

Verteilte Systeme 2

Prinzipien und Paradigmen

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

Kapitel 04: Kommunikation

Version: 30. Oktober 2018

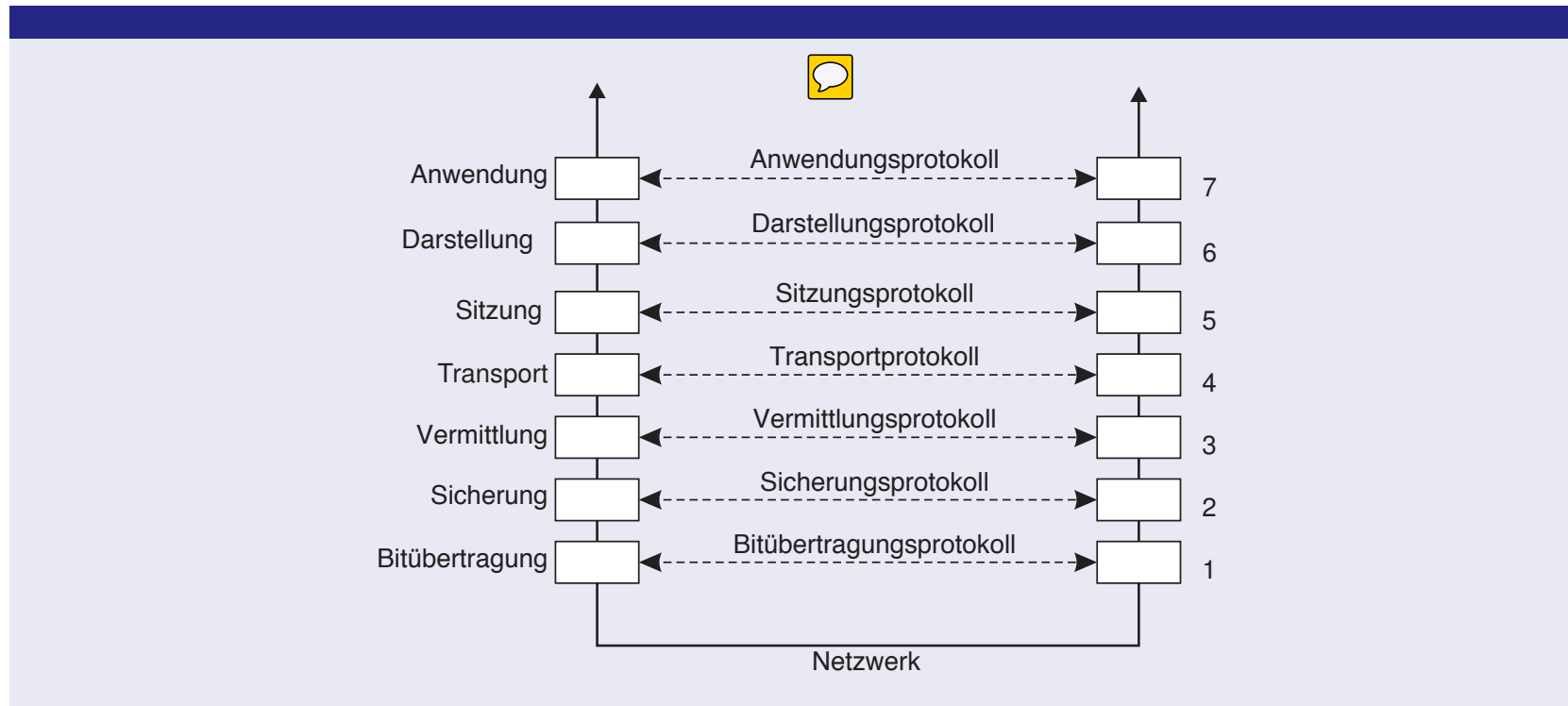


Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Inhalt

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

Basismodell der Nachrichtenkommunikation



Nachteile

- Fokus auf reinem Nachrichtenaustausch
- Z.T. unnötige oder unerwünschte Funktionalität
- Verletzt Zugriffstransparenz

Hardwarenahe Schichten

Recap

- **Bitübertragungsschicht**: enthält die Spezifikation und Implementierung von Bits und ihrer Übertragung zwischen Sender und Empfänger
- **Sicherungsschicht**: schreibt die Übertragung einer Serie von Bits als **Frame** vor, um Fehler- und Flusskontrolle zu ermöglichen
- **Vermittlungsschicht**: beschreibt, wie Pakete in einem Netzwerk aus Rechnern (und zwischen Netzwerken) geleitet werden (**routing**)

Observation

Für viele verteilte Systeme ist die Vermittlungsschicht am Hardware-nächsten.

Transportschicht

Wichtig

Die Transportschicht stellt die Kommunikationsmechanismen für die meisten verteilten Systeme bereit.

Standard Internet Protokolle

- **TCP**: verbindungsorientierte, verlässliche, Stream-Kommunikation
- **UDP**: unzuverlässige (best-effort) Datagram-Kommunikation

Middleware Schicht

Beobachtung

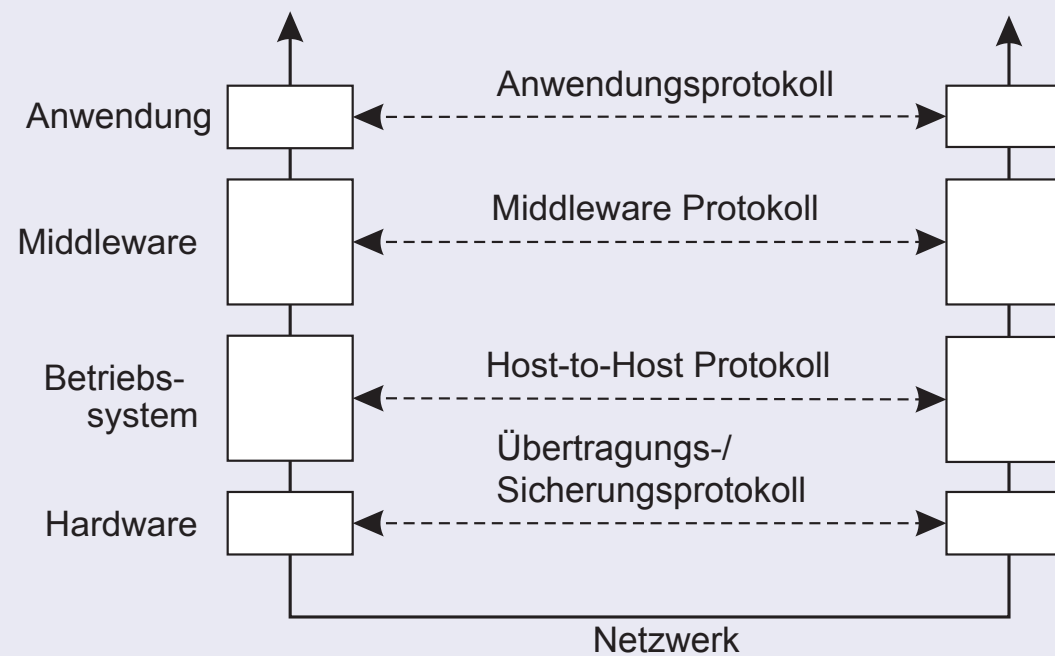
Middleware soll **allgemeine** Dienste und Protokolle bereitstellen, die von vielen **verschiedenen** Anwendungen genutzt werden können

- Vielfältige **Kommunikationsprotokolle**
- **(Un)marshaling** von Daten, für integrierte Systeme
- **Namensprotokolle**, für das Teilen von Ressourcen
- **Sicherheitsprotokolle** für sichere Kommunikation
- **Skalierungsmechanismen**, z.B. für Replikation/Caching

Anmerkung

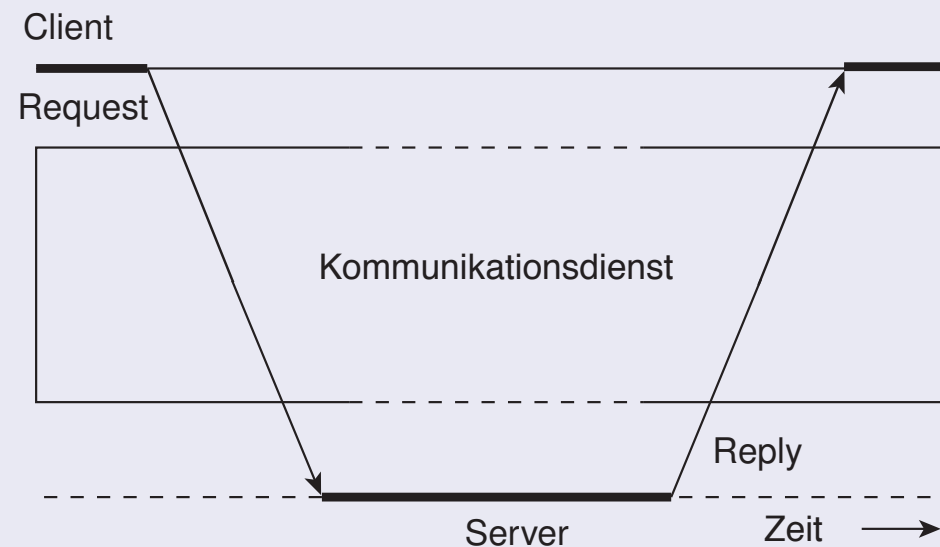
Was bleibt sind wirklich **anwendungsspezifische** Protokolle.

Ein abgewandeltes Schichtenmodell



Arten der Kommunikation

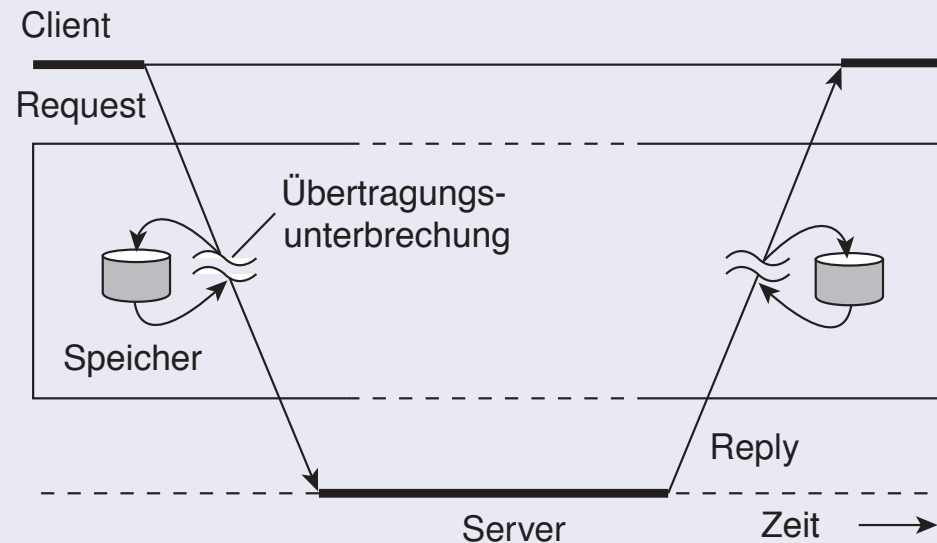
Vier Varianten hinsichtlich zweier Aspekte



- **Transiente** vs. **persistente** Kommunikation
- **Asynchrone** vs. **synchrone** Kommunikation

Arten der Kommunikation

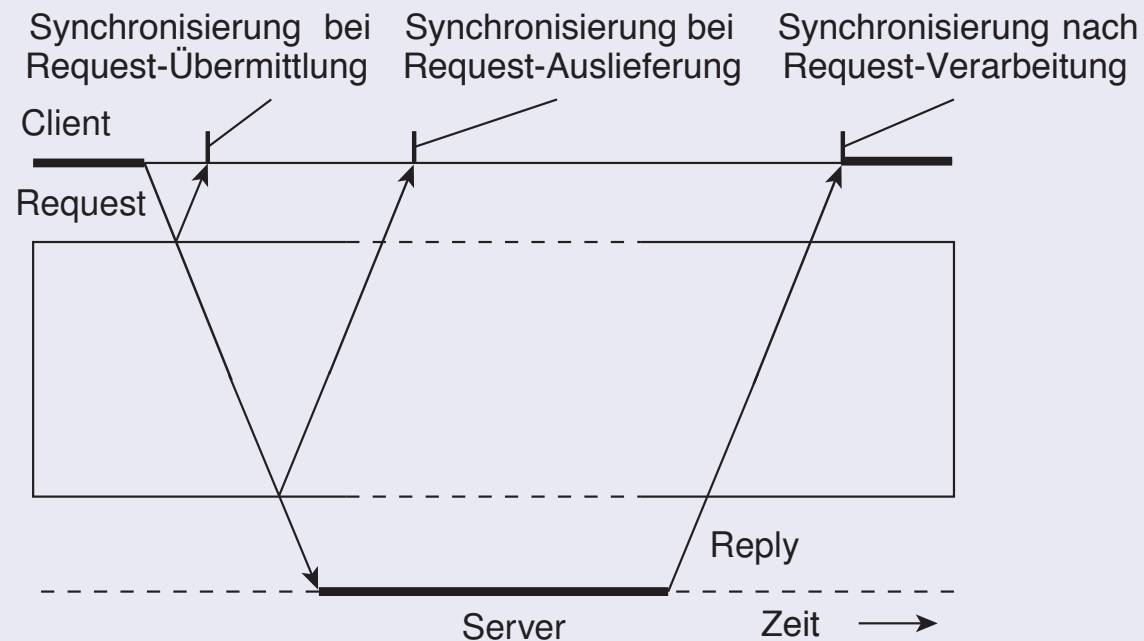
Transient vs. persistent



- **Transiente Kommunikation:** Komm.-Server verwirft Nachricht, wenn sie nicht an den nächsten Server oder Empfänger geliefert werden kann.
- **Persistente Kommunikation:** Nachricht wird beim Kommunikationsserver gespeichert, solange bis die Auslieferung erfolgt ist.

Arten der Kommunikation

Mögliche Punkte für Synchronisation



- Bei Anfrageübermittlung
- Bei Anfrageauslieferung
- Nach Anfrageverarbeitung

Client/Server

Beobachtungen

Client/Server-Verarbeitung basiert generell auf einem Modell **transienter, synchroner Kommunikation**:

- Client und Server müssen gleichzeitig aktiv sein
- Client blockiert nach Anfrageübermittlung bis zur Antwort
- Server wartet auf eingehende Nachrichten und verarbeitet diese

Nachteile synchroner Kommunikation

- Client kann nichts anderes tun während er auf Antwort wartet
- Ausfälle müssen sofort behandelt werden: der Client wartet
- Das Modell ist ggf. nicht passend (Mail)

Nachrichtenkommunikation (Messaging)

Message-oriented Middleware (MOM)

Zielt auf hohe Abstraktionsebene für **persistente asynchrone Kommunikation**:

- Prozesse senden sich Nachrichten, die in einer Warteschlange (Queue) zwischengespeichert werden
- Sender muss nicht auf direkte Antwort warten
- Middleware sichert oft Fehlertoleranz zu



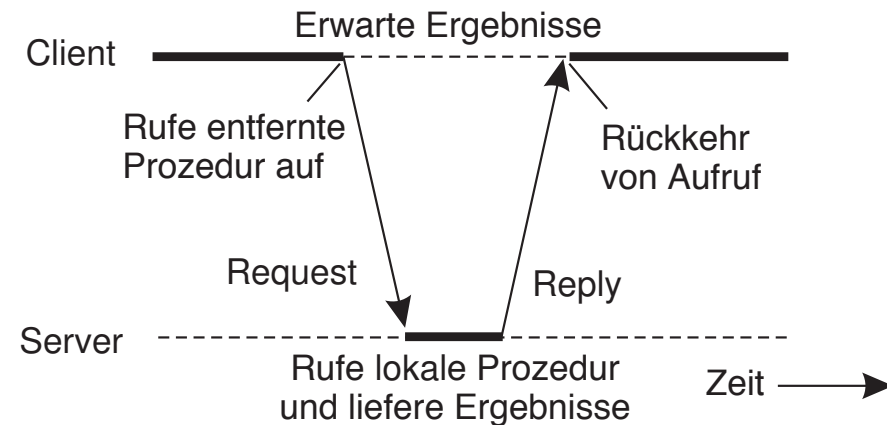
Grundlegende Funktion des RPC

Beobachtungen

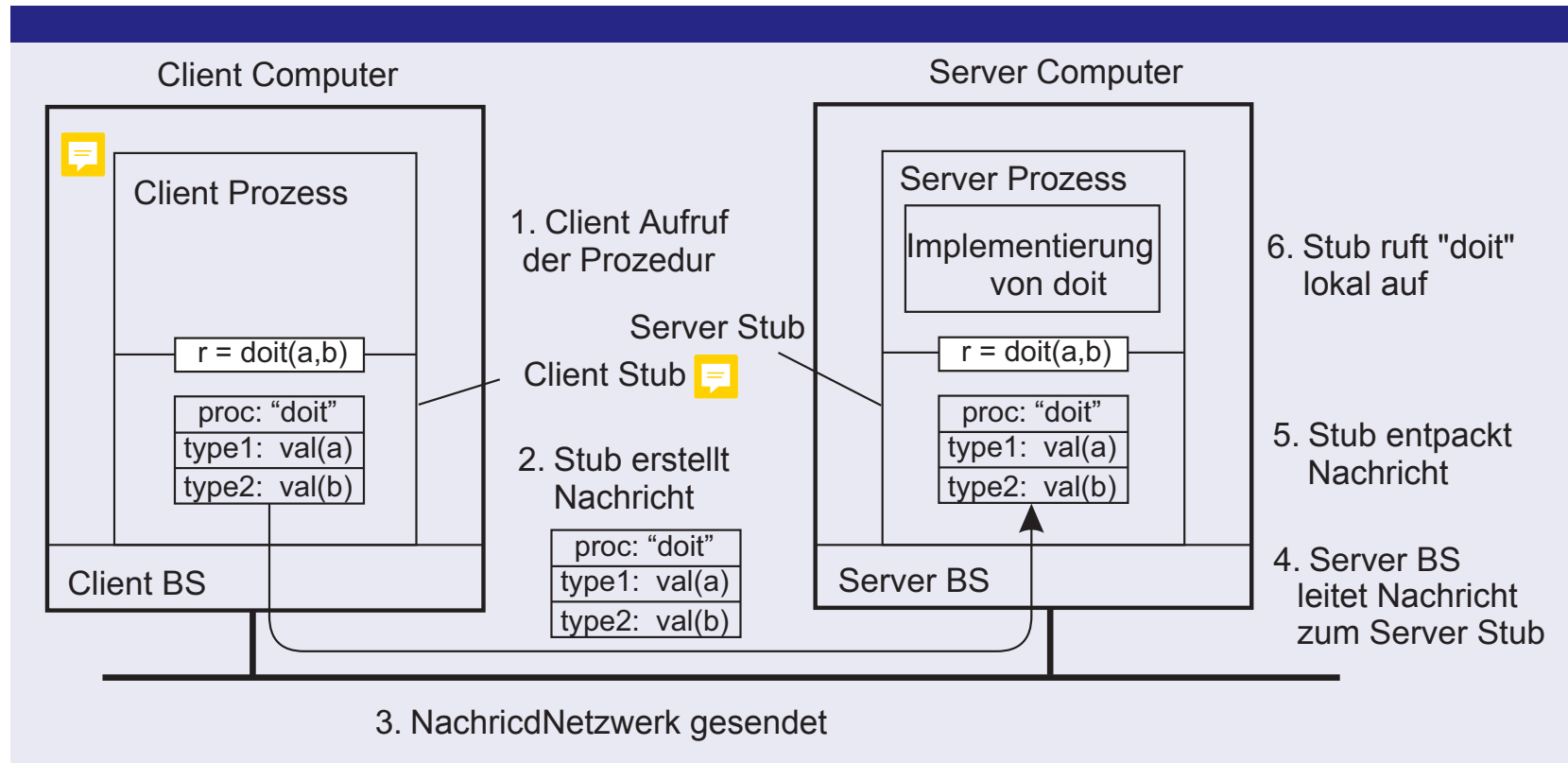
- Anwendungsentwickler kennen einfache Prozeduraufrufe
- Gut konstruierte Prozeduren laufen isoliert ab (Black Box)
- Es gibt keinen entscheidenden Grund gegen Ausführung von Prozeduren auf separaten Rechnern

Folgerung

Kommunikation zwischen Aufrufer & Aufgerufenem kann durch Prozeduraufruf-mechanismus versteckt werden.



Grundlegende Funktion des RPC




- 1 Client Prozedur ruft Client **Stub** auf.
- 2 Stub baut Nachricht; ruft lokales BS auf.
- 3 BS sendet Nachricht an entferntes BS.
- 4 Entferntes BS gibt Nachricht an Stub.
- 5 Stub entpackt Parameter, ruft Server auf.

- 6 Server gibt lokales Ergebnis an Stub.
- 7 Stub baut Nachricht; ruft BS auf.
- 8 BS sendet Nachricht an Client BS.
- 9 Client BS gibt Nachricht an Stub.
- 10 Stub entpackt Ergebnis, gibt es Client.

RPC: Parameterübergabe

Mehr als nur Parameter in eine Nachricht zu verpacken

- Client und Server Rechner haben ggf. **verschiedene Datenrepräsentationen** (man denke an Byte Ordnung) 
- Verpacken von Parametern heißt, **einen Wert in eine Bytesequenz zu transformieren**
- Client und Server müssen sich **auf eine Kodierung einigen**:
 - Wie werden **Basistypen** repräsentiert (Integer, Float, Char)
 - Wie werden **komplexe Typen** repräsentiert (Arrays, Unions)

Folgerung

Client und Server müssen **Nachrichten richtig interpretieren** und sie in maschinenabhängige Repräsentationen transformieren.

RPC: Parameterübergabe

RPC Parameterübergabe: Annahmen

- **Copy in/Copy out** Semantik: während der Prozedurausführung kann nichts über Parameterwerte angenommen werden.
- **Alle** zu verarbeitenden Daten werden als Parameter übergeben. Schließt die Übergabe von **Referenzen auf (globale) Daten** aus.

Folgerung

Komplette Zugriffstransparenz kann nicht realisiert werden.



Beobachtung

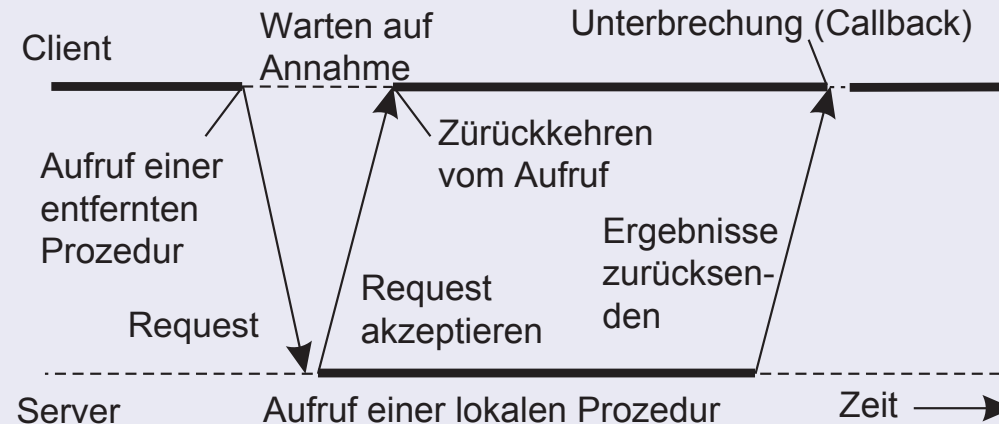
Mechanismus für **entfernte Referenzen** verbessert Zugriffstransparenz:

- Entfernte Referenzen bieten **uniformen Zugriff** auf entfernte Daten.
- Entfernte Referenzen können als **Parameter übergeben** werden.
- **Anmerkung**: z.T. können Stubs als Referenzen dienen.

Asynchrone RPCs


Charakteristik

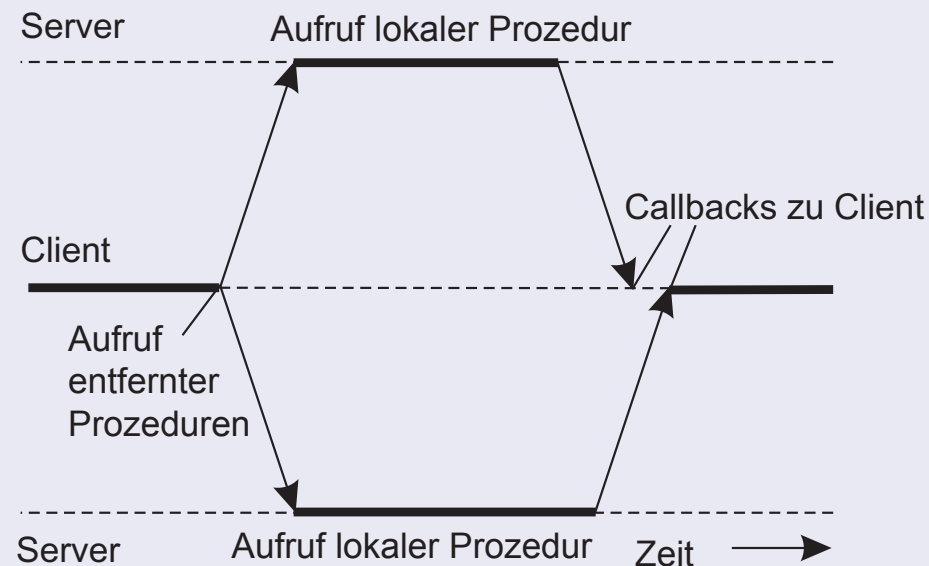
Umgehung des strikten Request-Reply Verhaltens: Clients können weitermachen, ohne auf Antwort vom Server zu warten.



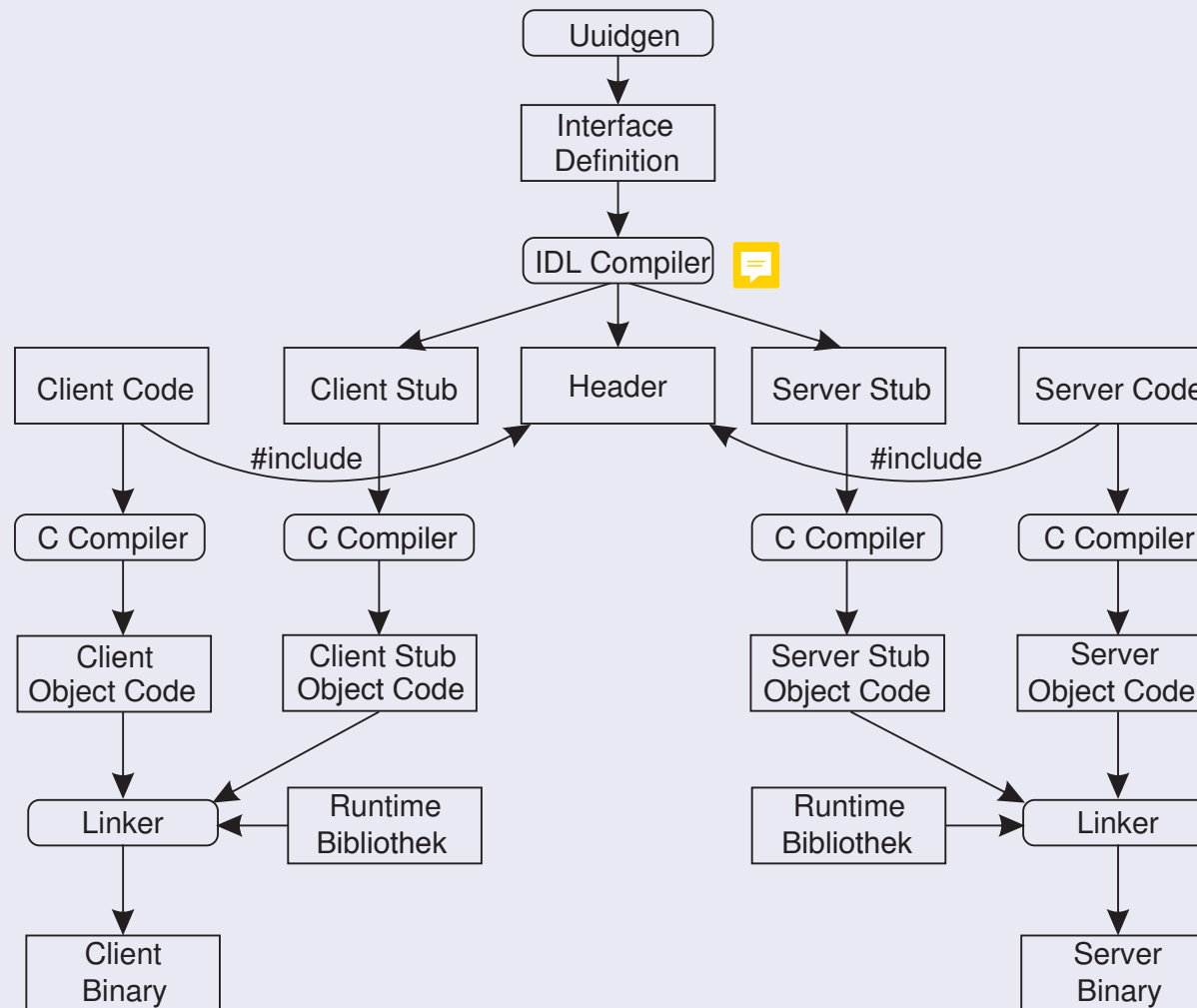
Senden mehrerer RPCs

Charakteristik

Senden eines RPC Requests an eine Gruppe von Servern. 



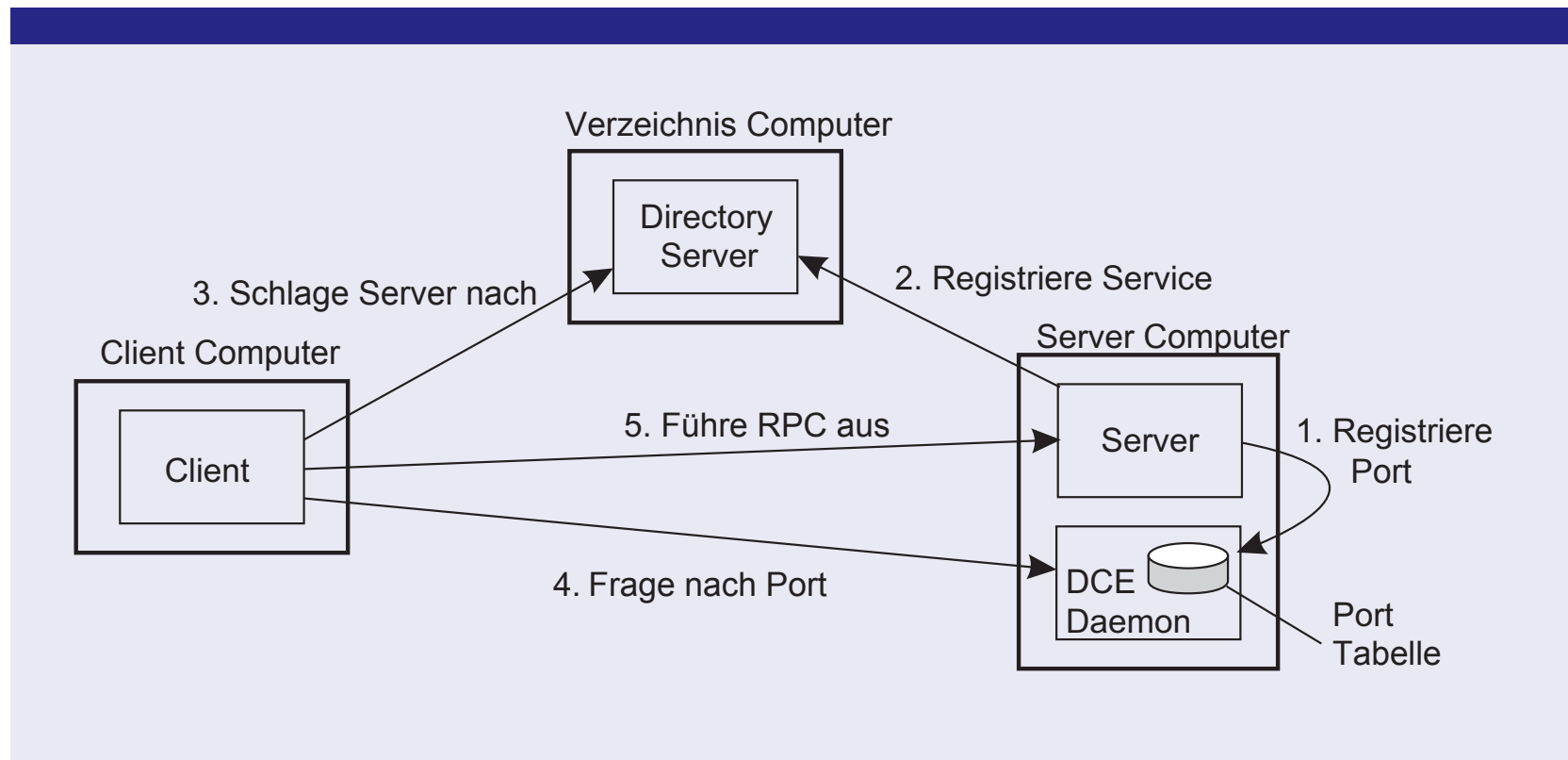
RPC in der Praxis



Client-Server Bindung (Binding) in DCE

Probleme

(1) Client muss Server-Rechner lokalisieren, und (2) Server-Prozess bestimmen (Port).





Übung 4: RPC Semantik

Fallbeispiel: Eine einfache Prozedur: `incr`

Betrachten Sie eine Prozedur `incr` mit zwei ganzzahligen Parametern. Die Prozedur addiert eins zu jedem Parameter.

Nehmen Sie nun an, dass `incr` zweimal mit **derselben** Variable `i` aufgerufen wird, also `incr(i, i)`. Sei `i` anfänglich mit dem Wert 0 initialisiert.

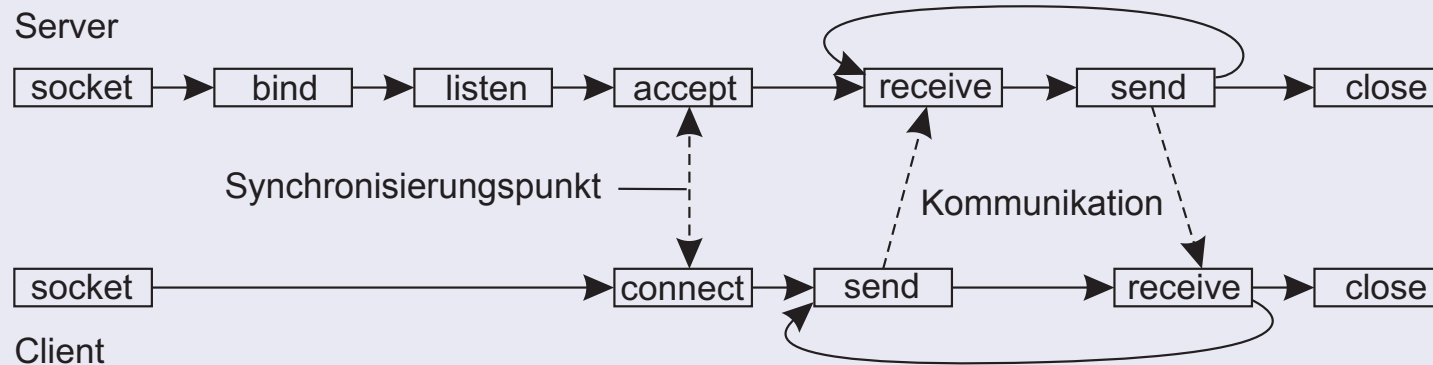
Frage

- 1 Welchen Wert hat `i`, nachdem ein **Aufruf als Referenz** verwendet wird? (Wir wollen hier annehmen, dass eine Ganzzahl als Referenz übergeben und modifiziert werden kann) 
- 2 Was ist das Ergebnis, wenn der Aufruf als RPC erfolgt und die Parameterübergabe mit **Copy in/Copy out Semantik** realisiert ist? 

Transientes Messaging: Sockets

Berkeley Socket Interface

Operation	Beschreibung
socket	Einen neuen Kommunikationsendpunkt erstellen
bind	Lokale Adresse einem Socket zuordnen
listen	Nenne BS max. Zahl ausstehender Verbindungsanfragen
accept	Blockieren, bis eine Verbindungsanfrage eingeht
connect	Aktiver Versuch, eine Verbindung aufzubauen
send	Daten über die Verbindung senden
receive	Daten über die Verbindung empfangen
close	Die Verbindung freigeben



Sockets: Python Code

Server

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

Vereinfachung von Socket Programmierung

Beobachtung

Sockets sind relativ systemnah und Programmierfehler passieren schnell. Auch ist die Verwendung oft ähnlich (z.B. Client-Server).

Alternative: ZeroMQ

Bietet höherwertige Ausdrücke durch **Kopplung** von Sockets: einer zum Senden in Prozess P und ein korrespondierender in Prozess Q zum Empfang. Alle Kommunikation ist **asynchron**.

Drei Muster

- Request-Reply
- Publish-Subscribe
- Pipeline

Request-Reply

Server



```
1 import zmq
2 context = zmq.Context()
3
4 p1 = "tcp://" + HOST + ":" + PORT1 # how and where to connect
5 p2 = "tcp://" + HOST + ":" + PORT2 # how and where to connect
6 s = context.socket(zmq.REP)        # create reply socket
7
8 s.bind(p1)                         # bind socket to address
9 s.bind(p2)                         # bind socket to address
10 while True:
11     message = s.recv()             # wait for incoming message
12     if not "STOP" in message:      # if not to stop...
13         s.send(message + "*")      # append "*" to message
14     else:                           # else...
15         break                      # break out of loop and end
```

Request-Reply

Client

```
1 import zmq
2 context = zmq.Context()
3
4 php = "tcp://" + HOST + ":" + PORT # how and where to connect
5 s = context.socket(zmq.REQ) # create socket
6
7 s.connect(php) # block until connected
8 s.send("Hello World") # send message
9 message = s.recv() # block until response
10 s.send("STOP") # tell server to stop
11 print message # print result
```

Publish-Subscribe

Server

```
1 import zmq, time
2
3 context = zmq.Context()
4 s = context.socket(zmq.PUB)           # create a publisher socket
5 p = "tcp://" + HOST + ":" + PORT     # how and where to communicate
6 s.bind(p)                            # bind socket to the address
7 while True:
8     time.sleep(5)                    # wait every 5 seconds
9     s.send("TIME " + time.asctime()) # publish the current time
```

Publish-subscribe

Client

```
1 import zmq
2
3 context = zmq.Context()
4 s = context.socket(zmq.SUB)           # create a subscriber socket
5 p = "tcp://" + HOST + ":" + PORT     # how and where to communicate
6 s.connect(p)                         # connect to the server
7 s.setsockopt(zmq.SUBSCRIBE, "TIME")  # subscribe to TIME messages
8
9 for i in range(5):                   # Five iterations
10     time = s.recv()                  # receive a message
11     print time
```

Pipeline

Source

```
1 import zmq, time, pickle, sys, random
2
3 context = zmq.Context()
4 me = str(sys.argv[1])
5 s = context.socket(zmq.PUSH)           # create a push socket
6 src = SRC1 if me == '1' else SRC2      # check task source host
7 prt = PORT1 if me == '1' else PORT2    # check task source port
8 p = "tcp://" + src + ":" + prt         # how and where to connect
9 s.bind(p)                             # bind socket to address
10
11 for i in range(100):                  # generate 100 workloads
12     workload = random.randint(1, 100)  # compute workload
13     s.send(pickle.dumps((me, workload))) # send workload to worker
```

Pipeline

Worker

```
1 import zmq, time, pickle, sys
2
3 context = zmq.Context()
4 me = str(sys.argv[1])
5 r = context.socket(zmq.PULL)           # create a pull socket
6 p1 = "tcp://" + SRC1 + ":" + PORT1    # address first task source
7 p2 = "tcp://" + SRC2 + ":" + PORT2    # address second task source
8 r.connect(p1)                         # connect to task source 1
9 r.connect(p2)                         # connect to task source 2
10
11 while True:
12     work = pickle.loads(r.recv())      # receive work from a source
13     time.sleep(work[1]*0.01)           # pretend to work
```

MPI: Wenn mehr Flexibilität benötigt wird

Repräsentative Operationen

Operation	Beschreibung
MPI_bsend	Ausgehende Nachricht an lokalen Sendepuffer anhängen
MPI_send	Sende Nachricht und warte, bis sie in lokalen oder entfernten Puffer kopiert wurde
MPI_ssend	Sende Nachricht und warte, bis die Übertragung beginnt
MPI_sendrecv	Sende Nachricht und warte auf Antwort
MPI_issend	Verweise auf die ausgehende Nachricht und fahre fort
MPI_issend	Verweise auf ausgehende Nachrichten und warte, bis Empfang beginnt
MPI_recv	Empfange Nachricht. blockieren, wenn es keine gibt
MPI_irecv	Überprüfe, ob eine eingehende Nachricht vorhanden ist, blockiere jedoch nicht

Message-oriented Middleware (MOM)

Charakteristik

Asynchrone persistente Kommunikation durch Unterstützung von Middleware **Warteschlangen (Message Queues (MQ))**. Queues ähneln dem Puffer eines Kommunikationsservers.

Operationen

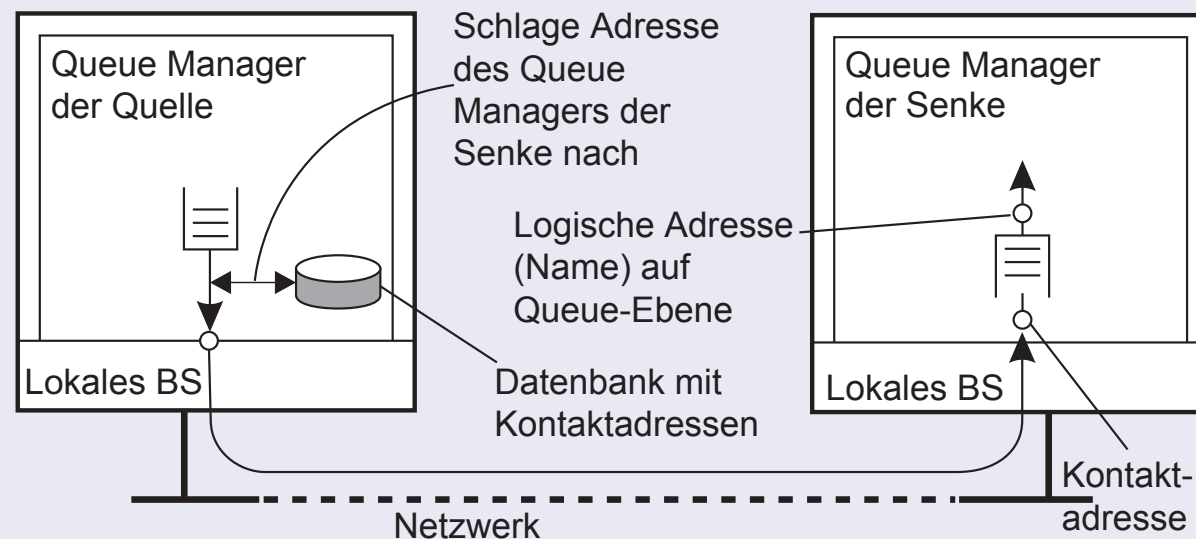
Operation	Beschreibung
put	Nachricht an Warteschlange anhängen
get	Blockieren, bis die Warteschlange nicht leer ist, und erste Nachricht entfernen
poll	Warteschlange auf Nachrichten überprüfen und erste entfernen; niemals blockieren
notify	“Handler” installieren, der aufgerufen wird, wenn Nachricht in Warteschlange gestellt wird

Generelles Modell

Queue Manager

Queues werden von **Queue Managern** verwaltet. Anwendung kann Nachrichten nur in **lokale** Queue einstellen. Eingehende Nachrichten werden immer von lokaler Queue extrahiert \Rightarrow Queue Manager müssen Nachrichten **routen**.

Routing



Message Broker

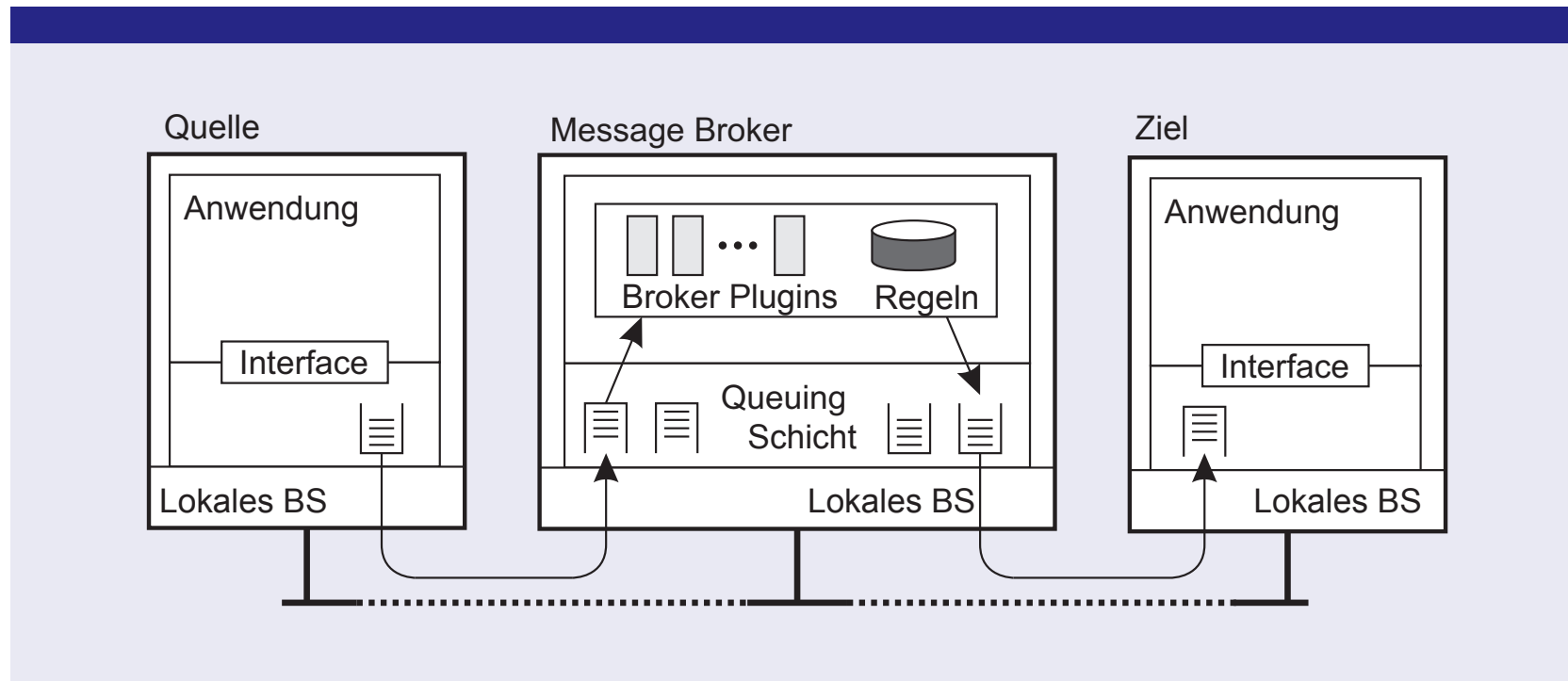
Beobachtung

Warteschlangensysteme erwarten **gemeinsames Protokoll**: alle Anwendungen einigen sich auf ein Nachrichtenformat (Struktur und Datenrepräsentation)

Kümmert sich um Anwendungsheterogenität eines MQ-Systems

- Transformiert eingehende Nachrichten in Zielformat
- Fungiert oft als **Gateway** für Anwendungen
- Bietet ggf. **themenbezogenes** Routing (d.h., **Publish-Subscribe** Mechanismus)

Message Broker: generelle Architektur



IBM WebSphere MQ

Grundkonzepte

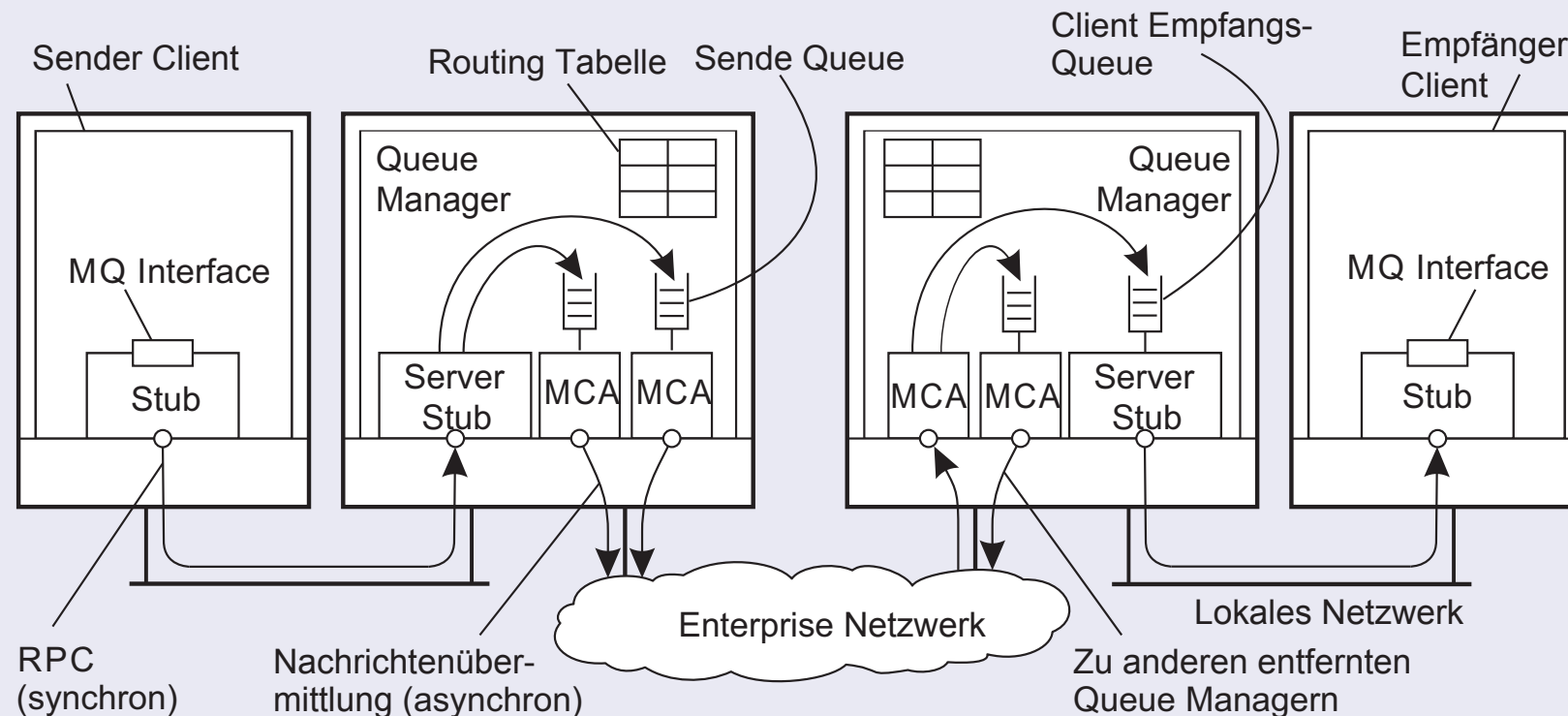
- **Anwendungsspezifische Nachrichten** werden in/aus **Queues** gestellt/entnommen
- Queues befinden sich unter Kontrolle eines **Queue Managers**
- Prozesse können Nachrichten nur in lokale Queues stellen (oder RPC)

Nachrichtenübertragung

- Nachrichten werden zwischen Queues übertragen
- Nachrichtenübertragung zwischen Queues verschiedener Prozesse erfordern **Channel**
- Channel Endpunkte realisiert durch **Message Channel Agent**
- Message Channel Agents sind verantwortlich für:
 - Channel auf Netzwerkebene aufsetzen (z.B., TCP/IP)
 - (Ver/Ent)packen von Nachrichten in/aus Transportschicht-Paketen
 - Senden/Empfangen von Paketen

IBM WebSphere MQ

Schematische Übersicht



- Channels sind inherent unidirektional
- Automatischer Start von MCAs wenn Nachrichten eintreffen
- Beliebige Netze von Queue Managern können erstellt werden
- Routen werden manuell erstellt (Systemadministration)

Message Channel Agents

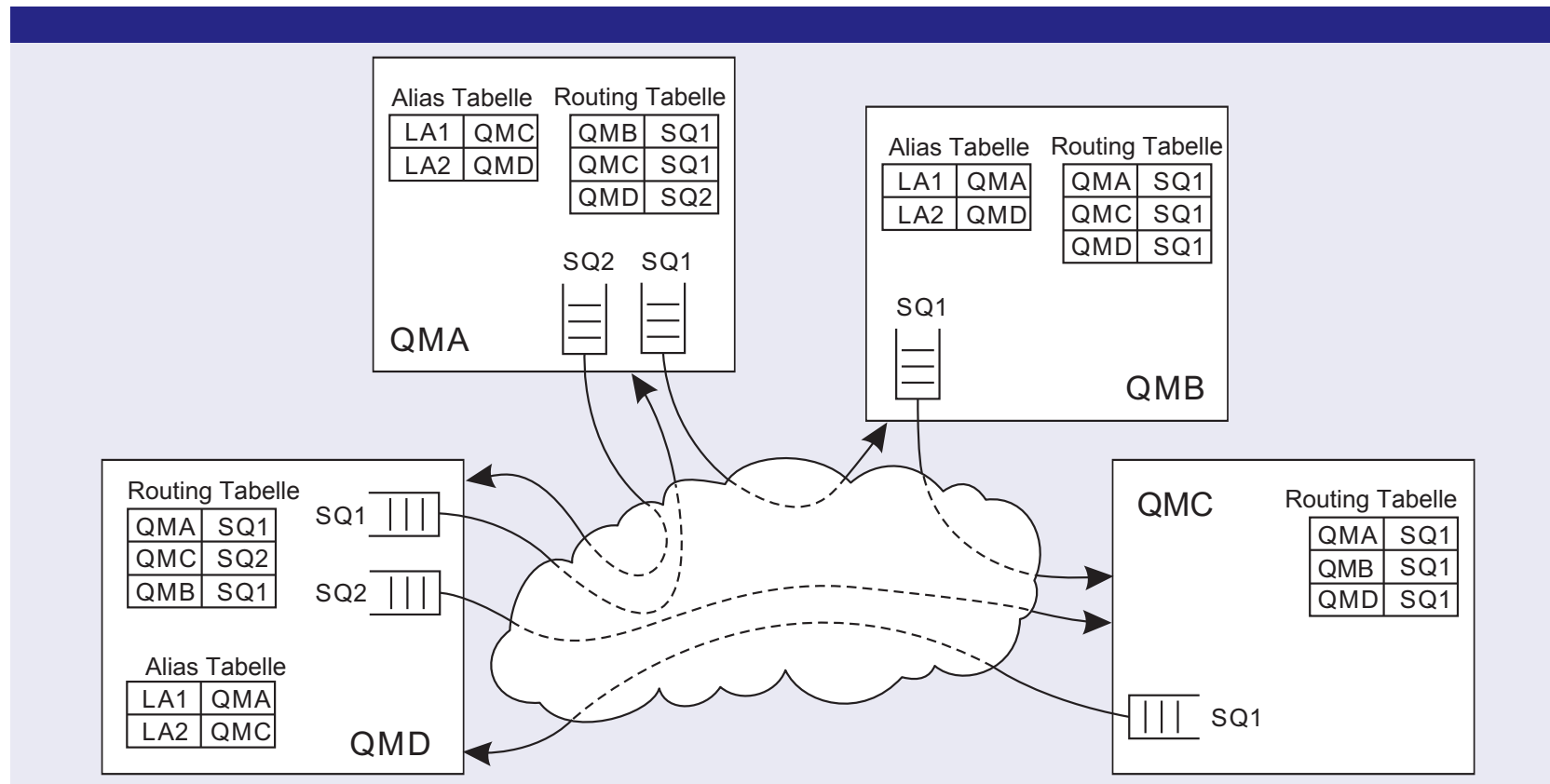
Einige Attribute von Message Channel Agents

Attribut	Beschreibung
Transport type	Bestimmt das Transportprotokoll
FIFO delivery	Zeigt an, dass Nachrichten in der Reihenfolge empfangen werden sollen wie gesendet
Message length	Max Länge einer Nachricht
Setup retry count	Max Anzahl Versuche um entfernten MCA zu starten
Delivery retries	Max Anzahl Versuche von MCA um Nachricht in Queue zu legen

IBM WebSphere MQ

Routing

Durch **logische Namen** mit Namensauflösung auf lokale Queues, können Nachrichten in **entfernte Warteschlange** gestellt werden.





Multicasting auf Anwendungsebene: Application-Level Multicasting (ALM)

Im Kern

Knoten eines verteilten Systems werden als **Overlay Netz** organisiert. Dieses Netz wird dann genutzt, um Daten zu verbreiten:

- Oft ein **Baum**, der zu einzigartigen Pfaden führt 
- Alternativ auch ein **Mesh-Netzwerk**, was eine Form von **routing** erfordert

Application-Level Multicasting in Chord

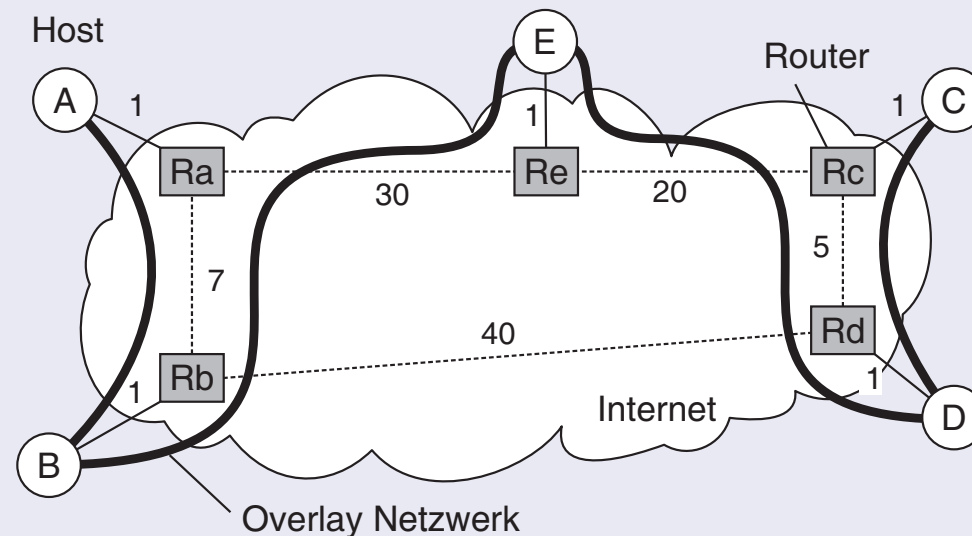
Grundlegender Ansatz

Mit Chord (P2P-Ring aus Kap. 2) einen Multicast Baum konstruieren:

- 1 Initiator generiert **Multicast Bezeichner** *mid*.
- 2 Suche ***succ(mid)***, den Knoten der für *mid* zuständig ist.
- 3 Request wird an ***succ(mid)*** geleitet, der dann **Wurzel** wird.
- 4 Wenn *P* beitreten möchte, sendet er **Beitritt** Request zur Wurzel.
- 5 Wenn Request zu *Q* kommt (nächster Knoten auf Route zur Wurzel):
 - *Q* hat noch keinen Beitritt-Request gesehen \Rightarrow er wird **Versender** (Hilfsknoten); *P* wird Kind von *Q*. **Beitritt-Request wird weitergeleitet zu *mid***.
 - *Q* kennt den Baum schon \Rightarrow *P* wird Kind von *Q*. **Request braucht nicht weitergeleitet zu werden**.

ALM: Einige Kosten

Unterschiedliche Metriken



- **Link Stress:** Wie oft passiert ALM Nachricht denselben physischen Link? **Beispiel:** Nachricht von *A* an *D* muss $\langle Ra, Rb \rangle$ zweimal passieren.
- **Stretch:** Verhältnis der Verzögerung zwischen Pfad auf ALM-Ebenen und Netzwerk-Ebene. **Beispiel:** Nachricht von *B* an *C* folgt Pfad der Länge 73 auf ALM-, aber 47 auf Netzwerk-Ebene $\Rightarrow \text{Stretch} = 73/47$.

Flooding

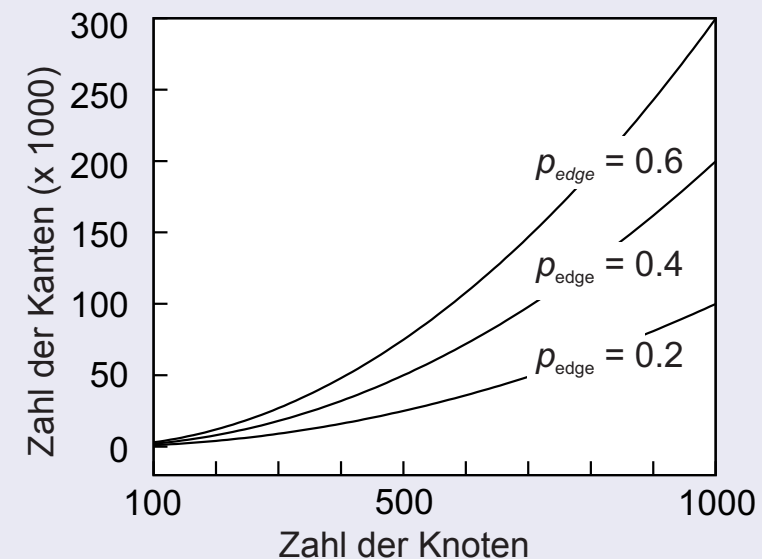
Charakteristik

P sendet Nachricht m an alle Nachbarn. Jeder Nachbar leitet m weiter, außer zu P und nur wenn er m nicht schon kennt.

Performanz

Je mehr Kanten, desto teurer!

Größe zufälliger Overlays als Funktion der Knotenanzahl



Variation

Q leitet Nachricht mit bestimmter Wahrscheinlichkeit p_{flood} weiter, ggf. abhängig von der Anzahl seiner Nachbarn (d.h., **Knotengrad**) oder Grad seiner Nachbarn.

Epidemische Protokolle

Angenommen es gibt keine write–write Konflikte

- Aktualisierungen werden auf einem einzelnen Server ausgeführt
- Replikate leiten aktualisierte Zustände zu wenigen Nachbarn
- Propagierung ist 'lazy', d.h., nicht unmittelbar
- Schließlich erreicht jedes Update jedes Replikat

Two forms of epidemics

- **Anti-Entropy**: Jedes Replikat wählt regelmäßig ein zufälliges anderes Replikat und tauscht Zustandsänderungen aus, was zu identischen Zuständen führt.
- **Gossiping**: Ein Replikat, das gerade aktualisiert (bzw. **infiziert**) wurde, leitet die Aktualisierung an einige andere Replikate weiter (und infiziert auch diese)

Anti-Entropy

Mögliche Operationen

- Knoten P wählt anderen Knoten Q des Systems zufällig.
- **Push**: P sendet Aktualisierungen nur zu Q .
- **Pull**: P empfängt nur Aktualisierungen von Q .
- **Push-Pull**: P und Q **tauschen** wechselseitig Aktualisierungen **aus** (haben dann gleiche Informationen).

Beobachtung

Für push-pull werden $\mathcal{O}(\log(N))$ Runden gebraucht, um Aktualisierungen an alle N Knoten zu verbreiten (**Runde** = jeder Knoten hat einen Austausch gestartet).

Gerüchte verbreiten

Grundmodell

Ein Server S , der eine Aktualisierung zu melden hat, kontaktiert andere Server. Wenn er einen Server kontaktiert, der schon aktualisiert wurde, stoppt S mit Wahrscheinlichkeit p_{stop} .

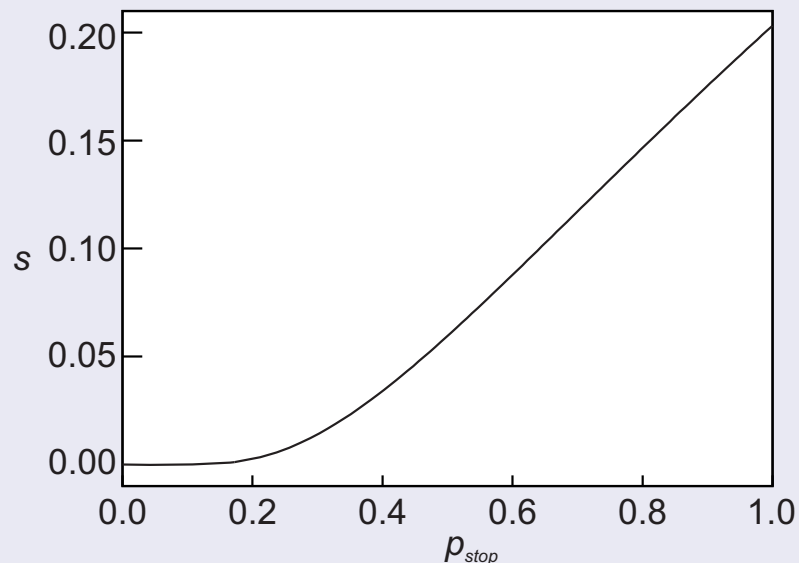
Beobachtung

Wenn s der Bruchteil anfälliger Server ist (die die Aktualisierung nicht kennen), kann gezeigt werden, dass bei vielen Servern gilt:

$$s = e^{-(1/p_{stop}+1)(1-s)}$$

Gerüchte verbreiten

Der Effekt des Stoppens



Angenommen 10,000 Knoten

$1/p_{stop}$	s	N_s
1	0.203188	2032
2	0.059520	595
3	0.019827	198
4	0.006977	70
5	0.002516	25
6	0.000918	9
7	0.000336	3

Bemerkung

Wenn wirklich sichergestellt werden muss, dass alle Server irgendwann aktualisiert werden, reicht Gossiping alleine nicht aus.

Werte Löschen

Fundamentales Problem

Man kann nicht alte Werte vom Server löschen und erwarten, dass das Löschen propagiert wird. Stattdessen wird das Löschen in kurzer Zeit durch den epidemischen Algorithmus rückgängig gemacht.

Lösung

Löschung muss durch spezielle Aktualisierung registriert werden, indem ein **Totenschein** eingesetzt wird.

Werte Löschen

Wann soll Totenschein gelöscht werden? (darf nicht ewig bleiben)

- Führe globalen Algorithmus aus, um zu prüfen, ob Löschen überall bekannt ist und sammle dann Totenscheine ein. (wie Garbage Collection)
- Angenommen Totenscheine verbreiten sich in endlicher Zeit, dann kann max. Lebenszeit mit Zertifikaten assoziiert werden. (Risiko bleibt, nicht alle Server zu erreichen)

Anmerkung

Es ist notwendig, dass das Löschen alle Server erreicht.