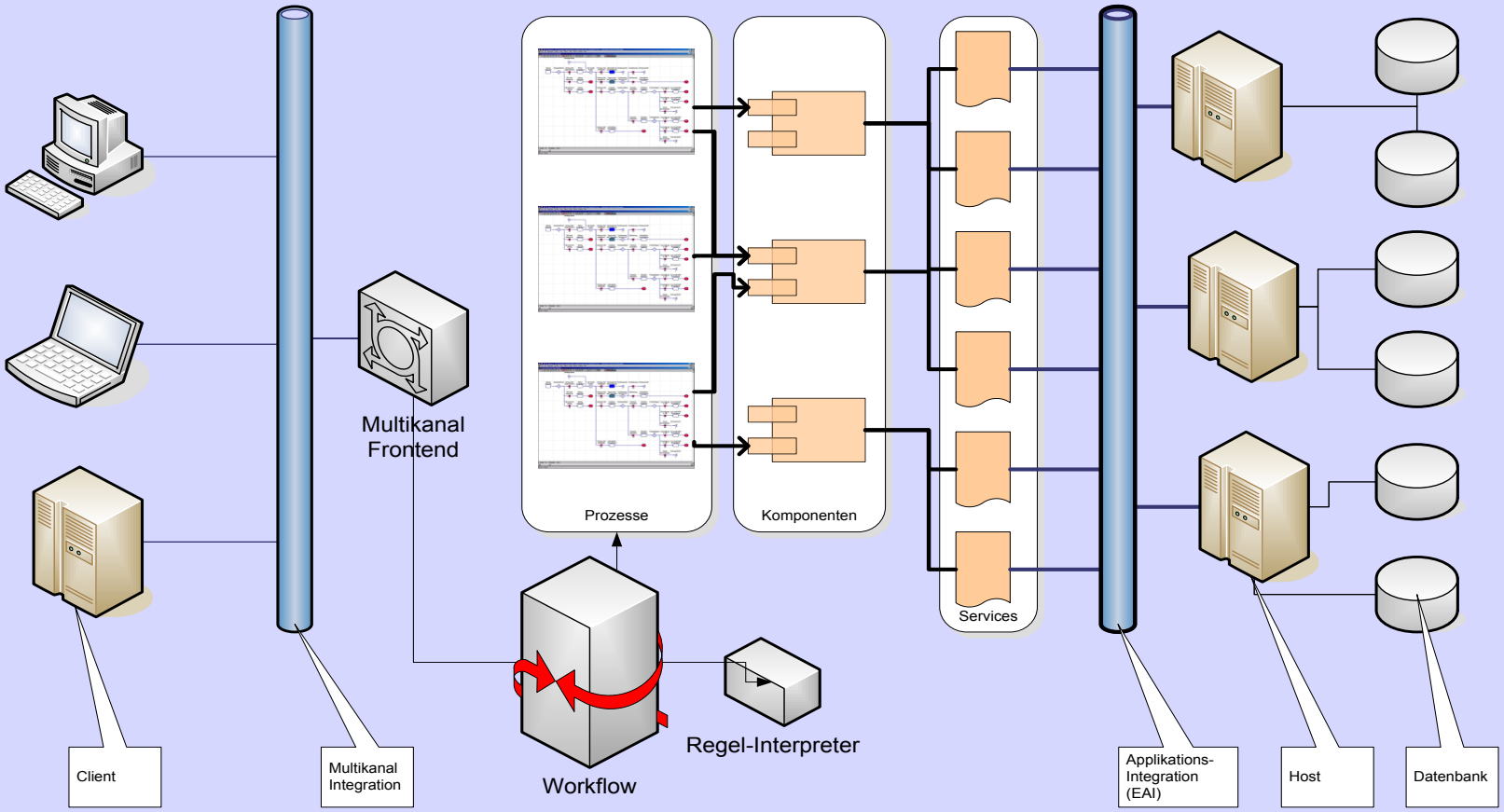


Beispiel: 4-Tier-Internetanwendung



nach „Verteilte Systeme und Anwendungen“, Hammerschall, Abb.13-7

Die IBM Referenzarchitektur



Die IBM Referenzarchitektur

- Client Tier: Vielzahl unterschiedlicher Clients
- Multichannel Integration
  - verschiedene Zugänge: WAP-Handy, Geldautomat, Palmtop, Laptop, 3270
  - Umsetzung der Client-Protokolle und Erzeugung der Client Views
- Application Tier
  - Workflow System implementiert Geschäftsprozesse
    - basierend auf Geschäfts-Komponenten
    - WAS Process Server implementiert WS-BPEL, interpretiert Regeln
  - Geschäftskomponenten integrieren einzelne Dienste zu Business Funktionen (z.B. Authentifizierungs-Dienst benötigt für Auszahlung)
  - Einzeldienste konventionell oder als Web-Services
- EAI (Enterprise Application Integration) Schicht
  - bindet konventionelle Systeme ein
  - verbindet (klassische) Anwendungen (high volume application mediation)
- Host / Datenbanken
  - wie bisher