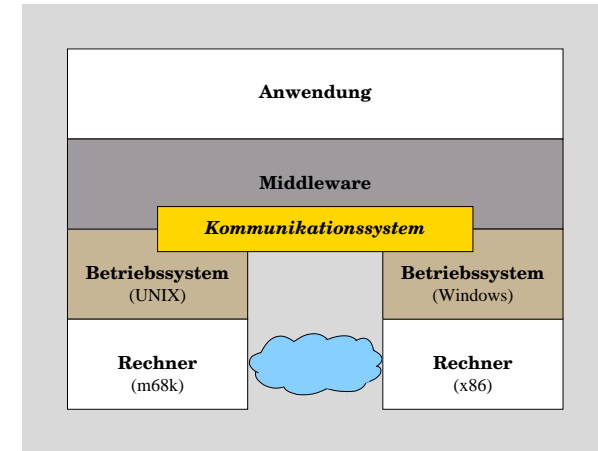


## Überblick

- Interprozesskommunikation
  - Botschaftenaustausch
    - (a)synchron, (un)gepuffert, (nicht)blockierend, (un)zuverlässig
  - IPC-Semantiken und -Varianten, Zustellungsfehler
- Kommunikationsendpunkte
  - Briefkasten, Pforte/Tor, Prozessinkarnation, Prozedur
  - Kommunikationsverlauf
- Zusammenfassung



## Funktionseinheit von Middleware und Betriebssystem

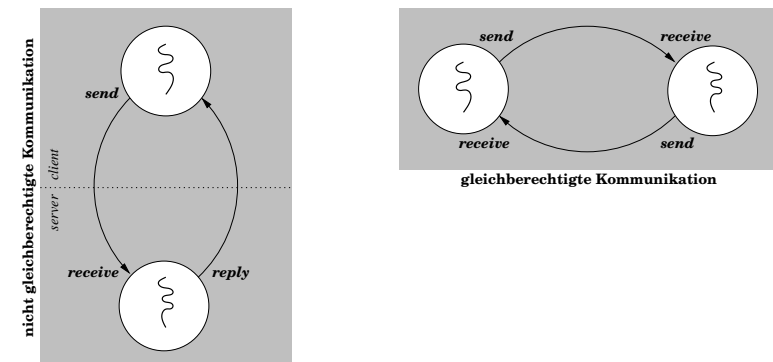


## Interprozesskommunikation

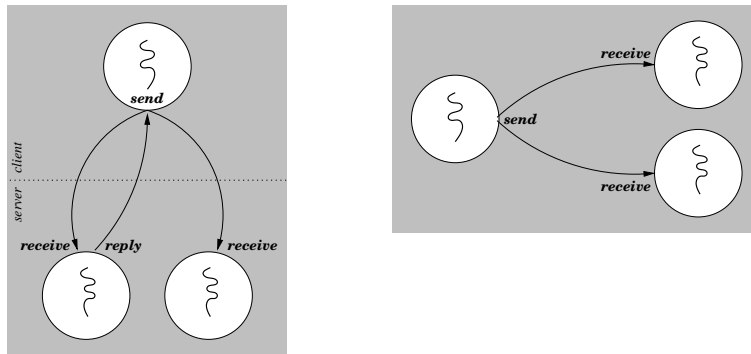
- IPC *inter process communication*
  - als Konsequenz der physikalischen und logischen Verteiltheit
    - d.h., der Trennung der Komponenten eines verteilten Systems
  - die Interaktionen basieren auf **Botschaftenaustausch**
    - d.h., der Übermittlung von Nachrichten (*message passing*)
  - eine **Prozessinkarnation**[4] bildet dabei eine kommunizierende Instanz
    - gleichberechtigte vs. nicht gleichberechtigte Kommunikation
    - Gruppen- bzw. Mehrteilnehmerkommunikation
  - die Mechanismen sind von Plattform zu Plattform unterschiedlich



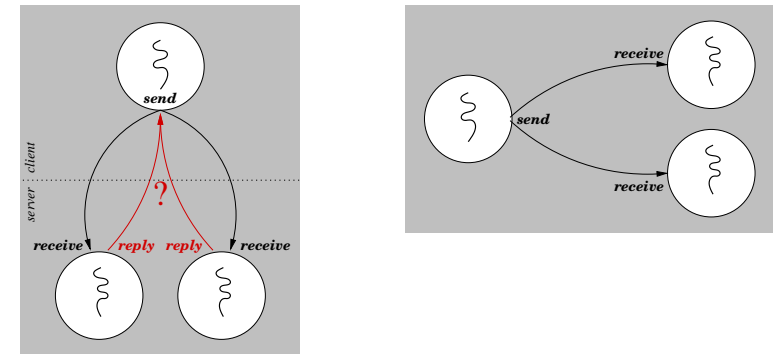
## {,Nicht} Gleichberechtigte Kommunikation



## Gruppenkommunikation (1)



## Gruppenkommunikation (2)



## IPC — Prinzipielle Aktionen

- Datentransfer vom Sende- zum Empfangsprozessadressraum
  - Botschaftenaustausch über einen gemeinsamen Kommunikationskanal
- Synchronisation von Sende- und Empfangsprozess
  - der Fortschritt des Empfangsprozesses hängt ab vom Sendeprozess
    - die Nachricht ist ein konsumierbares Betriebsmittel
    - der Sendeprozess muss dem Empfangsprozess eine Nachricht zustellen
  - der Fortschritt des Sendeprozesses hängt ab vom Empfangsprozess
    - der Nachrichtenpuffer ist ein wiederverwendbares Betriebsmittel
    - der Empfangsprozess muss Nachrichten verarbeiten und Puffer entsorgen
  - die Koordination geschieht implizit mit der angewandten Primitive

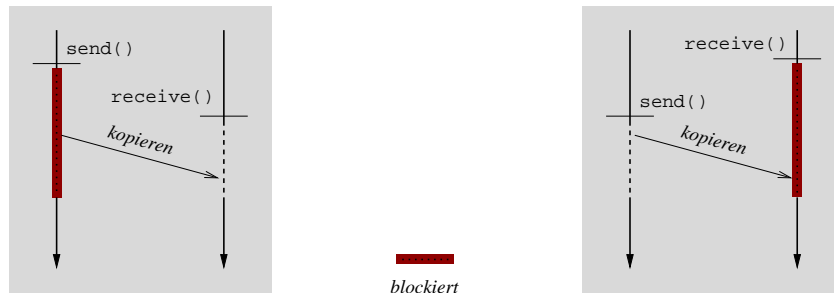


## Botschaftenaustausch — *Message Passing*

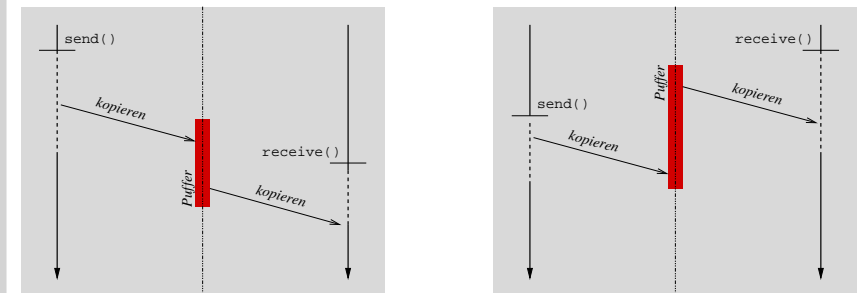
- synchron und blockierend
  - der Sender wartet passiv im `send()` auf das `receive()` des Empfängers
  - der Empfänger wartet passiv im `receive()` auf das `send()` des Senders
  - Unterstützung von End-zu-End Datentransfers ohne Zwischenpufferung
- asynchron und blockierend oder nicht-blockierend
  - passives Warten im `send()` bzw. `receive()` erfolgt ausnahmsbedingt
    - im Falle von Zwischenpufferung der Nachrichten (*bounded buffer*)
    - zur Abwendung von Pufferüber- und/oder -unterlauf
  - Unterstützung bzw. Ausnutzung von Fließbandverfahren (*pipelining*)



## IPC — Synchrone Kommunikation



## IPC — Asynchrone Kommunikation



## IPC — *copy on write* — COW

- Kommunikation beeinflusst die „Fähigkeit“ (*capability*) von Prozessen:
  - `send()` entzieht dem Sender das Schreib- und übergibt dem Empfänger das Lesezugriffsrecht auf die Nachricht
  - `receive()` beansprucht das Lesezugriffsrecht auf die Nachricht durch den Empfänger
- Kopieren wird dadurch zum **Ausnahmefall** [7]: *segment swapping* bzw. *paging*
  - wenn der Sender nach dem `send()` die Nachricht zu überschreiben wünscht
  - wenn der Empfänger nach dem `receive()` die Nachricht zu schreiben wünscht
- **Nachrichten** müssen vom Betriebssystem als **Segmente** verwaltet werden



## Blockierende vs. Nichtblockierende Kommunikation

- die Blockade synchronisiert den Prozess auf die Betriebsmittelbereitstellung
  - Sender** benötigt ein wiederverwendbares Betriebsmittel „Puffer“
    - synchroner IPC  $\Rightarrow$  in den Zielpuffer
    - asynchroner IPC  $\Rightarrow$  in den Zwischenpuffer
  - Empfänger** benötigt ein konsumierbares Betriebsmittel „Nachricht“
    - synchroner IPC  $\Rightarrow$  aus dem Quellpuffer
    - asynchroner IPC  $\Rightarrow$  aus dem Zwischenpuffer
- beide Verfahren haben ihre Vor- und Nachteile, sie ergänzen sich einander



- gepuffert nach Übergabe an das Kommunikationssystem kann der durch die Nachricht belegte Speicherbereich wiederverwendet werden
  - Kopieraufwand ist durch Einsatz von *Wechsellufferverfahren* vermeidbar
- ungepuffert eine sofortige Wiederverwendung des durch die Nachricht belegten Speicherbereichs ist konfliktbehaftet
  - ein *Signal* zeigt an, dass eine Konfliktgefahr nicht (mehr) besteht<sup>1</sup>

Beide Verfahren wirken sich **blockierend** auf den Sendeprozess aus, wenn (a) ein *Pufferdeskriptor* als Betriebsmittel nicht verfügbar ist und (b) die dadurch entstehende Ausnahmesituation nicht zum Scheitern der Operation führen soll.



- gepuffert nach Übergabe an das Kommunikationssystem kann der durch die Nachricht belegte Speicherbereich von einem anderen Faden desselben Programms wiederverwendet werden
  - das Programm kann problemlos ausgelagert werden (*swapping, paging*)
- ungepuffert der Datentransfer kann End-zu-End, d.h., direkt zwischen dem Send- und Empfangsadressraum stattfinden
  - der Kommunikationsvorgang läuft (im System) sehr effizient ab

Beide Verfahren wirken sich **blockierend** für den Sendeprozess aus. Die Operation kann nicht wegen Betriebsmittelmangel scheitern, da der Sendeprozess zu einem Zeitpunkt nicht mehr als einen Kommunikationsvorgang auslösen kann.



## Nichtblockierende Kommunikation

- alle erforderlichen Betriebsmittel werden „von oben“ geliefert:
  - **Puffer** direkt von der Anwendungsebene oder von der Systemebene, die gepufferte Kommunikation implementiert
  - **Pufferdeskriptor** von der Systemebene, die ungepufferte Kommunikation implementiert
- die *Kommunikationsbetriebsmittel* werden lediglich „delegiert“
  - sie werden Verarbeitungseinheiten (Protokollmaschine, Treiber) zugeführt
  - ihre Freigabe wird den Prozessen signalisiert
- den Prozessen obliegt es, ggf. auf diese Signale („oben“) zu warten



## IPC Semantiken

- **no-wait send** der Sendeprozess wartet, bis die Nachricht im Transportsystem zum Absenden bereitgestellt worden ist
  - *Pufferung* oder *Signalisierung* (dass der übergebene Puffer wieder frei ist)
- **synchronization send** der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess angenommen worden ist
  - *Rendezvous* zwischen Send- und Empfangsprozess
- **remote-invocation send** der Sendeprozess wartet, bis die Nachricht vom Empfangsprozess verarbeitet und beantwortet worden ist
  - *Fernauf* einer vom Empfangsprozess auszuführenden Funktion



## IPC Varianten (1)

*non-blocking send* → *no-wait send*

*blocking send* der Sendeprozess wartet, bis die Nachricht den Rechner verlassen hat, d.h., bis sie ausgegeben und ins Netz eingespeist worden ist

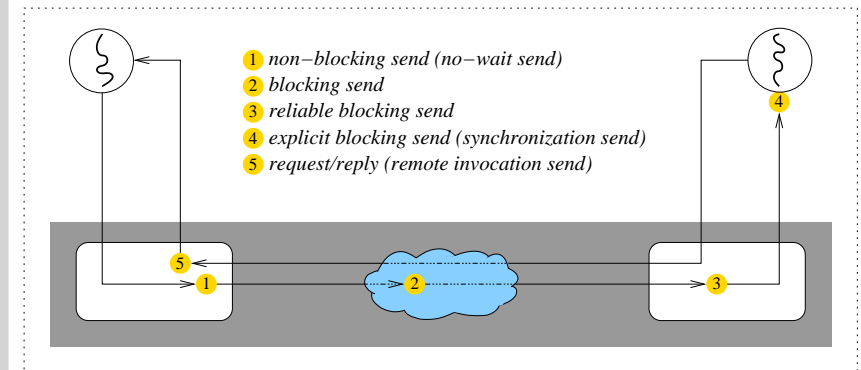
*reliable-blocking send* der Sendeprozess wartet, bis die Nachricht beim Empfangsrechner eingetroffen ist bzw. von dem den Empfangsprozess verwaltenden Betriebssystem angenommen worden ist

*explicit-blocking send* → *synchronization send*

*request/reply* → *remote-invocation send*



## IPC Varianten (2)



## IPC vs. Fernaufrufe

- miteinander kommunizierende Prozesse kennen die *Bedeutung* der Nachrichten
  - sie ist ihnen *implizit* durch den Verarbeitungsalgorithmus bewusst oder
  - sie machen sie sich gegenseitig *explizit* über „Anweisungen“ bekannt
- die Nachrichten enthalten (problemspezifische) Daten und/oder Text:
  - function shipping* der Empfangsprozess interpretiert Programme
    - mobiler Code (Java Bytecode, PostScript) ggf. mit Daten unterfüttert
  - data shipping* der Empfangsprozess interpretiert Daten
- im „Normalfall“ bewirken Nachrichten die Ausführung entfernter Routinen
  - die aufzurufenden Prozeduren/Funktionen sind implizit oder explizit kodiert



## IPC Protokolle für Fernaufrufe (1)

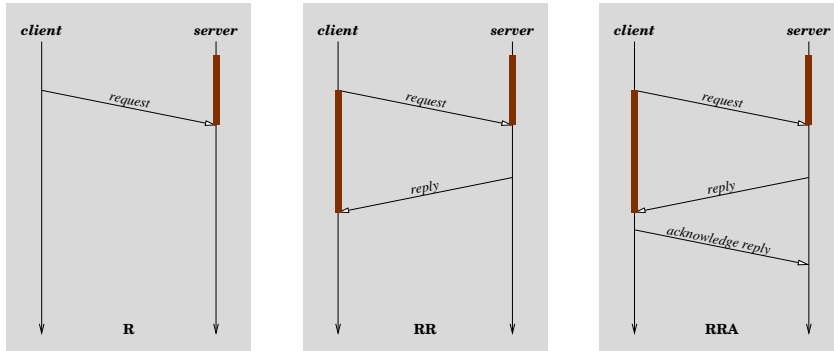
**request (R)** kann genutzt werden, wenn die entfernte Prozedur/Funktion keinen Rückgabewert liefert und der Sendeprozess keine Bestätigung für die erfolgte Ausführung benötigt  
1 Nachricht

**request-reply (RR)** ist das geläufige Verfahren, da die Antwortnachricht implizit die Anforderungsnachricht bestätigt und dadurch explizite Bestätigungen entfallen . . . . . 2 Nachrichten

**request-reply-acknowledge reply (RRA)** gestattet es, die zum Zwecke der *Fehlermaskierung* (beim Server) gespeicherten Antwortnachrichten zu verwerfen, wenn (vom Client) keine weitere Anforderungsnachricht gesendet wird . . . . . 3 Nachrichten



## IPC Protokolle für Fernaufrufe (2)

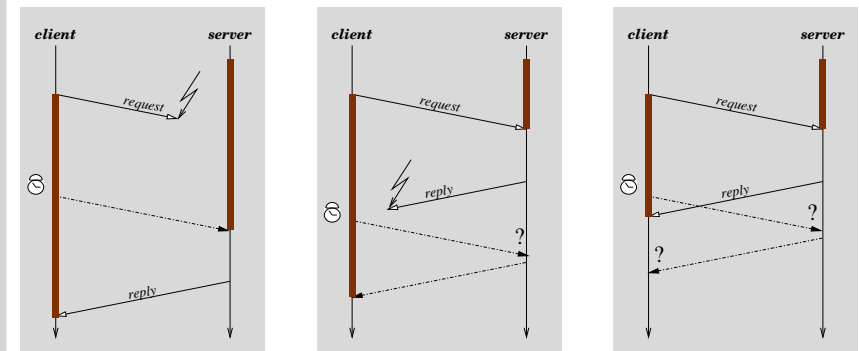


## Zustellungsfehler

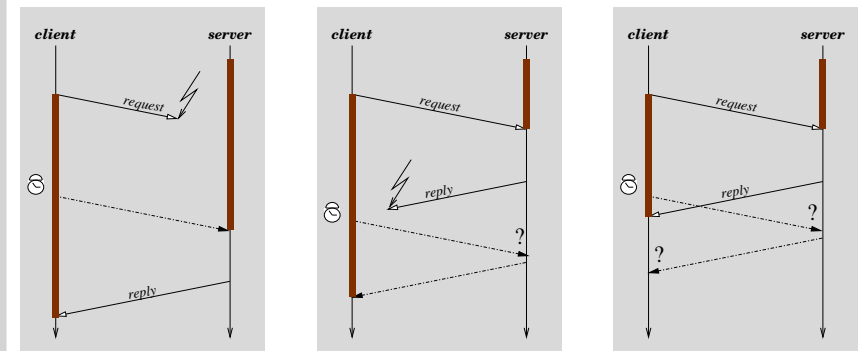
- Botschaftenaustausch unterliegt bestimmten (typischen) *Fehlerannahmen*:
  1. Nachrichten können verloren gehen
    - beim Sender, beim Empfänger oder im Netz
  2. Netzwerke können sich partitionieren
    - ein oder mehrere Rechner (Knoten) werden „abgetrennt“
  3. Prozesse können scheitern (d.h. „abstürzen“)
    - Prozess-, Rechner- oder Netzwerkausfälle sind nicht unterscheidbar
  4. Daten können verfälschen
- als Folge sind unterschiedliche (typische) *Protokollvarianten* [5] entstanden

## Fehlermaskierung (1)

- Anforderungs- und ggf. auch Antwortnachrichten wiederholen
  - nach einer Pause (*time-out*) werden die Nachrichten erneut versendet
  - die „optimale“ Länge der Pause zu bestimmen ist äußerst schwierig
- eingetroffene Nachrichtenduplikate sind zu erkennen und zu ignorieren
  - ggf. bereits versandte Antwortnachrichten wiederholt versenden
  - auf Client- bzw. Server-Seite ist ggf. „*duplicate supression*“ anzuwenden
- *idempotente Operationen*/Zustandsfreiheit tolerieren  
Anforderungsduplikate



## Fehlermaskierung (2)



## Kommunikationsendpunkt (1)

- der Bezeichner (*identifier*) des Bestimmungsortes einer Nachricht
  - seine „Bedeutung“ ist je nach Anforderung/Auslegung orts{,un}abhängig
  - ein *trade-off* zwischen Performanz und Flexibilität bzw. Transparenz
- je nach Modell werden darüber unterschiedliche Instanzen identifiziert:

**Prozedur** das die Nachricht verarbeitende passive Objekt ..... *method*

**Prozessinkarnation** das verarbeitende aktive Objekt ..... *thread*

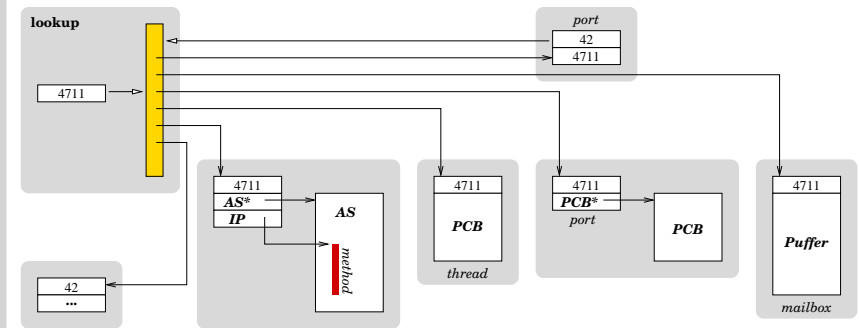
**Tor** das Nachricht weiterleitende Objekt ..... *port*

**Briefkasten** das die Nachricht zwischenspeichernde Objekt ..... *mailbox*

- sein Wert muss (für eine gewisse Zeitdauer) **systemweit eindeutig** sein



## Kommunikationsendpunkt (2)

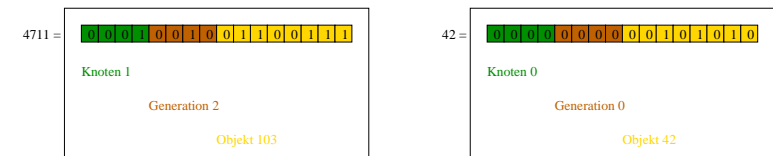


## Eindeutigkeit

- dem Bezeichner ist ein Wert zu geben, der Mehrdeutigkeiten ausschließt:
  - **Zufallszahl** steht und fällt mit der Güte des Zufallszahlengenerators
  - **Zeitstempel** setzt eine einheitliche Zeitbasis voraus
  - **Prozessornummer** die „Hardware-Adresse“ ist abhängig vom Hersteller
- auch lokal ist bereits die eindeutige Wertzuordnung erforderlich:
  - **Unikat** eine nur einmal vergebene Objektidentifikation → z.B. *pid*
  - **Generationsnummer** zählt die Wiedervergabe einer Objektidentifikation
  - **Adresse** des Objektes im Hauptspeicher
- der *Eindeutigkeitsgrad* hängt stark ab vom gewählten Wertebereich → Y2K



## Strukturierte Bezeichner



- die Struktur ist im Regelfall „nach außen“ nicht sichtbar, sie ist transparent
  - „nach innen“ gestattet sie ein effizientes Auffinden des Bestimmungsortes
- der Bezeichner bleibt damit ortsunabhängig, enthält jedoch „Ortshinweise“



## Kommunikationsverlauf

**direkt** der Bezeichner identifiziert eine Prozessinkarnation oder Prozedur

- Nachrichten werden direkt einem Adressraum bzw. Prozess zugestellt

**indirekt** der Bezeichner identifiziert ein Tor oder einen Briefkasten

- Nachrichten werden einem Prozess indirekt über ein Tor zugestellt
- Prozesse nehmen Nachrichten indirekt über Briefkästen in Empfang

**verbindungsorientiert** der Bezeichner identifiziert ein Tor

- die Verbindungen bestehen zwischen Toren: einem Sende- und Empfangstor
- der Verbindungsaufbau dient u.a. der *Betriebsmittelreservierung*



## Direkte Kommunikation (1)



- adressiert wird die **entfernte Prozedur** zur *Nachrichtenverarbeitung*
  - die Botschaft wird als aktive Nachricht [6] verschickt und behandelt
  - die Verarbeitung erfolgt nebenläufig zum entfernten aktiven Prozess
- das Schema unterstützt weder Migrationstransparenz noch Fehlertransparenz



## Direkte Kommunikation (2)



- adressiert wird der **entfernte Prozess** zur *Nachrichtenverarbeitung*
  - den Moment der Nachrichtenannahme bestimmt die *Fadeneinplanung*
  - die Verzögerung (*scheduling latency*) bedingt eine Zwischenlagerung
- das Schema unterstützt weder Migrationstransparenz noch Fehlertransparenz



## Indirekte Kommunikation (1)



- adressiert wird das **entfernte Tor** zur *Nachrichtenweiterleitung*
  - der Empfangsprozess ist „lose“ mit einem *Eingangstor* gekoppelt
  - die Bindung ist dynamisch und kann sich auf mehrere Eingangstore beziehen
- das Schema unterstützt Migrationstransparenz, aber Fehlertransparenz nicht





## Indirekte Kommunikation (2)



- adressiert wird der **entfernte Briefkasten** zur *Nachrichtenspeicherung*
  - mehrere Prozesse können sich denselben Briefkasten teilen
  - typisch ist die *mehrfädige Verarbeitung* eingegangener Nachrichten
- das Schema unterstützt weder Migrationstransparenz noch Fehlertransparenz



## Verbindungsorientierte Kommunikation (1)



- adressiert wird das **lokale Tor** zur *Nachrichtenweiterleitung*
  - die Bindung zwischen *Ausgangstor* und entferntem Prozess ist dynamisch
  - der Bezeichner des entfernten Prozesses kann als *Replik* verteilt vorliegen
- das Schema unterstützt Migrationstransparenz und Fehlertransparenz



## Verbindungsorientierte Kommunikation (2)



- adressiert wird das lokale Tor zur *Nachrichtenweiterleitung* über eine **Torkette**
  - die dynamische Bindung „*Ausgangstor* zu *Eingangstor*“ ist 1 : 1 oder  $N : 1$
  - durch Rechnerausfälle ggf. aufgebrochene Ketten sind wieder zu schließen
- das Schema unterstützt Migrationstransparenz und Fehlertransparenz



## Zusammenfassung

- Interprozesskommunikation ist in vielfältiger Art und Weise möglich:
  - {,a}synchron, {,un}gepuffert, {,nicht}blockierend, {,un}zuverlässig
  - {no-wait,blocking,reliable-blocking,synchronization,remote-invocation} send
  - {request,request-reply,request-reply-acknowledge reply} Protokoll
- Kommunikationsendpunkte können sehr unterschiedliche Bedeutungen besitzen
  - Bezeichner für Tore, Briefkästen, Prozessinkarnationen und/oder Prozeduren
  - systemweite Eindeutigkeit ist (je nach Anwendungsszenario) zu gewährleisten
- die Kommunikation verläuft direkt, indirekt oder verbindungsorientiert





W. M. Gentleman.

Message Passing between Sequential Processes: The Reply Primitive and the Administrator Concept.  
*Software Practice and Experience*, 11(5):435–466, May 1981.



B. H. Liskov.

Primitives for Distributed Computing.  
In *Proceedings of the Seventh ACM Symposium on Operating System Principles*, volume 13 of *Operating Systems Review*, pages 33–42, 1979.



J. H. Saltzer, D. P. Reed, and D. D. Clark.

End-To-End Arguments in System Design.  
*Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.



W. Schröder-Preikschat.

Betriebssysteme.  
<http://www4.informatik.uni-erlangen.de>, 2002.



A. Z. Spector.

Performing Remote Operations Efficiently on a Local Computer Network.  
*Communications of the ACM*, 25(4):246–260, 1982.



T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser.

Active Messages: A Mechanism for Integrated Communication and Computation.  
Technical Report UCB/CSD 92/675, University of California, Berkeley, CA, 1992.



M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron.

The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System.  
In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, volume 21 of *Operating Systems Review*, pages 63–76, Austin, TX, 1987.

