

AI-B.41: Verteilte Systeme

- Lecture Notes [SL] -

VII. Interprozesskommunikation [I]

C. Schmidt | SG AI | FB 4 | HTW Berlin

Stand: WiSe 18

Urheberin: Prof. Dr. Christin Schmidt

Verwertungsrechte: keine außerhalb des Moduls

Rückblick letzte Veranstaltung

Nach dieser Lehrveranstaltung kennen Studierende idealerweise:

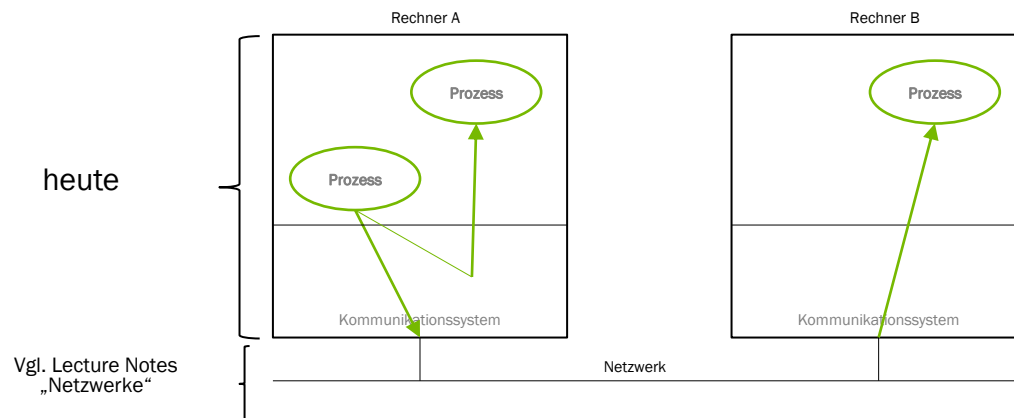
- Den Einfluss von Netzwerken auf das Kommunikations-Teilsystem eines verteilten Systems
 - Anforderungen an Netzwerke für verteilte Systeme
 - Unterscheidungskriterien für Netzwerke
 - Die Bedeutung von Protokollen für die Netzwerkkommunikation
 - Unterschiedliche Protokoll-Schichten des Open Systems Interconnection Reference Models (OSI)
 - Protokoll-Schichten des TCP/IP-Stacks im Kontext der OSI-Layers
 - Ausgewählte Aspekte im Zusammenhang mit Protokoll-Schichten am Beispiel TCP/IP
 - Ausgewählte Beispiele zu den o.g. Themen
- ✓ Studierende lernen grundlegende Aspekte im Themenbereich Netzwerke kennen, da verteilte Systeme per definitionem die Existenz eines Netzwerkes voraussetzen und sich je nach Protokollfolge Implikationen für den Entwurf und die Implementierung verteilter Systeme auf verschiedenen Schichten ergeben.

Ausgangspunkt

vgl. Weber [1998:21ff.]



Wir erinnern uns (vgl. Lecture Notes „Netzwerke“)

- Um die ausgetauschten Informationen in einem Netzwerk verarbeiten zu können, müssen diese an einen entsprechenden Prozess gerichtet werden (Interprozesskommunikation)
- Soll Interprozesskommunikation lokationstransparent erfolgen, so muss sie immer über ein Kommunikationssystem abgewickelt werden, da die Aufrufe zur Kommunikation gleich sein sollen, egal ob der Empfangsprozess lokal oder auf einem anderen Rechner liegt
- ✓ Fokus der letzten Lehrveranstaltung: Kommunikation über das Netzwerk (zweier Computer ohne gemeinsamen Speicher) und damit verbundenen Technologien, Konzepten (und Hardwarekomponenten)
- ✓ **Fokus heute:** Software des Kommunikationssystems (Programme und Prozesse, die untereinander Informationen austauschen).



Quelle: Weber [1998:21]

Roadmap (Stand: 5 | 2018)

	SL
1	Organisatorisches, Einführung
2	Systemmodelle & Architekturstile
3	Systemarchitekturen I: C/S
4	Systemarchitekturen II: P2P
5	Cloud Computing
6	Netzwerke
 	7 Interprozesskommunikation I
8	Interprozesskommunikation II - Verteilte Objekte und entfernter / externer Aufruf
9	Synchronisation und Koordination I - Nebenläufigkeit und Threads
10	Synchronisation und Koordination II - Zeit
11	Synchronisation und Koordination III- Ausgewählte Algorithmen
12	Sicherheit und Schutz
13	Ausgewählte Themen I
14	Ausgewählte Themen II
15	Prüfungsvorbereitung/Klausurtipps

Lernziele

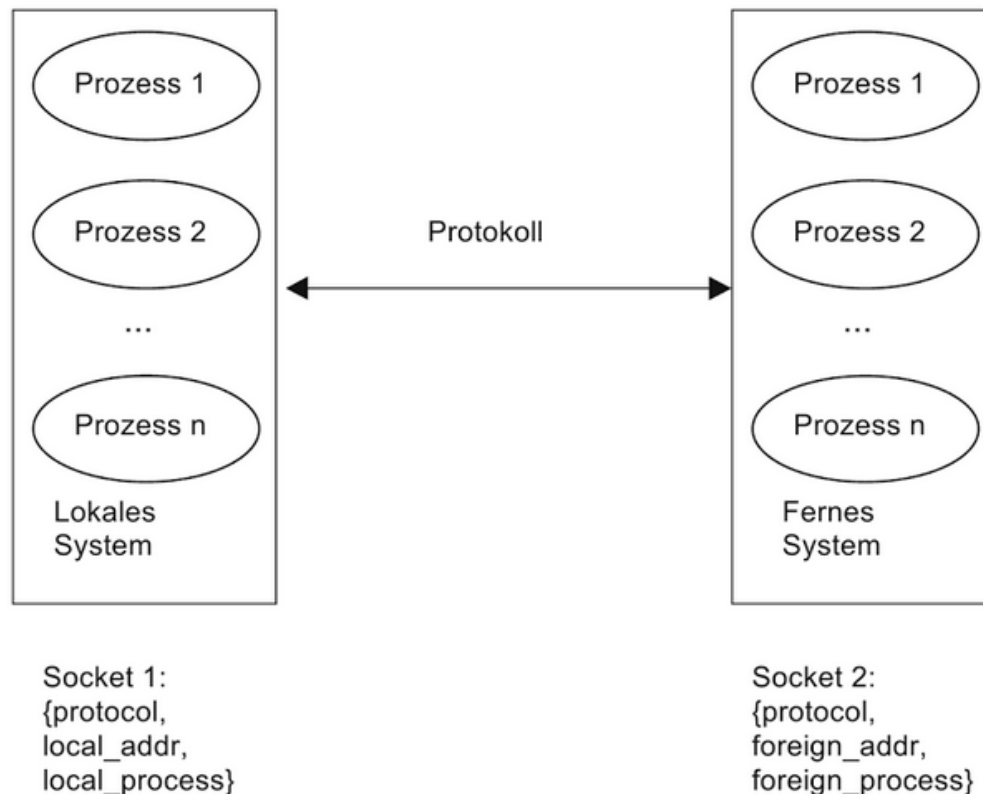
Nach dieser Lehrveranstaltung kennen Studierende idealerweise:

- Aspekte und Eigenschaften der Interprozesskommunikation
 - Bedeutung, Eigenschaften und Charakteristika von Sockets als logische Assoziation (Endpunkt für die Kommunikation) zwischen Prozessen
 - Ports als Prozessen zugeordnete und an Sockets gebundene Nachrichtenziele innerhalb eines Hosts/Computers sowie damit verbundene Adressierungsaspekte
 - Socket-Primitive bei:
 - » Verbindungsorientierter Kommunikation über TCP sowie die Besonderheit asymmetrischer Initialisierung
 - » Verbindungsloser Kommunikation über UDP
 - Abstrakte Kommunikationsmodelle bei der Interprozesskommunikation und die Kriterien Adressierung, Blockierung, Pufferung, Kommunikationsform
 - Ausgewählte Beispiele
- ✓ Studierende kennen grundlegende Aspekte der Interprozesskommunikation auf Transportebene, damit verbundene Technologien als Basis zur praktischen Implementierung einfacher verteilter Systeme und Grundlage sowie Bestandteil weiterführender Middleware-Konzepte (z.B. RPC, vgl. Lecture Notes „IPC II“)

Interprozesskommunikation [I]

= Nachrichtenübertragung zwischen einem Prozesspaar

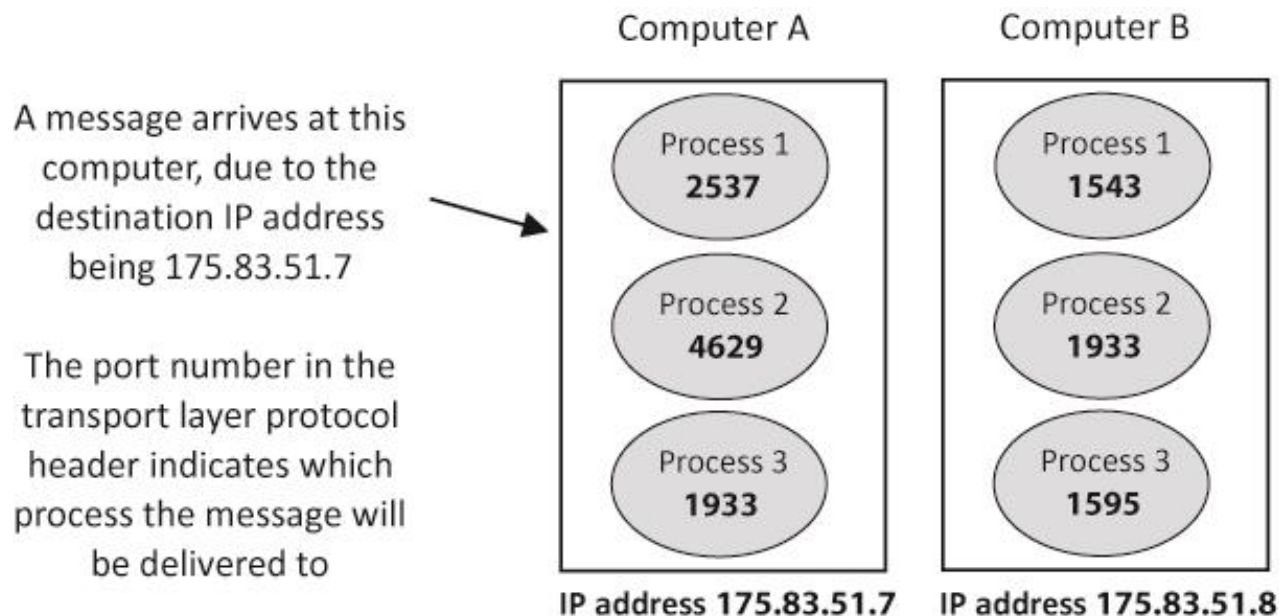
- Ziele der Interprozesskommunikation
 - » Zuverlässigkeit/Integrität: Nachrichtenübertragung ohne Beschädigungen, Duplikate
 - » Korrekte (Sende-)Reihenfolge



Interprozesskommunikation [II]

= Nachrichtenübertragung zwischen einem Prozesspaar, oder exakter formuliert: Übertragung einer Nachricht zwischen einem *Socket* in einem Prozess und einem *Socket* in einem anderen Prozess

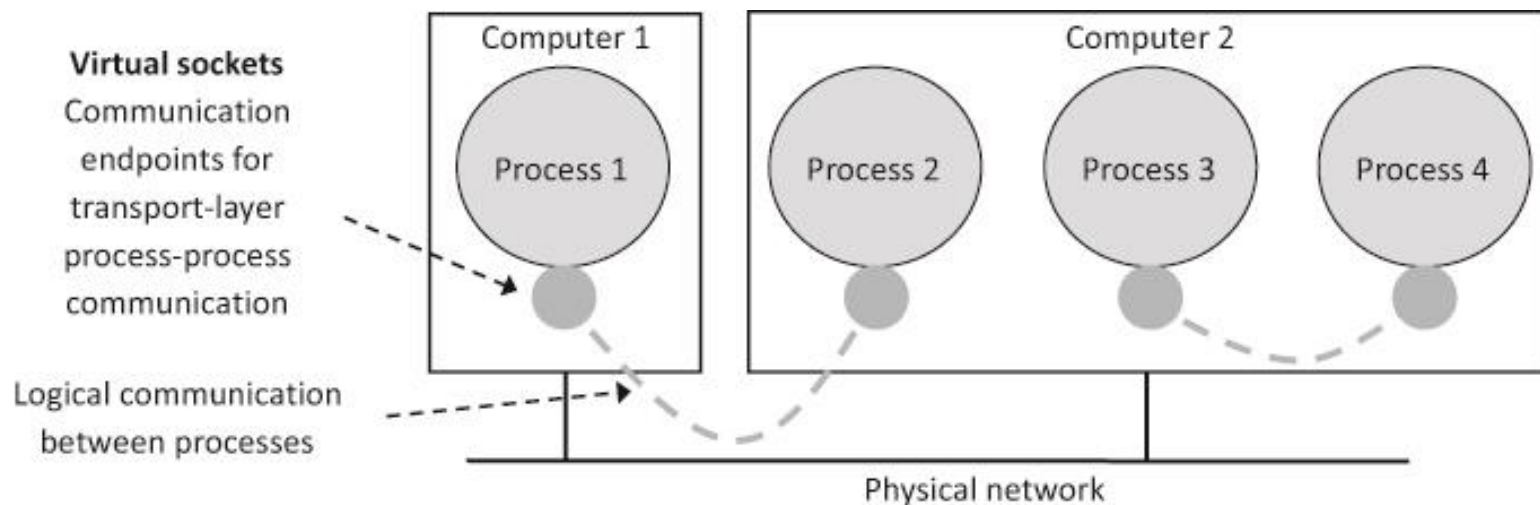
- Operationen: send, receive
- Jedem *Socket* ist ein Protokoll zugeordnet (z.B. UDP, TCP)
- Jeder Prozess, der Nachrichten senden oder empfangen will, muss zunächst ein *Socket* erzeugen, das an eine IP-Adresse und einen lokalen *Port* gebunden ist (Sockets der jeweiligen Prozesse müssen an einen *Port* gebunden sein)



„Socket“: Begriffseingrenzung

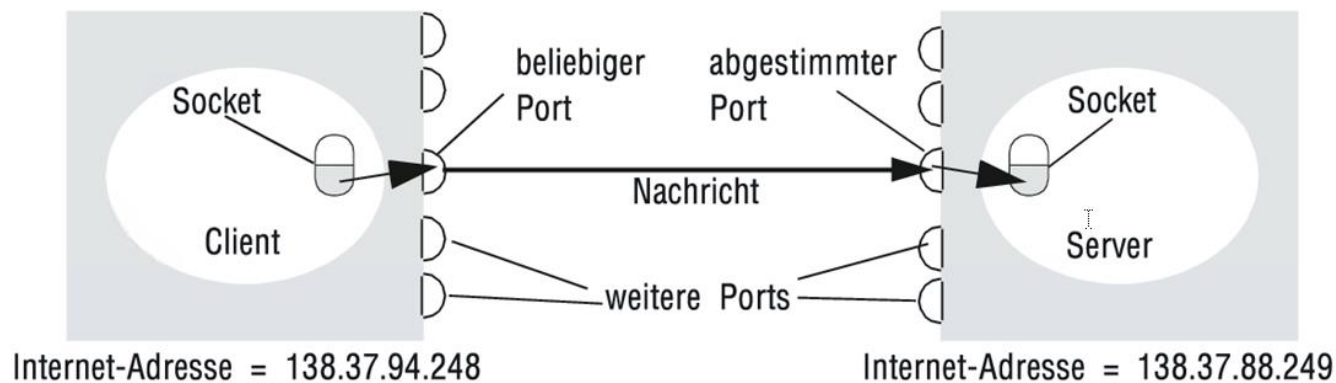
Beschreibungen / Definitionen aus der Fachliteratur (Auszug):

Definition	Quelle(n)
„Begrifflich ist ein Socket ein Kommunikationsendpunkt, in dem eine Anwendung Daten schreiben kann, die dann über das zu Grunde liegende Netzwerk gesendet werden, und aus dem Daten gelesen werden können.“	Tanenbaum & van Steen [2008: 166]
„Bei den Sockets (engl. Für Steckdose, Buchse) handelt es sich um die klassische Inkarnation des Client-Server-Modells. Mit einem Socket kann man die Kommunikation zwischen Client und Server realisieren, indem ein definierter Kommunikationskanal zwischen beiden festgelegt und etabliert wird. In der Praxis werden Sockets meist nicht als alleiniger Mechanismus verwendet, sondern dienen als Basis für komplexere Kommunikationsverfahren wie z.B. RMI oder CORBA.“	Dunkel et al. [2008:27]
„Sockets ermöglichen die Kommunikation zwischen Prozessen, die auf einem System oder zwei getrennten Systemen ablaufen können. Sockets stellen eine Kommunikationsverbindung zwischen Prozessen her.“	Bengel [2004: 86f.]
„A socket is a structure in memory that represents the endpoint for communication (i.e. sockets are the means by which processes are identified by the communication system). [...] The socket Application Programming Interface (API) is a set of library routines (called socket primitives) that a programmer uses to configure and use the TCP and UDP communication protocols from within an application. [...] the socket is effectively the process' interface to the communication system [...] regardless of whether its communication partner is local or remote “	Anthony [2016:151f.]



Sockets [I]: Eigenschaften und Charakteristika

- Logische Assoziation / Endpunkt für die Kommunikation zwischen Prozessen / Anwendungen („Steckdose“)
- Schnittstelle zwischen Transportschicht und Anwendungsschicht
- Standard einer Programmierschnittstelle für die direkte Nutzung von TCP/IP und UDP/IP
- Prozesse können zum Senden und Empfangen von Nachrichten dasselbe Socket verwenden (bidirektionale Kommunikation)
- Prozesse dürfen keine *Ports* (vgl. ff.) mit anderen Prozessen auf demselben Computer gleichzeitig verwenden, d.h. Sockets sind jeweils an einen Port gebunden

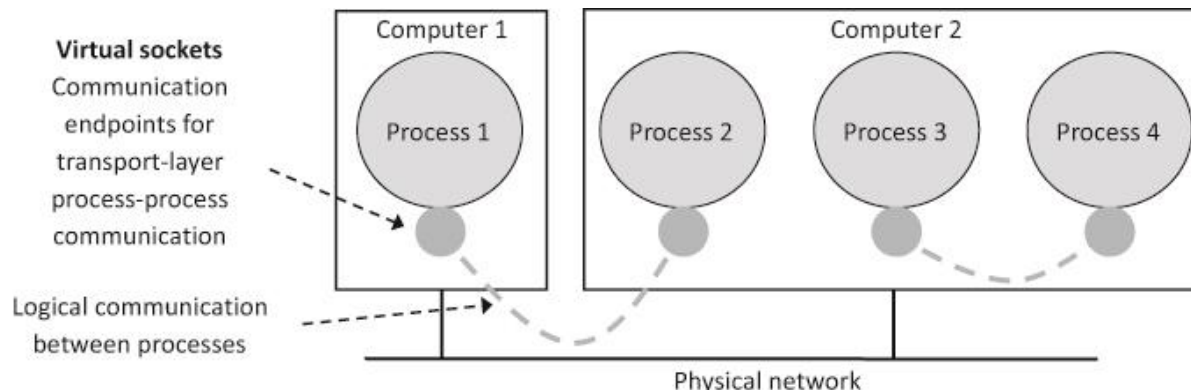


Quelle: Coulouris et al. [2002, 161]

Sockets [II]: Eigenschaften und Charakteristika

vgl. Weber [1998:53f.]

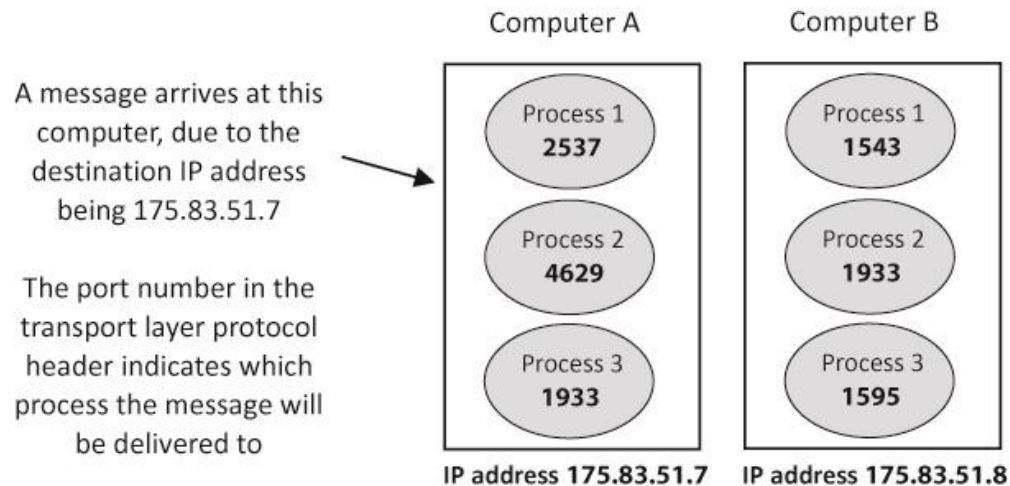
- Jede Kommunikation eines Prozesses mit einem anderen (entfernten) Prozess verwendet Sockets als Zugangspunkt zum Transportsystem
- Ein Socket kann mit einem oder mehreren Prozessen verbunden sein. Sockets können also gemeinsam verwendet werden.
- Sockets sind wie Variablen in einem Programm zu verwenden. Eine Namensgebung ist deshalb möglich.
- Je nach Art der gewünschten Kommunikation muss eine entsprechende Typangabe erfolgen:
 - » Für verbindungslose Kommunikation (UDP): Typ „Datagram“
 - » Für verbindungsorientierte Kommunikation (TCP): Typ „Stream“
- Sockets können aktiv oder passiv sein. Clients verwenden aktive, Server passive Sockets.



Quelle: Anthony [2016:151]

Ports [I]

- = über Software definierte Zielpunkte für die Kommunikation innerhalb eines Hosts
- Ports sind Prozessen zugeordnet und ermöglichen diesen, in Paaren zu kommunizieren
- Ports sind unabhängig von der Implementierung dahinter liegender Prozesse
 - » Ports unterstützen selektiven Nachrichtenempfang (Prozesse können auf mehreren Ports senden oder empfangen)
- Portadressierung
 - » Adressierung durch Transportschicht via Transportadresse
 - » Paketauslieferung durch Vermittlungsschicht
- Clients müssen Adresse des Servers (Informationspaar: IP-Adresse und Portnummer) kennen



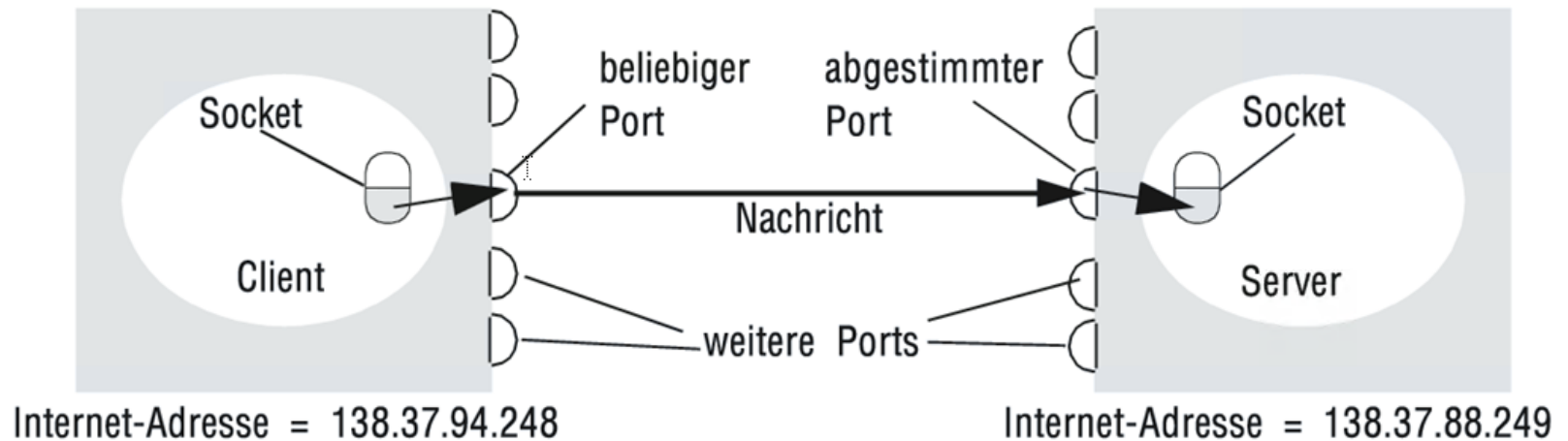
Quelle: Anthony [2016:149]

Ports [II]

- (Lokaler) Port =
 - Nachrichtenziel innerhalb eines Hosts/Computers
 - Ganzzahliger Wert; vgl. Portnummern ff.)
- Spezifikation von Nachrichtenzielen gemäß Schema wie folgt:

„IP-Adresse“ + „:“ + „Portnummer des lokalen Ports“

- Prozesse können mehrere Ports verwenden, um Nachrichten entgegenzunehmen
- Problem: Dienste mit feststehender Internetadresse können von Clients nur genutzt werden, solange der Dienst auf dem selben Computer bleibt.



Quelle: Coulouris et al. [2002:161]

Ports [III]: Portnummern

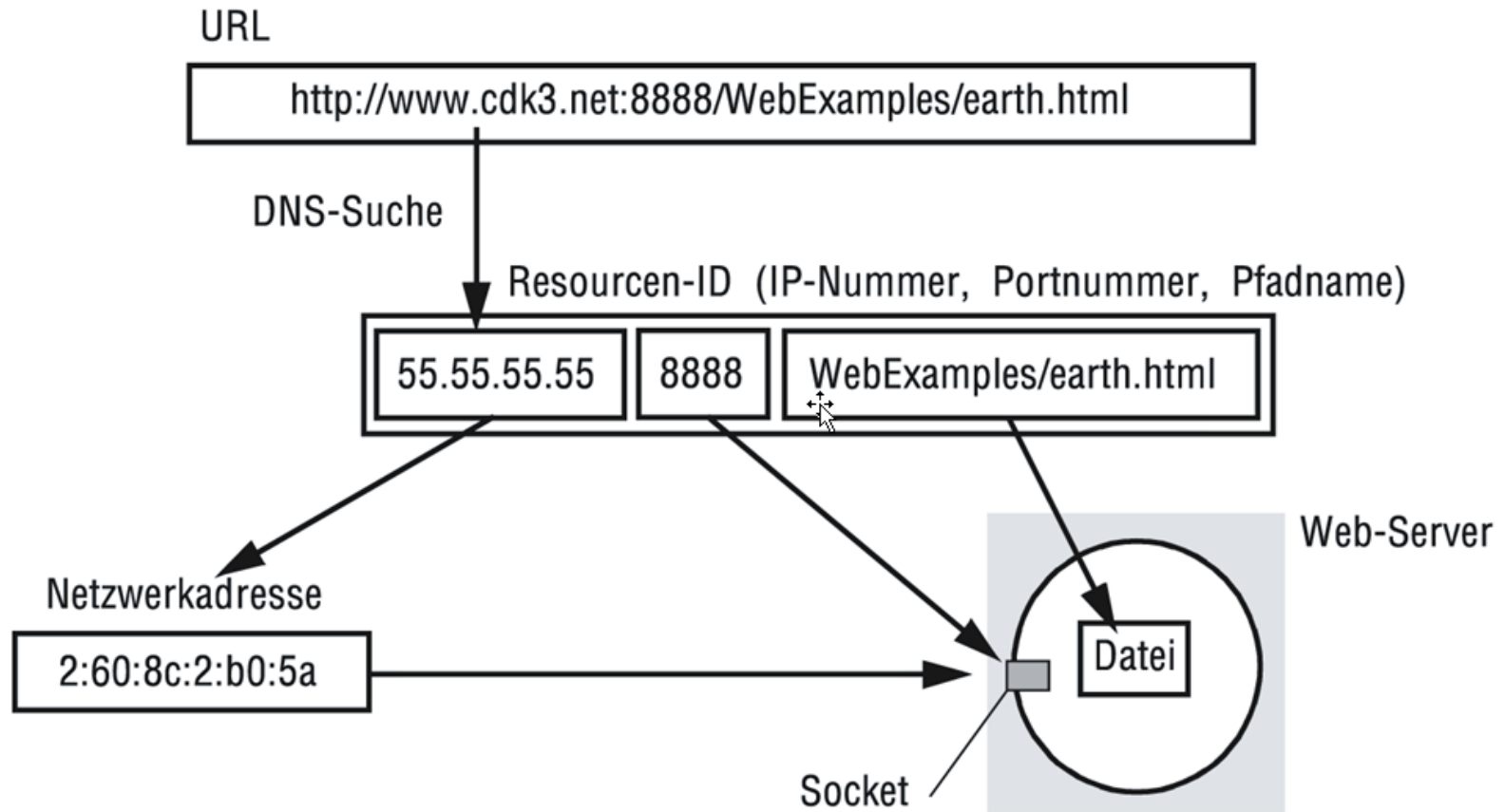
vgl.: Internet Assigned Numbers Authority (IANA), RFC 6335 <https://tools.ietf.org/html/rfc6335> [letzter Zugriff: 22.11.18]

- Regelung/Registrierung durch Internet Assigned Numbers Authority (IANA))
- Ergebnis aus der Art des in Anspruch zu nehmenden Dienstes
- Kategorisierung / Unterscheidung nach Gebrauch: speziell (reserviert) vs. Allgemein
 - » 0-1023: system ports / well-known port numbers
 - 1-255: TCP/IP-Anwendungen
 - 256-1023: UNIX-Anwendungen
 - » 1024-49151: user Ports / registered ports
 - » 49152-65535: dynamic / private Ports mit vergänglicher Nutzung (von kurzer Dauer, „ephemeral“)

Port	Service	Port	Service
5	Remote Job Entry	118	Structured Query Language (SQL) services
7	Echo	123	Network Time Protocol (NTP)
20	File Transfer Protocol (FTP) data	264	Border Gateway Multicast Protocol (BGMP), routing
21	File Transfer Protocol (FTP) command	389	Lightweight Directory Access Protocol (LDAP)
23	Telnet	513	Rlogin (remote access)
25	Simple Mail Transfer Protocol (SMTP)	520	Routing Information Protocol (RIP)
53	Domain Name System (DNS)	521	Routing Information Protocol Next Generation (RIPng)
69	Trivial File Transfer Protocol (TFTP)	530	Remote Procedure Call (RPC)
80	HyperText Transfer Protocol (HTTP)	546	Dynamic Host Configuration Protocol v6 Client (DHCPv6)
88	Kerberos (authentication system)	547	Dynamic Host Configuration Protocol v6 Server (DHCPv6)
110	Post Office Protocol version 3 (POP3)	944	Network File Service (NFS)
111	ONC Remote Procedure Call (SUN RPC)	976	Network File Service (NFS) over IPv6
115	Simple File Transfer Protocol (SFTP)		

Wir erinnern uns ... [I]

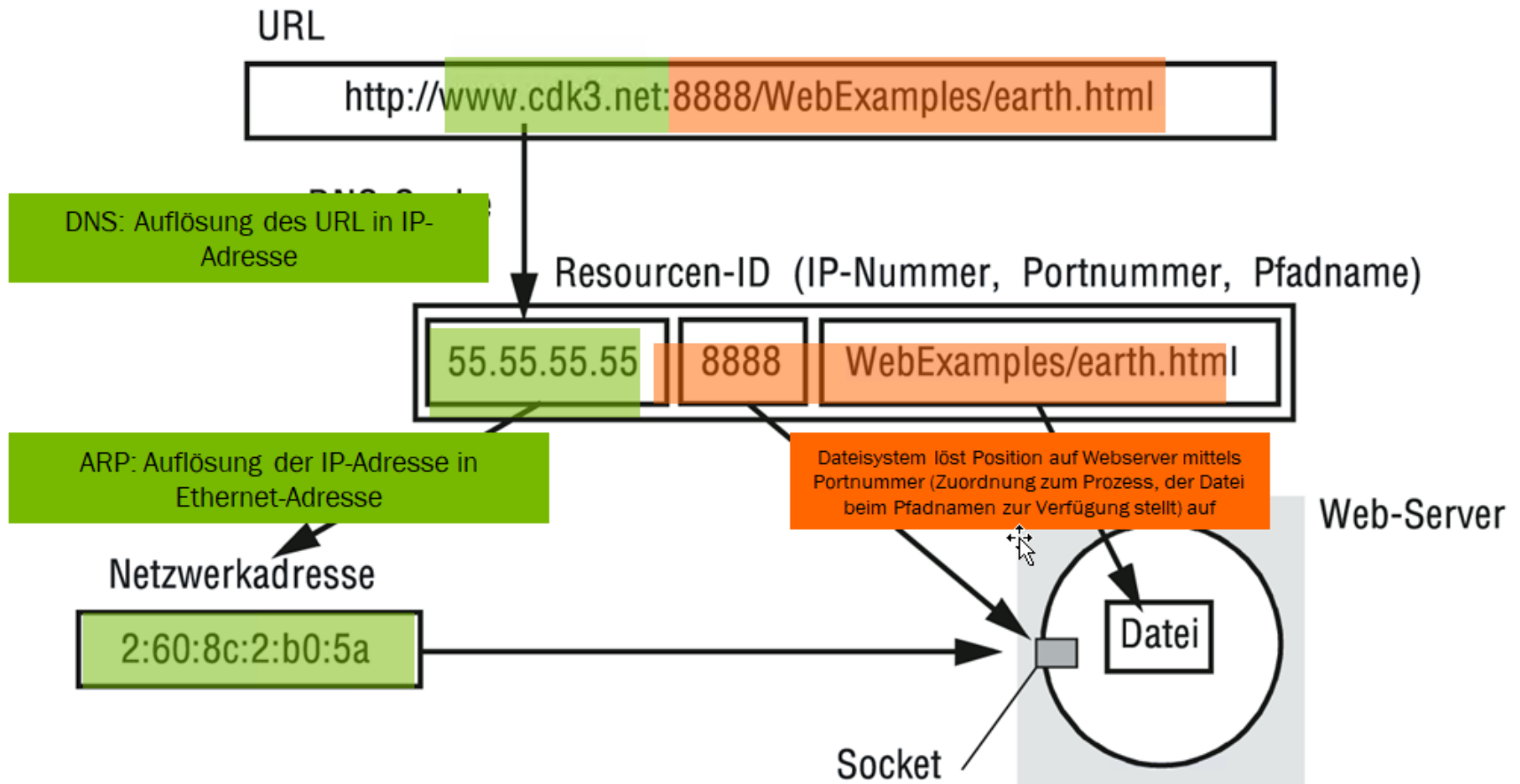
... Angaben von Ressourcen per URL enthalten Portnummern



Quelle: Coulouris et al. [2002:415]

Wir erinnern uns ... [II]

... Angaben von Ressourcen per URL enthalten Portnummern

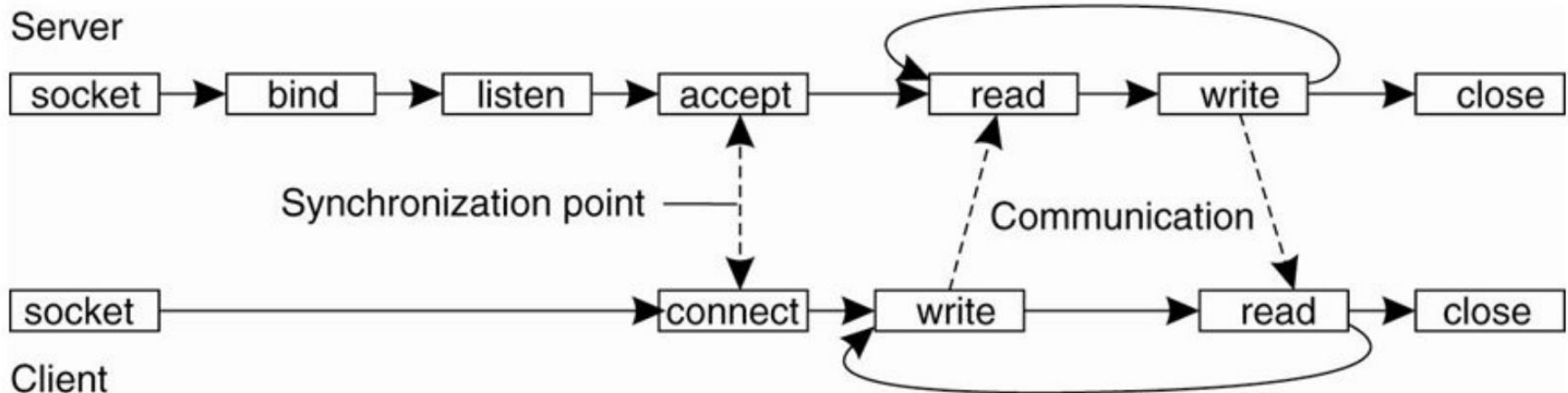


Quelle: Coulouris et al. [2002:415]

Socket-Primitives: TCP/IP [I]

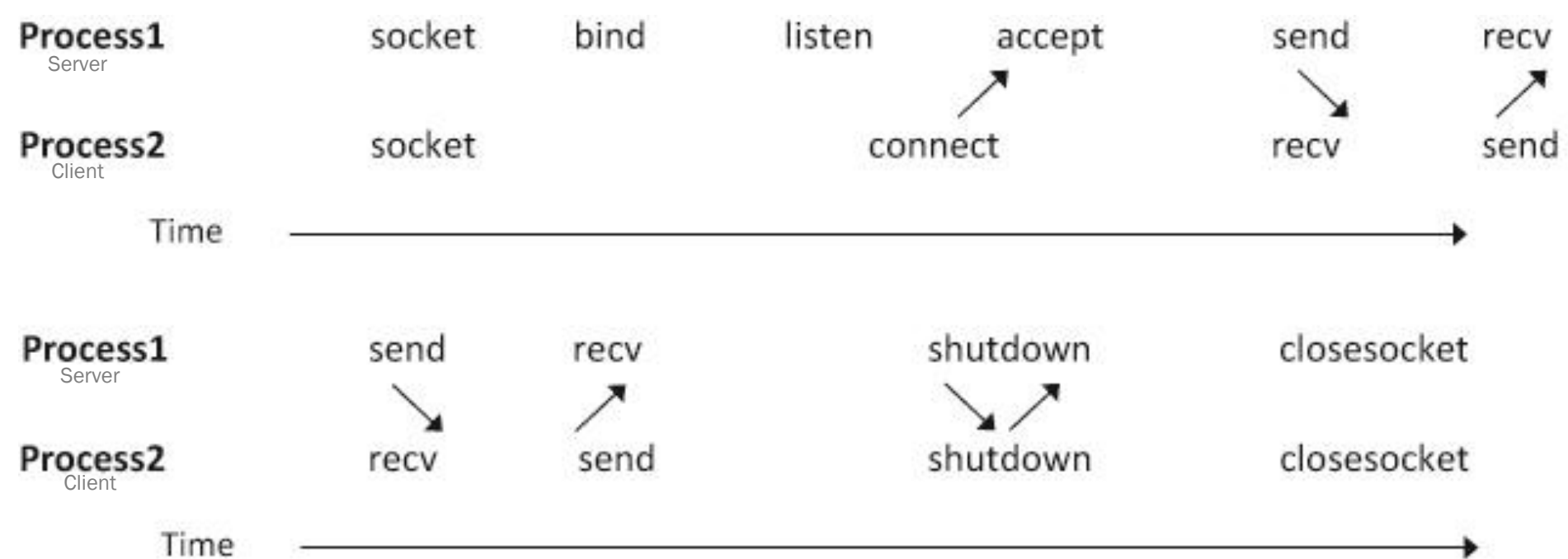
Primitive		Bedeutung
Socket		Einen neuen Kommunikationsendpunkt erstellen
Bind		Eine lokale Adresse einem Socket zuordnen (präziser: „map a process to a port“ [Anthony 2016:153])
Listen		Die Bereitschaft zum Empfangen von Verbindungen anzeigen (nicht blockierender Aufruf)
Accept		Den Aufrufer blockieren, bis eine Verbindungsanfrage eingeht
Connect		Aktiver Versuch, eine Verbindung aufzubauen
Send		Daten von einem Prozess über die Verbindung zu einem anderen Prozess senden
Receive		Daten eines Prozesses über die Verbindung empfangen
Close	Shutdown	Die Verbindung freigeben
	Closesocket	Socket schließen

Quelle: i.A. an Tanenbaum & van Steen. [2008:167] & Anthony [2016:152ff.]



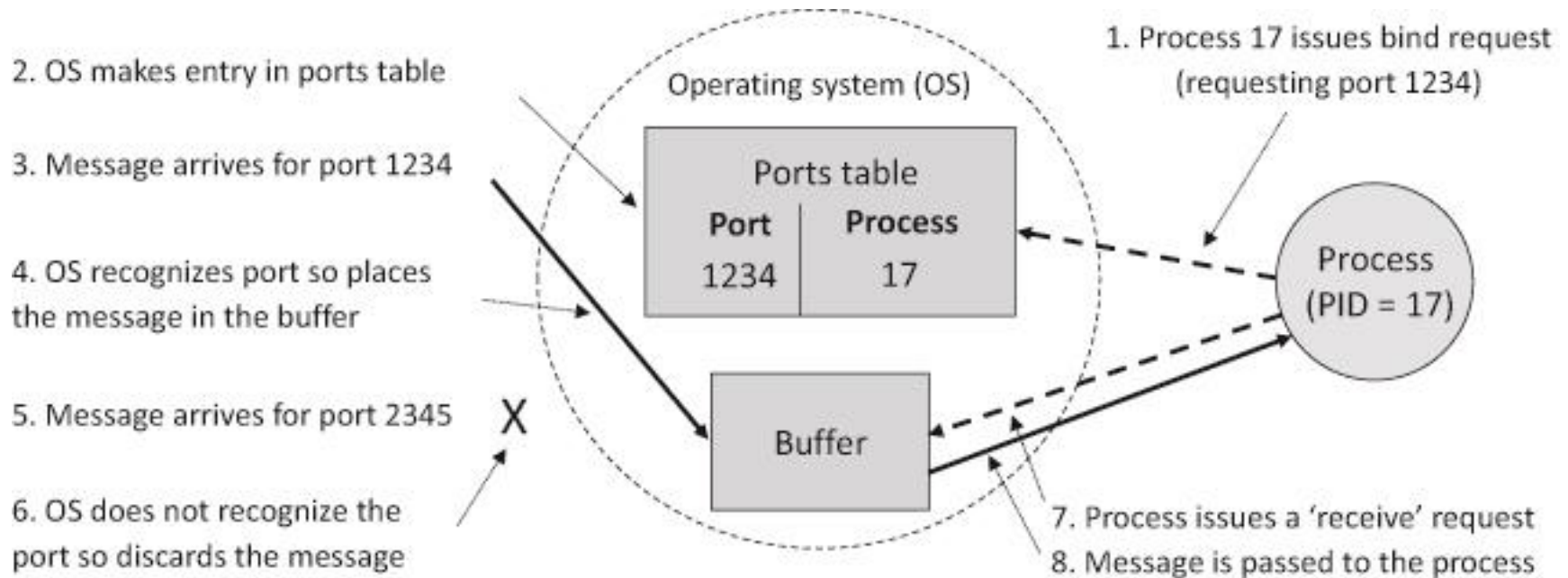
Quelle: Tanenbaum & van Steen. [2008:168]

Socket-Primitives: TCP/IP [II]



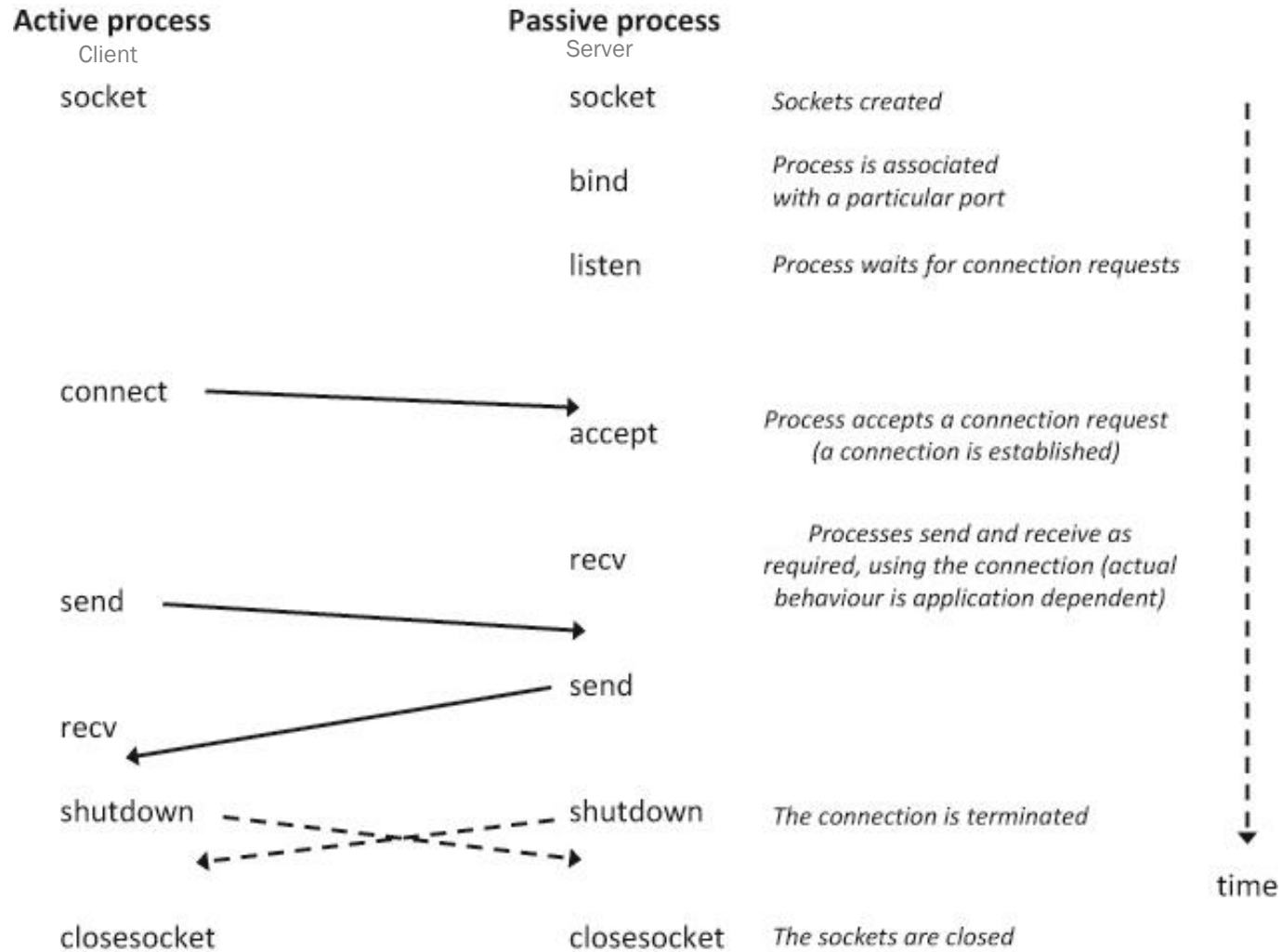
Quelle: Anthony [2016:156]

Socket-Primitives: Binding (Mapping: Prozess zu Port)



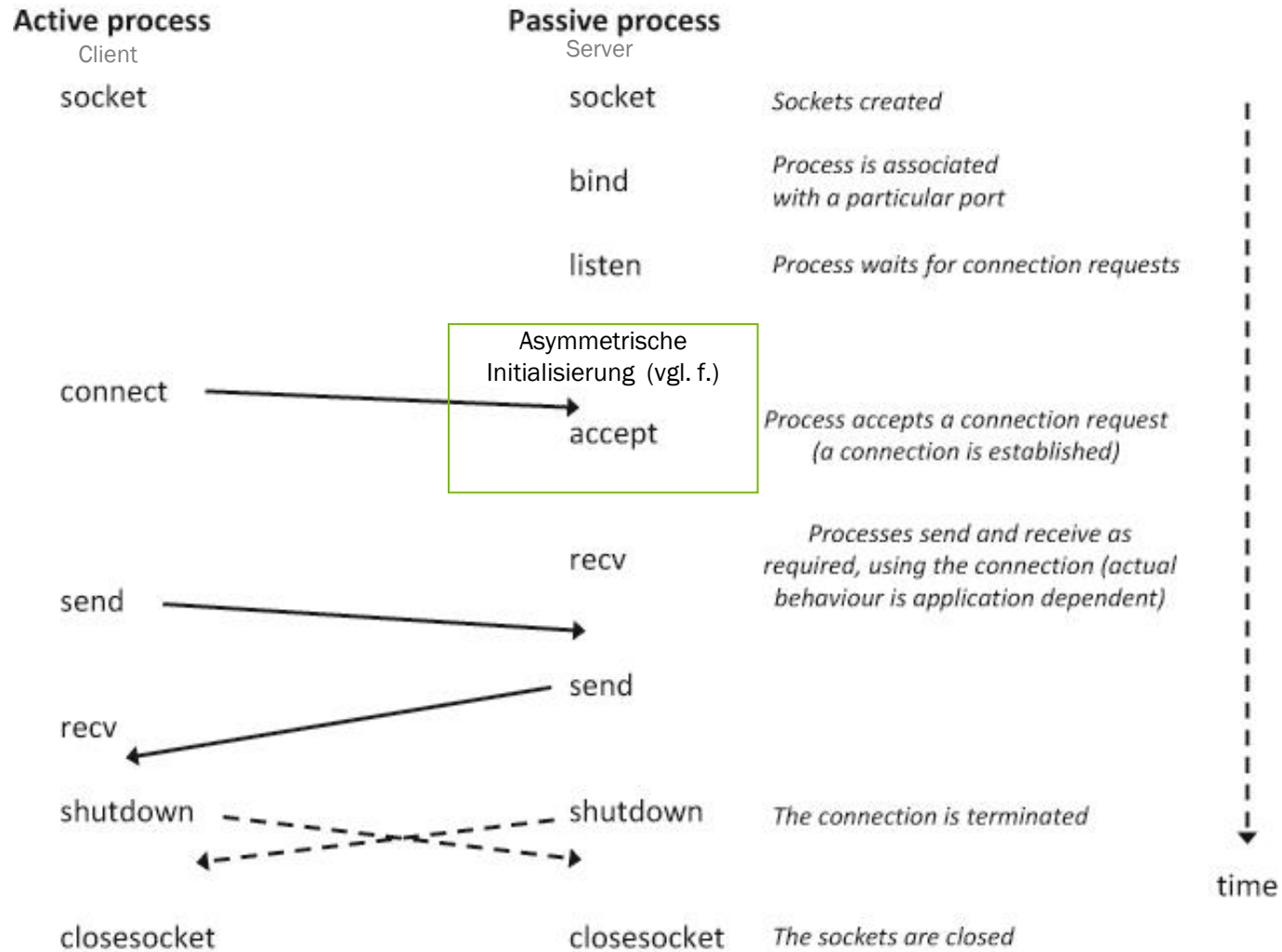
Quelle: Anthony [2016:159]

Socket-Primitives: TCP/IP [III]



Quelle: Anthony [2016:164]

Socket-Primitives: TCP/IP [IV]

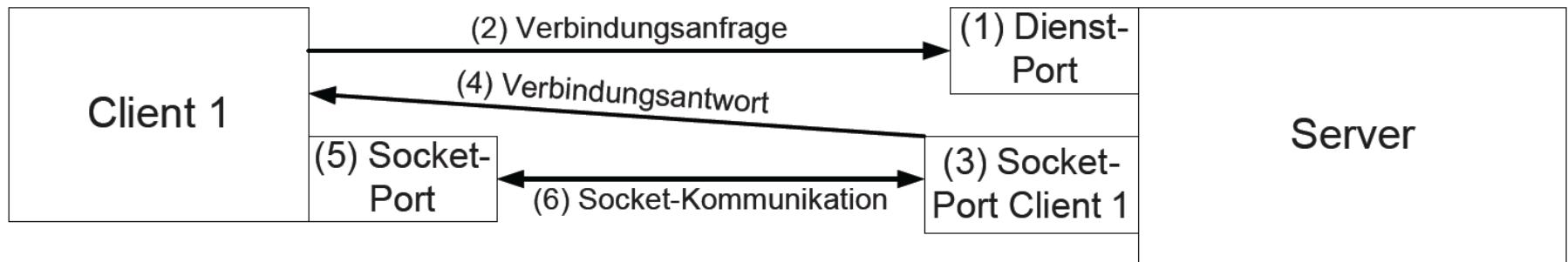


Quelle: Anthony [2016:164]

„accept“-Primitive: Asymmetrische Initialisierung [I]

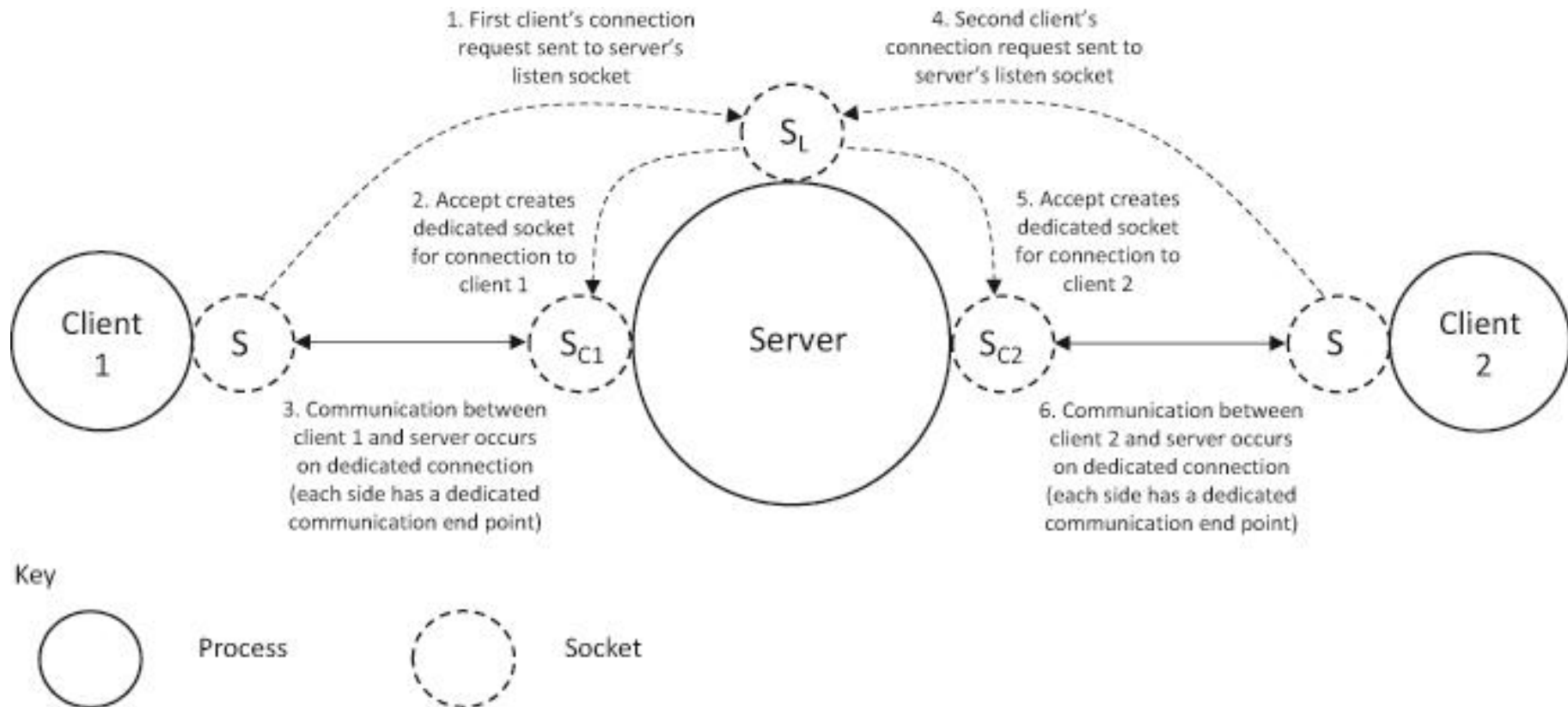
vgl. Tanenbaum & van Steen [2008:167f.]

- Aufruf von `accept` blockiert den Aufrufer, bis eine neue Verbindungsanfrage eingeht. Ist das der Fall, erstellt das lokale OS einen neuen Socket mit den gleichen Eigenschaften, die der ursprüngliche hatte, und gibt ihn an Aufrufer zurück.
- Tatsächliche Kommunikation erfolgt über „neuen“ Socket
- Server kann erneut Anfragen auf „altem“ Socket bearbeiten



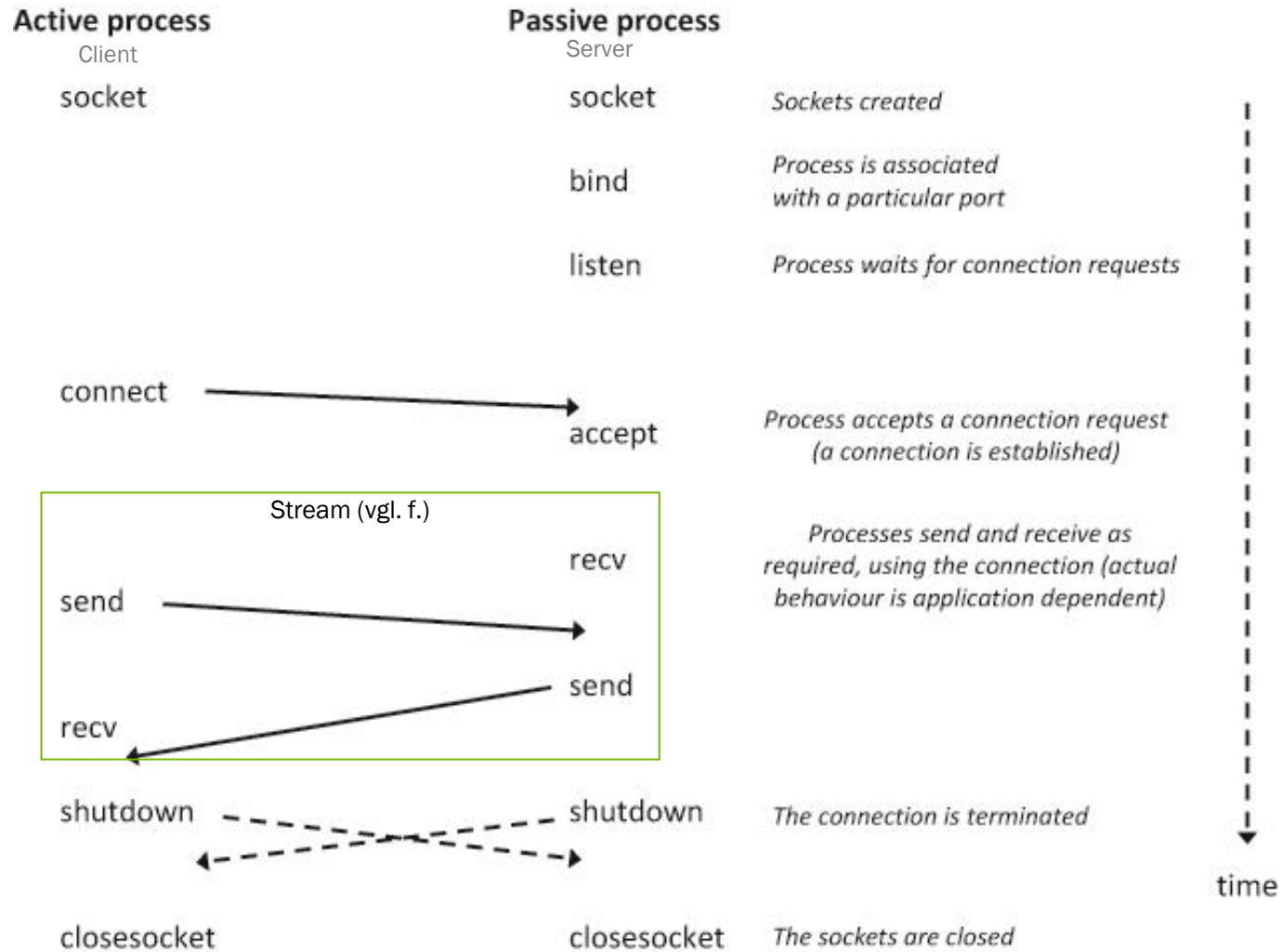
Quelle: Dunkel et al. [2008:28]

„accept“-Primitive: Asymmetrische Initialisierung [II]



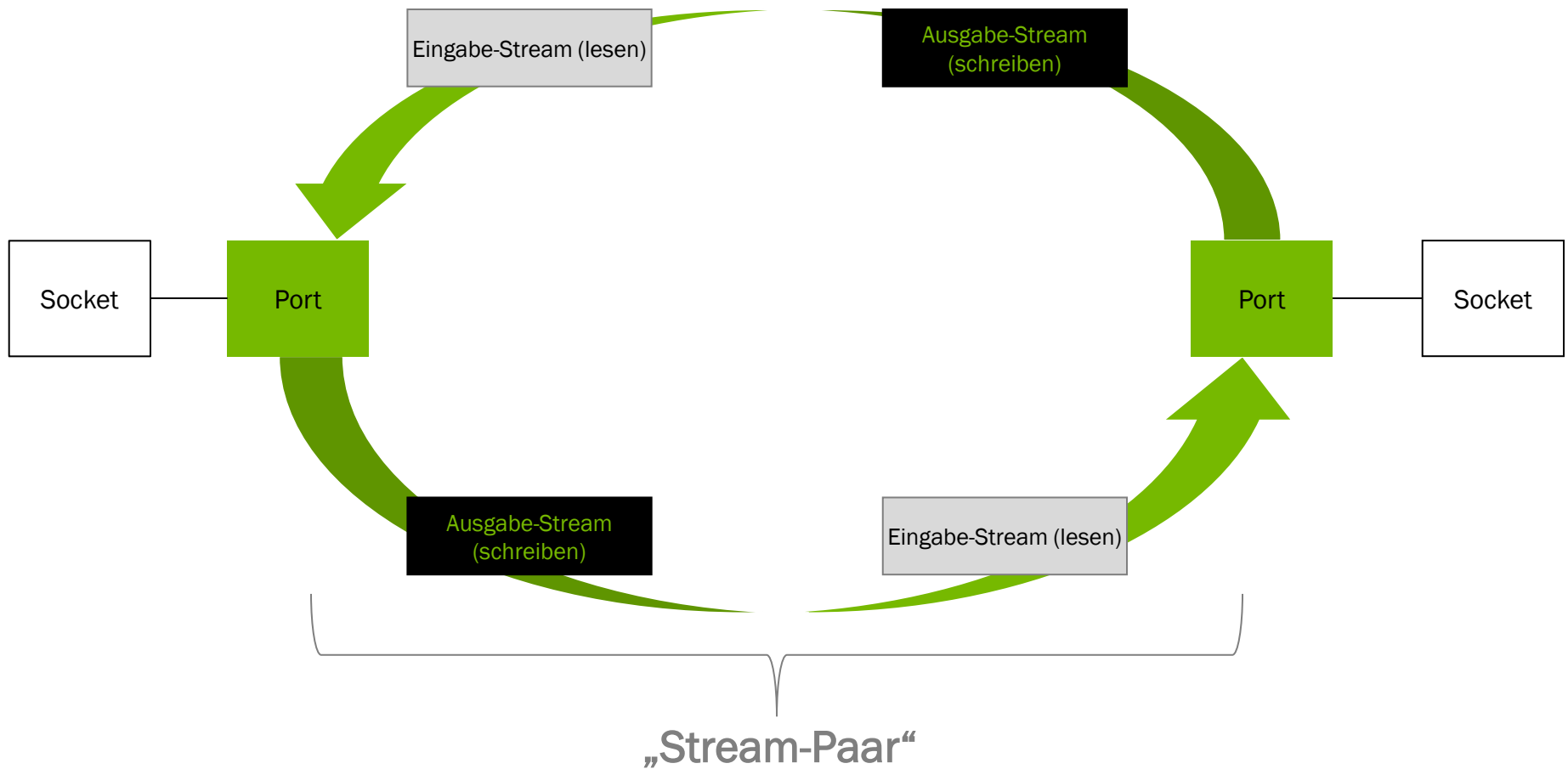
Quelle: Anthony [2016:139]

Socket-Primitives: TCP/IP [IV]

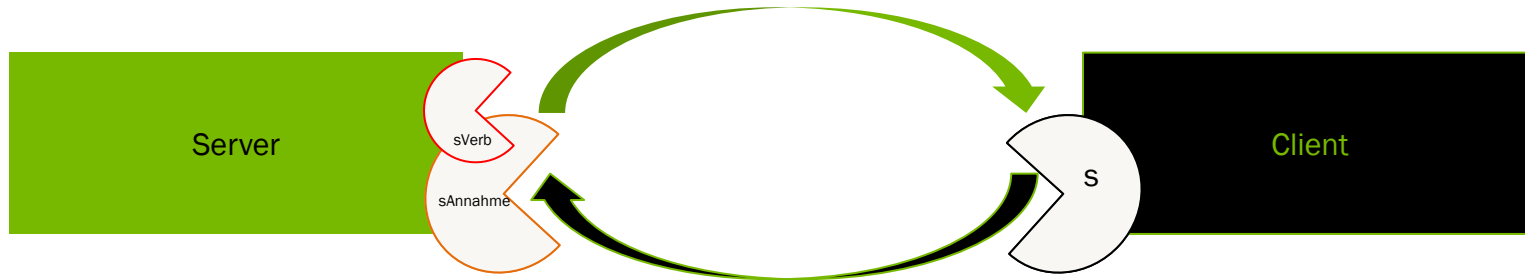


Quelle: Anthony [2016:164]

Socket-Primitives: TCP/IP [V]: TCP-Stream Kommunikation



Beispiel: Pseudo-Code von C/S-Programmen zur trivialen Kommunikation über Socket-Schnittstellen (TCP)



Erzeuge einen TCP-Socket sAnnahme mit einer spezifischen Portnummer;
wiederhole immer wieder:

```
{
  warte auf Verbindungsaufbau an sAnnahme,
  dabei wird ein neuer Socket sVerb erzeugt;
  wiederhole solange die Verbindung besteht:
  {
    analysiere den Auftrag;
    führe den Auftrag aus;
    sende über sVerb das Ergebnis der
    Ausführung zurück;
    /* dabei ist die Angabe von IP-Adresse und
    Portnummer nicht möglich / nicht nötig */
  }
  schließe sVerb;
}
```

*Initialisierung/Bindung
der Sockets an Ports*

Erzeuge einen TCP-Socket s;
/* Socket besitzt irgendeine, im Moment nicht
benutzte Portnummer */
Baue über s eine Verbindung zu einer vorgegebenen
IP-Adresse und Portnummer auf;
/* dabei wird der Client so lange blockiert, bis
der Server die Verbindung annimmt */
wiederhole so oft wie nötig:
{
 sende über s einen Auftrag;
 /* dabei ist die Angabe von IP-Adresse und
 Portnummer nicht möglich / nicht nötig */
 warte auf das Ergebnis an s;
 analysiere das Ergebnis;
}
schließe die Verbindung über s;

- ✓ Asymmetrische Initialisierung: Unterscheidung zwischen Socket für Empfang („sAnnahme“) und Socket für Ergebnisversand („sVerb“) beim Server

Exkurs: Java-Klassen zur (Stream-)Kommunikation über TCP

„Socket“

- Vorgesehen für Client/Server
 - » Verwendet Konstruktor zur Erzeugung eines Sockets unter Angabe von
 - DNS-Host-Name
 - Port eines Servers
 - » Verbindung mit dem angegebenen Computer und der Port-Nummer

„ServerSocket“

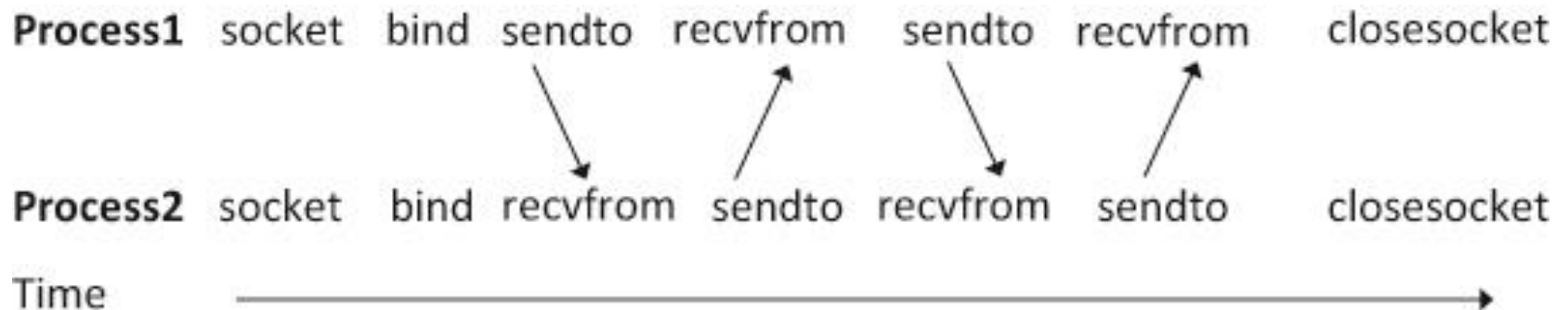
- Vorgesehen für Server
- Erzeugung eines Sockets auf dem Server-Port zur Anmeldung von Clients
 - » Warten auf connect
 - » Antwort nach Verbindung mit Client durch Methode `accept()`, die einen Socket zurückgibt

- ✓ Zweck der Klassen: Auf- und Abbau von Verbindungen
- ✓ Erweiterung durch Klassen aus dem Package `java.io`

Socket-Primitives: UDP/IP [I]

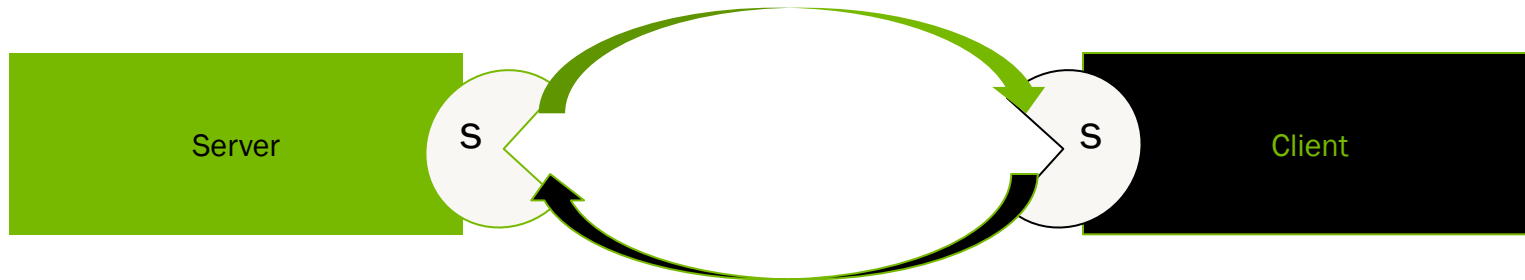
Primitive	Bedeutung
Socket	Einen neuen Kommunikationsendpunkt erstellen
Bind	Eine lokale Adresse einem Socket zuordnen (präziser: „map a process to a port“ [Anthony 2016:153])
Sendto	Daten zu einem anderen Prozess senden
Recvfrom	Abfrage von Daten aus dem (Receive-)Puffer
Closesocket	Socket schließen

Quelle: i.A. an Anthony [2016:152ff.]



Quelle: Anthony [2016:153]

Beispiel: Pseudo-Code von C/S-Programmen zur trivialen Kommunikation über Socket-Schnittstellen (UDP)



Erzeuge einen UDP-Socket *s* mit einer spezifischen Portnummer;
wiederhole immer wieder:

```
{  
    Warte auf einen Auftrag am UDP-Socket s;  
    Analysiere den Auftrag;  
    Führe den Auftrag aus;  
    Sende über s das Ergebnis der  
    Ausführung an die IP-Adresse und  
    Portnummer, von der der Auftrag  
    kam;  
}
```

*Initialisierung/Bindung
der Sockets an Ports*

Erzeuge einen UDP-Socket *s*
/* Socket besitzt irgendeine, im Moment nicht
benutzte Portnummer */;
wiederhole so oft wie nötig:

```
{  
    sende über s einen Auftrag an die  
    IP-Adresse und Portnummer des Server;  
    warte auf das Ergebnis am UDP-Socket s;  
    analysiere das Ergebnis;  
}
```

- ✓ Symmetrische Initialisierung:
Keine Unterscheidung zwischen Sender und Empfänger Socket („UDP-Socket *s*“)

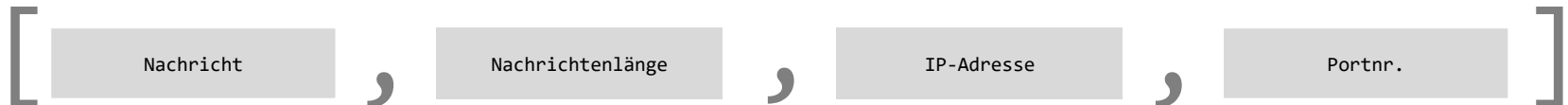
Exkurs: Java-Klassen zur (Datagramm-)Kommunikation über UDP

„DatagramSocket“

- Bereitstellung eines Konstruktors, der eine Port-Nummer als Argument entgegennimmt
- Methoden:
 - » Übertragungsmethoden
 - „send()“
 - „receive()“
 - » Time-Out Methode „SetSoTimeout“
 - » Verbindungsmethode „Connect“: Verbindung (send/receive) mit spezifischem Port, spezifischer Internetadresse

„DatagramPacket“

- Bereitstellung einer Instanz von „DatagramPacket“ bestehend aus einem Array von Bytes
 - » Senden: Array gefüllt
 - » Empfangen: Array wird vom sendenden Prozess gefüllt (via Methode „getData()“)
- Instanzen von „DatagramPacket“ können zwischen Prozessen übertragen werden, wenn sie gesendet oder empfangen werden
- Methoden:
 - » Empfang per „getData()“



Ausblick: Message Passing Interface (MPI)

- ✓ Schnittstelle zum Umgang mit fortgeschrittenen Funktionen (z.B. Formen der Pufferung, Synchronisation) in proprietären Hochgeschwindigkeits-Netzwerkverbindungen

Primitive	Bedeutung
MPI_bsend	Die ausgehende Nachricht an einen lokalen Sendepuffer anhängen
MPI_send	Senden einer Nachricht und warten, bis sie in den lokalen oder entfernten Puffer kopiert wurde
MPI_ssend	Nachricht senden und warten, bis der Empfang startet
MPI_sendrecv	Nachricht senden und auf Antwort warten
MPI_issend	Referenz an die ausgehende Nachricht übergeben und weitermachen
MPI_issend	Referenz an die ausgehende Nachricht übergeben und warten, bis der Empfang startet
MPI_recv	Nachricht empfangen; blockieren, wenn es keine gibt
MPI_irecv	Überprüfen, ob es eine eingehende Nachricht gibt, aber nicht blockieren

Quelle: i.A. an Tanenbaum & van Steen. [2008:167] & Anthony [2016:152ff.]

Kommunikationsmodelle

vgl. Weber [1998:61]

- Ausgangspunkt: Prozesse, die untereinander Informationen austauschen.
- Voraussetzung: Jede Kommunikation eines Prozesses mit einem anderen (entfernten) Prozess verwendet Sockets (vgl. ff.) als Zugangspunkt zum Transportsystem [Weber 1998:53]
- Je nach Rolle / Szenario werden diese Prozesse unterschiedlich bezeichnet:
 - » Sender und Empfänger,
 - » Erzeuger und Verbraucher,
 - » Client und Server,
 - » Peers, ...
- Unabhängig von den Namensgebungen können die am Informationsaustausch beteiligten Einheiten auf unterschiedliche Art und Weise miteinander kommunizieren (vgl. ff.):

Kriterium	Optionen / Ausprägungen	
Adressierung	Direkt	Indirekt
Blockierung	Synchron	Asynchron
Pufferung	Ungepuffert	Gepuffert
Kommunikationsform	Meldungsorientiert	Auftragsorientiert

Adressierung

vgl. Weber [1998:61ff.]

Grundfrage: Wohin soll eine Nachricht im Rahmen der Kommunikation geschickt werden

- **Direkte Adressierung:** feste Ansprache eines Empfängers durch einen Sender
 - » Symmetrisch: Jeder Teilnehmer nennt seinen Gegenpart
 - Beispiel: `send (P, message); receive (Q, message)`
 - Kommunikationsverbindung zwischen zwei Prozessen P und Q.
 - Identifikationen von P und Q: Rechneradresse + Portnummer.
 - » Asymmetrisch:
 - Sender benennt Empfänger in Senderroutine und ergänzt seine Identifikation.
 - Empfänger empfängt die Client-ID (ID) in der Nachricht (so könnte beispielsweise ein Server mehrere Clients bedienen)
 - Beispiel: `send (S, message); receive (ID, message)`
- ✓ Direkte Adressierung ist eher starr und unflexibel (Adressänderungen erfordern Programmänderungen, ggf. Rekompilierung)
- ✓ Lösungsansätze:
Schaffung von **Lokationstransparenz durch indirekte Adressierung:** Sender spricht seinen Empfänger nicht direkt, sondern über eine Empfangsstelle an, z.B. *Ports*

Blockierung [I]: Synchrone / blockierende Kommunikation

vgl. Weber [1998:61]; Coulouris et al. [2002:161f.]; Bengel [2004:86ff.]

Ausgangspunkt: Nachrichtenübertragung zwischen einem Prozesspaar wird von zwei Operationen unterstützt: Kommunikation mit *send* / *receive*

Synchron: *send* + *receive* sind blockierend

- Nach Absetzen eines *send* wird sendender Prozess blockiert bis entsprechendes *receive* abgesetzt wurde
- Nach Absetzen eines korrespondierenden *receive* wird empfangender Prozess blockiert bis eine Nachricht ankommt

Vorteile [vgl. Weber 1998:65f.] :

- Durch blockierenden Nachrichtenaustausch erfolgt automatische Synchronisation zwischen Sender und Empfänger, deshalb sind keine weiteren Mechanismen zur zeitlichen Synchronisation notwendig.
- Da auf jede Nachricht explizit gewartet wird, ist keine Pufferung von Nachrichten notwendig (vergleichsweise geringerer benötigter Speicherplatz, Verwaltungsaufwand).

Nachteile

- Blockierung schränkt Parallelität bei Sender und Empfänger ein
- Unflexibler Ablauf der Kommunikation (z.B. kann Empfänger bei blockierendem *receive* nur auf eine Art von Nachricht warten)
- Ist Sender bzw. Empfänger nicht bereit, so kann entsprechender Empfänger bzw. Sender für immer blockieren (Notwendigkeit von Timeouts)

Blockierung [II]: Asynchrone / nicht blockierende Kommunikation

vgl. Weber [1998:61]; Coulouris et al. [2002:161f.]; Bengel [2004:86ff.]

Nachrichten müssen zwischengespeichert / gepuffert werden

- Zentral auf einem Message Server
- Dezentral beim Sender / Empfänger, z.B. *send* / *receive* stellen Puffer bereit (Beispiele vgl. ff.)
 - » Nach Absetzen eines *send* wird Nachricht in Puffer gestellt und sendender Prozess ist nicht blockiert
 - » *Receive*:
 - Nicht-blockierend (erfordert Bestätigungsmeldungen, die zwischen empfangendem Prozess und bereitgestelltem Puffer ausgetauscht werden müssen)
 - Blockierend

Vorteile [vgl. Weber 1998:65f.]:

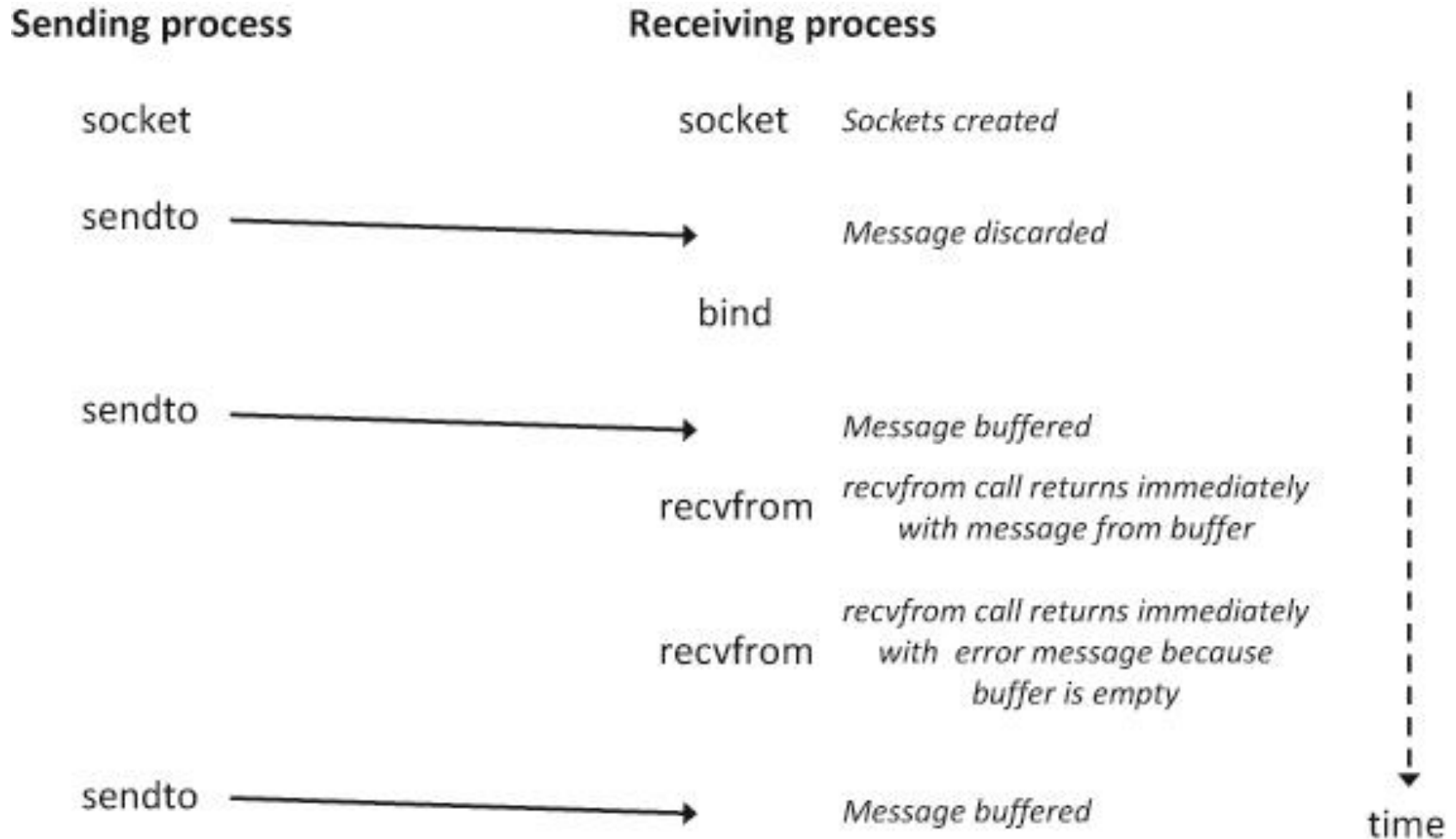
- Zeitliche Entkoppelung von Sender und Empfänger
- Ermöglichung von Parallelisierung

Nachteile:

- Wird Zuverlässigkeit verlangt, müssen Nachrichten sowohl beim Sender, als auch beim Empfänger gepuffert werden
- Sender weiß nicht, ob und wann Puffer frei sind (wenn Puffer voll sind, wird blockiert, was dem asynchronen Konzept widerspricht)

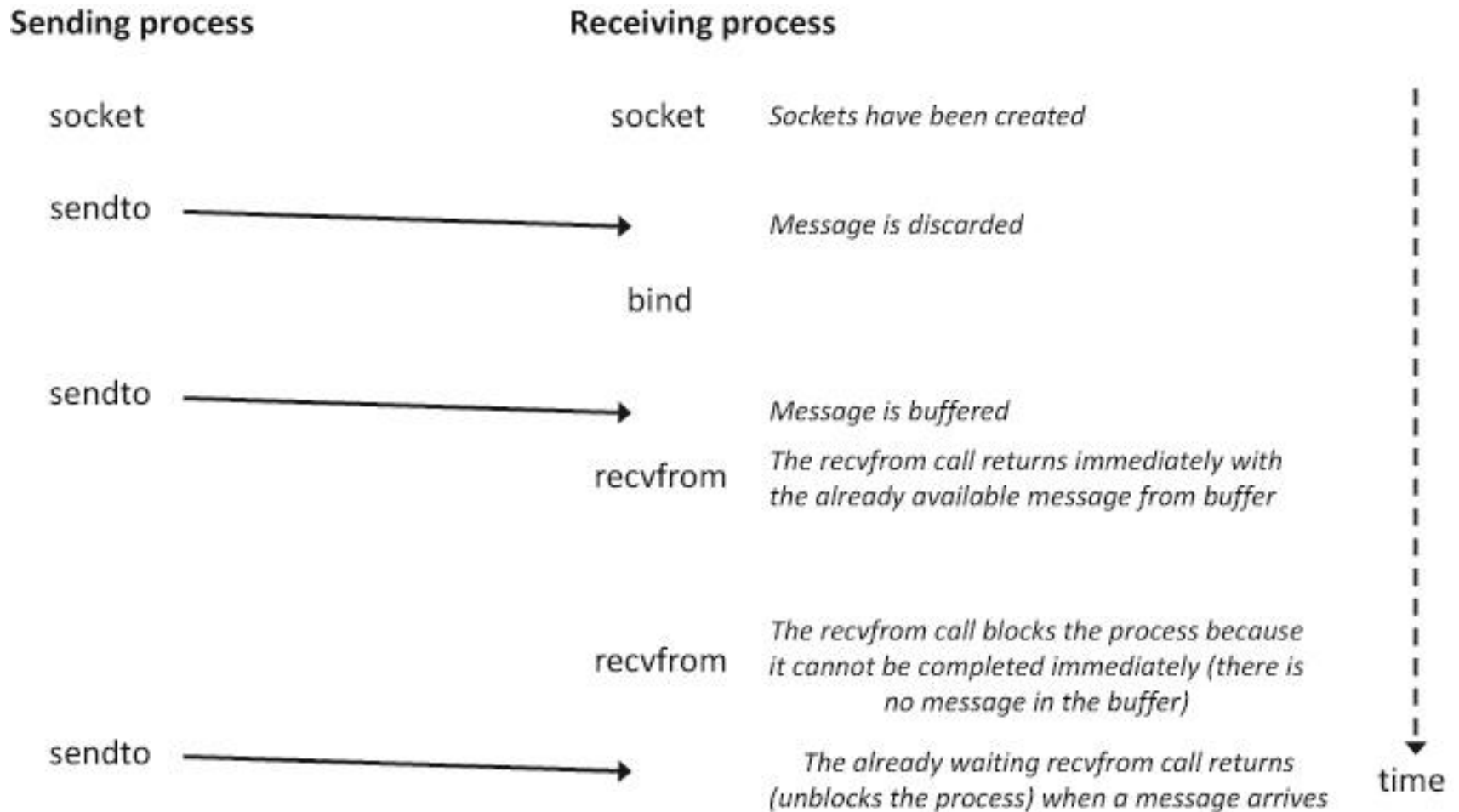
Exkurs:

UDP Socket Primitive: non-blocking socket mode



Quelle: Anthony [2016:163]

Exkurs: UDP Socket Primitive: blocking socket mode



Quelle: Anthony [2016:162]

Pufferung

vgl. Weber [1998:67f.]

Ungepufferte Kommunikation:

- Funktion receive stellt im Programm-Adressraum eine einzelne Datenstruktur (z.B. als Array) für eine Nachricht zur Verfügung.
- Bei Eingang einer Nachricht kopiert das Kommunikationssystem im Betriebssystemkern die Nachricht in diese Datenstruktur und setzt Empfangsprozess auf „ready“ (das bedeutet, dass ein Prozess bereit ist, vom Betriebssystem Prozessorleistung zugeteilt zu bekommen)
- Vorteil: Einsparung von Speicher- und Verwaltungsaufwand
- Nachteile:
 - » Wurde receive noch nicht aufgerufen, kann das Kommunikationssystem eine eingehende Nachricht nicht zustellen und muss sie verwerfen.
 - » Rufen mehrere Clients einen Server auf, so ist dieser evtl. nicht empfangsbereit, da er noch eine alte Nachricht bearbeitet (auch hier müssten ggf. weitere Nachrichten verworfen werden)

Gepufferte Kommunikation:

- Betriebssystem hält Puffer vor (evtl. in Form einer Mailbox zur Zwischenspeicherung eingehender / ausgehender Nachrichten, welche vom Empfänger / vom Netzwerk nicht abgenommen werden können)
- Falls Puffer voll sind, müssen Nachrichten dennoch verworfen werden (evtl. Notwendigkeit der Blockierung des Senders)

Kommunikationsform [I]

Klassifikation in Anlehnung an Weber [1998]

		Synchronisationsgrad	
		asynchron	synchron
Kommunikationsmuster	Meldungsorientiert	Datagramm	Rendezvous
	Auftragsorientiert ¹	Asynchroner entfernter Dienst-/Prozeduraufruf (Remote Procedure Call (RPC))	Synchroner entfernter Dienst-/Prozeduraufruf (Remote Procedure Call (RPC))

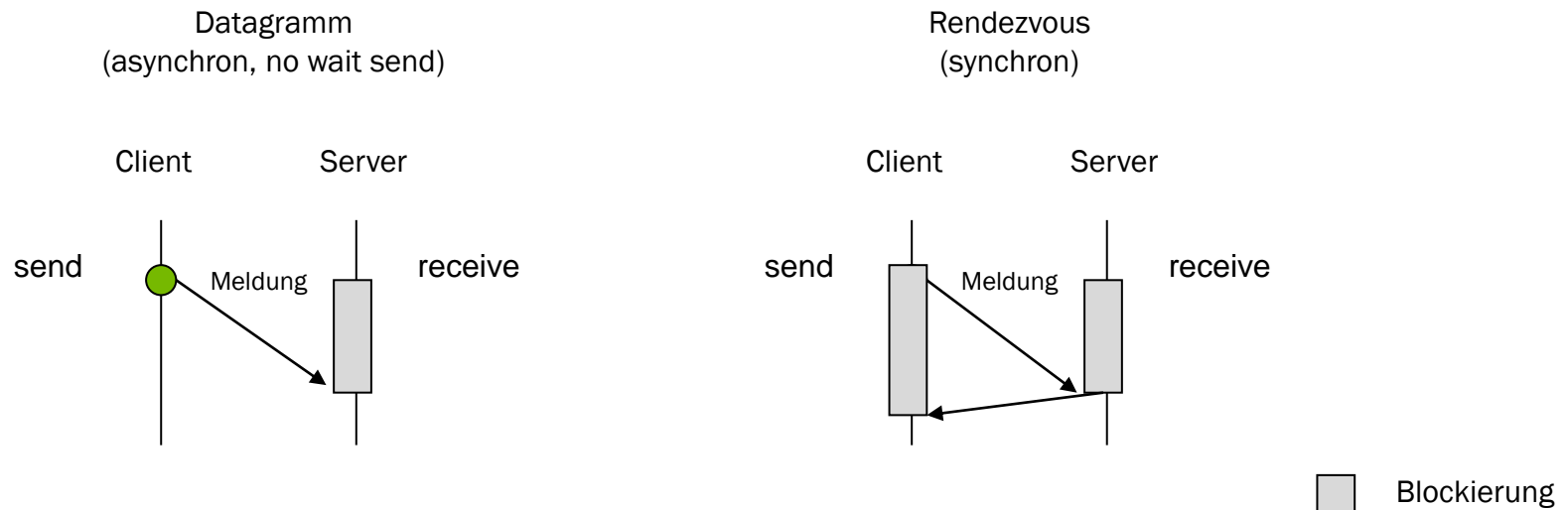
Quelle: i. A. an Weber [1998:68]

¹Vertiefung in der kommenden Lehrveranstaltung (vgl. Lecture Notes “IPC II”)

Kommunikationsform [II]: meldungsorientierte Kommunikation

vgl. Weber [1998:68f.]

- Bezeichnet Einwegnachrichten: Sender schickt Nachricht und erwartet höchstens („at-most-once“) eine Empfangsbestätigung, dass diese Nachricht empfangen wurde
- Unterscheidung nach Blockierung / zeitlichem Ablauf:

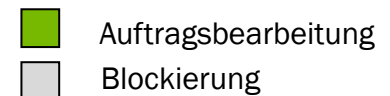
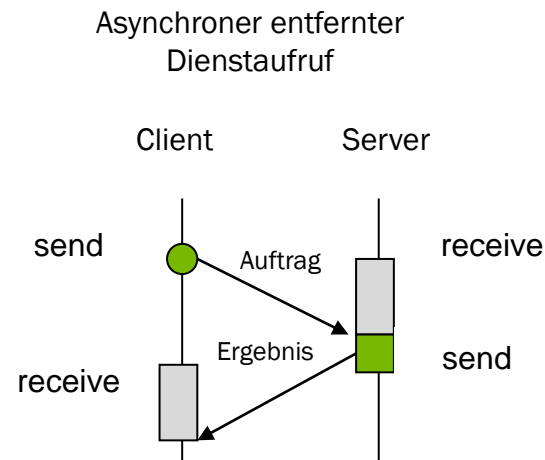
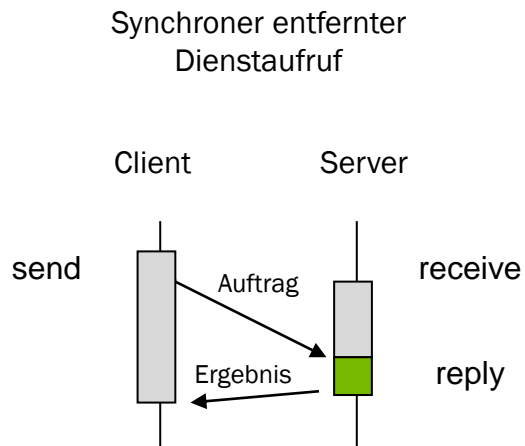


Quelle: i.A. an Weber [1998:69]

Kommunikationsform [III]: auftragsorientierte Kommunikation

vgl. Weber [1998:69f.]; Vertiefung in der kommenden Lehrveranstaltung (vgl. Lecture Notes “IPC II”)

- Bezeichnet Hin- und Rücknachrichten: Sender schickt Auftragsnachricht und erwartet daraufhin Rücksendung eines Ergebnisses
- Synchroner entfernter Dienstaufruf ist verwandt mit Rendezvous
- Asynchroner entfernter Dienstaufruf: Client und Server ähneln wechselweise dem Datagramm-Verfahren
- Unterscheidung nach Blockierung / zeitlichem Ablauf



Quelle: Weber [1998:70]

Alternative Klassifikation der Kommunikationsform nach Spector (1982)

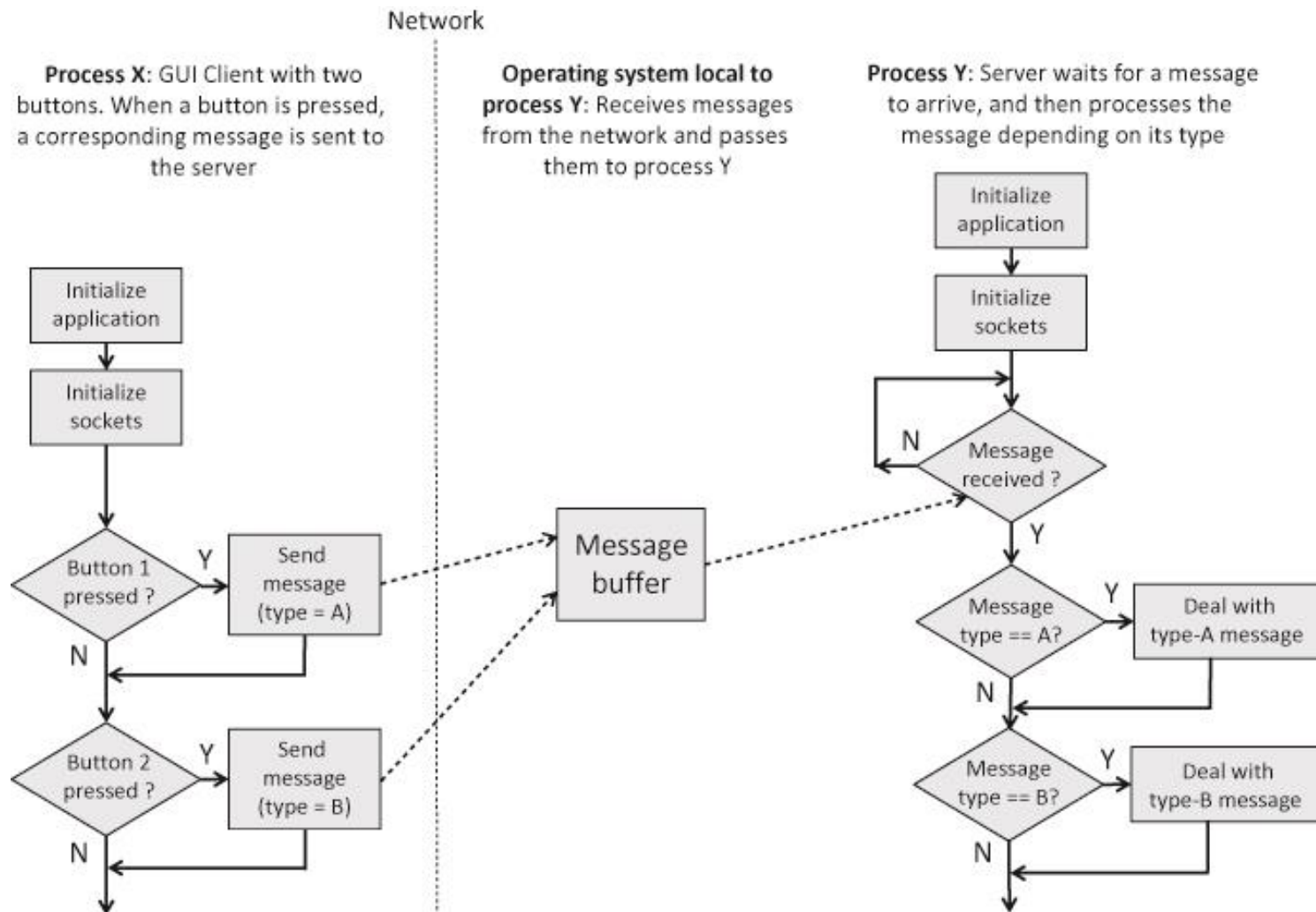
vgl. Weber [1998:70f.]

- R-Protokoll (Request): Protokoll übermittelt Nachrichten, die kein Ergebnis zurückfordern (entspricht Datagramm)
- RR-Protokoll (Request-Reply): Eine gesendete Nachricht verlangt eine Antwort (entspricht (a)synchronem entfernten Dienstaufwurf)
- RRA-Protokoll (Request-Reply-Acknowledge Reply): ähnlich RR-Protokoll, aber empfangender Prozess bestätigt empfangenes Ergebnis beim sendenden Prozess
- RR- und RRA-Protokolle erfordern Request-IDs und Sequenznummern in Nachrichten
- Übersicht Protokolle (für den RPC-Austausch, vgl. Lecture Notes „IPC II“)

Name	Nachrichten gesendet von		
	<i>Client</i>	<i>Server</i>	<i>Client</i>
R	<i>Anforderung</i>		
RR	<i>Anforderung</i>	<i>Antwort</i>	
RRA	<i>Anforderung</i>	<i>Antwort</i>	<i>Bestätigungsantwort</i>

Quelle: Coulouris et al. [2002: 183]

Exkurs: Integration von Interprozesskommunikation in Business Logik (Application level)



Quelle: Anthony [2016:171]

Herausforderungen bei der Interprozesskommunikation

- Sockets passen nicht „optimal“ zu den vorherrschenden Paradigmen des Softwareentwurfs: zu entwickelnde Systeme werden aufgeteilt in
 - » Module (Interaktion über Prozeduraufrufe) oder
 - » Objekte von Klassen (Interaktion über Methodenaufrufe)
- Limitierter „Quality of Service“:
 - » UDP ist unzuverlässig und erfordert programmatische Erweiterungen für den UDP-Server und UDP-Client zur Erhöhung der Zuverlässigkeit, z.B.:
 - Befristetes Warten auf eine Antwort
 - Wiederholtes Senden des Auftrages
 - » TCP erfordert explizite Ausprogrammierung des Verbindungsauf- und -abbaus (z.B. Parallelität mehrerer Client-Anfragen)
- Rechnerinterne Datendarstellungen in heterogenen Netzen können variieren, z.B.: Beispiel:
 - Zeichensatz: ASCII (ein Byte pro Zeichen), Unicode (zwei Byte pro Zeichen)
 - Reihenfolge von Integern (ganze Zahlen): Big-Endian (MSB zuerst), Little-Endian (MSB zuletzt)
 - Layout von Datenstrukturen
- Datenformate müssen vor und/oder nach der Übertragung gewandelt werden (vgl. Marshalling)
- ✓ Vertiefung von Techniken, Technologien und Konzepte (z.B. Middleware) auf höheren Ebenen (über der Transportebene) für Anwendungsentwickler_innen in der kommenden Lehrveranstaltung (vgl. Lecture Notes “IPC II”))

Danke. Lernziele erreicht?

Nach dieser Lehrveranstaltung kennen Studierende idealerweise:

- Aspekte und Eigenschaften der Interprozesskommunikation
 - Bedeutung, Eigenschaften und Charakteristika von Sockets als logische Assoziation (Endpunkt für die Kommunikation) zwischen Prozessen
 - Ports als Prozessen zugeordnete und an Sockets gebundene Nachrichtenziele innerhalb eines Hosts/Computers sowie damit verbundene Adressierungsaspekte
 - Socket-Primitive bei:
 - » Verbindungsorientierter Kommunikation über TCP sowie die Besonderheit asymmetrischer Initialisierung
 - » Verbindungsloser Kommunikation über UDP
 - Abstrakte Kommunikationsmodelle bei der Interprozesskommunikation und die Kriterien Adressierung, Blockierung, Pufferung, Kommunikationsform
 - Ausgewählte Beispiele
-
- ✓ Studierende kennen grundlegende Aspekte der Interprozesskommunikation auf Transportebene, damit verbundene Technologien als Basis zur praktischen Implementierung einfacher verteilter Systeme und Grundlage sowie Bestandteil weiterführender Middleware-Konzepte (z.B. RPC, vgl. Lecture Notes „IPC II“)

Quellen

Anthony, R. (2016) *Systems Programming - Designing and Developing Distributed Applications*; Amsterdam et al.: Morgan-Kaufman / Elsevier.

Bengel, G. (2004) *Grundkurs Verteilte Systeme*; Wiesbaden: Vieweg.

Coulouris, G.; Dollimore, J.; Kindberg, T. (2002) *Verteilte Systeme - Konzepte und Design*; 3., überarbeitete Auflage; München: Pearson Studium.

Dunkel, J; Eberhart, A.; Fischer, S.; Kleiner, C. ; Koschel, A. (2008) *Systemarchitekturen für Verteilte Anwendungen*; München: Hanser.

Oechsle, R. (2011) *Parallele und verteilte Anwendungen in JAVA*; 3., erweiterte Auflage; München: Carl Hanser.

Schill, A.; Springer, T. (2012) *Verteilte Systeme*; 2. Auflage; Berlin, Heidelberg: Springer Vieweg.

Tanenbaum, A.; van Steen, M. (2008) *Verteilte Systeme – Prinzipien und Paradigmen*; 2., überarbeitete Auflage; München: Pearson Studium.

Weber, M. (1998) *Verteilte Systeme*; Heidelberg, Berlin: Spektrum.

Appendix

Ergänzende Informationen (nicht klausurrelevant):

- ✓ Client/Server-Kommunikation (UDP) in JAVA
- ✓ Client/Server-Kommunikation (TCP) in JAVA

Java-API für IP-Adressen

- Ausgangspunkte: IP-Pakete, die UDP/TCP zu Grunde legen, werden an Internetadressen gesendet
- Benutzer müssen den expliziten Wert einer Internetadresse nicht kennen, sondern können einen DNS-Host Namen verwenden
- Beispiel: Java Klasse „InetAddress“
 - » Aufruf einer statischen Methode, der ein DNS-Host Name übergeben wird
 - » Erzeugung einer Instanz der Klasse InetAddress mit dem Wert der expliziten Internetadresse, der über DNS zurückgegeben wird
 - » Codebeispiel:

```
InetAddress einComputer = InetAddress.getByName(„localhost“)
```

Beispiel: Client/Server-Kommunikation (UDP) in JAVA

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByLine(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         }catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

```
1 // vgl. Coulouris et al. [2002:166]
2 // Abb. 4.4: UDP-Server empfaengt wiederholt eine Anforderung und sendet sie an den Client zurueck
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPServer{
7     public static void main(String args[]){
8         try{
9             DatagramSocket aSocket = new DatagramSocket(6789);
10             byte[] buffer = new byte[1000];
11             while(true){
12                 DatagramPacket request = new DatagramPacket(buffer, buffer.length);
13                 aSocket.receive(request);
14                 DatagramPacket reply = new DatagramPacket(request.getData(),
15                     request.getLength(), request.getAddress(), request.getPort());
16                 aSocket.send(reply);
17             }
18         }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
19         }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
20     }
21 }
```


Beispiel: UDP-Client [I]



```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [II]

Import der Klassenbibliotheken / Packages

✓ `java.net.*`:

- Enthält verschiedene Klassen zur Kommunikation im Internet
- Netzwerkverbindungen können geöffnet werden

✓ `java.io.*`:

- Enthält verschiedene Klassen zur Eingabe/Ausgabe
- Ohne den vorherigen Import eines Packages muss eine Klasse unter Angabe des Packages deklariert werden („`java.Package.Klasse`“)

✓ Andere Packages verfügbar, z.B. `Java.awt.*`, ...

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [III]

Deklaration einer Klasse „UDPCClient“ mit Zugriffsrechten auf die Klasse und deren Methoden und Daten

✓ **public:**

- Öffentliche Komponente/Klasse
- Zugriff auch durch andere Klassen möglich

✓ Andere Zugriffsrechte:

- „private“: Zugriff nur innerhalb der deklarierten Klasse erlaubt
- „protected“: Zugriff nur durch Klassen im selben Package (und daraus abgeleiteten Klassen)
- „default“: Zugriff nur durch Klassen eines Packages (keine daraus abgeleiteten Klassen)

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [IV]

Ein Java-Programm (=auszuführende Klasse) muss mindestens eine Methode „main()“ enthalten

- Diese wird als Startmethode automatisch ausgeführt
- Muss öffentlich (**public**) und statisch (**static**) sein
- Die Methode liefert keinen Wert zurück (Rückgabotyp „**void**“)
- Der Aufruf des Programms erfordert zusätzliche Übergabe von Argumenten in der Kommandozeile
- Als Parameter werden die Kommandozeilenargumente des Aufrufs als Zeichenkette (Feld von Zeichenketten „String args[]“) übergeben
 - Zeichenketten werden als Objekte der Klasse „String“ abgelegt
 - „args []“ ist
 - ✓ eine Referenz auf ein Feld vom Datentyp String (Zeichen)
 - ✓ Array von String-Referenzen

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [V]

Kommandozeilenargumente/übergebene Parameter des Aufrufs der Methode „main()“:

- ✓ 1. Parameter = args[0] = Nachricht, nämlich Referenz auf ein Feld m vom Typ byte
- ✓ 2. Parameter = args[1] = Host-Name (Server): eingegeben als DNS-Adresse (wird umgewandelt in explizite IP-Adresse)
- ✓ Überprüfung mit Informationspaar: „Nachricht“ + „localhost“

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [VI]

Fehlerbehandlung durch Umschließen der Anweisungen durch einen „try/catch“-Block

- ✓ try {}: enthält „riskante“ Anweisungen
- ✓ catch {}: wird ausgeführt, wenn im try{}-Block (oder bei Ausführung der gesamten Methode main()) eine Exception (Objekt der Methode) ausgeworfen wird

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [VII]

Erzeugung eines `DatagramSocket aSocket` ohne Übergabe von Parametern (der Portnummer) an den Konstruktor

- ✓ ermöglicht dem Client, den Port für die Verbindung frei zu wählen
- ✓ kann auch explizit mit übergeben werden, z.B. `DatagramSocket aSocket = new DatagramSocket(2345)`

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```


Beispiel: UDP-Client [VIII]

- ✓ Umwandlung des ersten in der Kommandozeile eingegebenen Parameters (Nachricht m) in ein Byte-Array

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```


Beispiel: UDP-Client [IX]

- ✓ Umwandlung des zweiten in der Kommandozeile eingegebenen Parameters aHost (DNS-Host-Name) in eine IP-Adresse

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [X]

- ✓ Deklaration des Server-Ports (muss bekannt sein) vom Typ int

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XI]

- ✓ Erzeugung eines `DatagramPacket` request
- ✓ Array enthält:
 - » Nachricht m (als Byte-Array)
 - » Die Länge der Nachricht
 - » Die Internetadresse des Servers aHost
 - » Den Port des Servers (serverPort = 6789)

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XII]

- ✓ Versand des DatagramPacket „request“ über den zuvor definierten DatagramSocket aSocket

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XIII]

- ✓ Initialisierung eines Eingangspuffers buffer als Byte-Array der Größe 1000

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XIV]

- ✓ Erzeugung eines `DatagramPackets` reply
- ✓ Array enthält:
 - » Inhalt des Eingangspuffers (Byte-Array buffer)
 - » Die Länge des Byte Arrays `buffer.length`

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XV]

- ✓ Empfang des DatagramPacket „reply“ über den zuvor definierten DatagramSocket aSocket

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XVI]

- ✓ Ausgabe der empfangenen Antwort auf der Konsole

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```


Beispiel: UDP-Client [XVII]

- ✓ Schließen/Deinitialisierung des Sockets aSocket

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XVIII]

- ✓ Fehlerbehandlung der Konstruktoren der Klasse DatagramSocket
- ✓ Ausgabe der Fehlermeldung auf der Konsole
- ✓ Beispiele:
 - Port wird bereits verwendet
 - Port ist reserviert (<1024)

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Client [XIX]

- ✓ Fehlerbehandlung der Ein-/Ausgabe der Methoden send + receive
- ✓ send abhängig von korrekter Eingabe bei Aufruf der Klasse UDPCClient und der darin enthaltenen Methode main()

```
1 // vgl. Coulouris et al. [2002:164]
2 // Abb. 4.3: UDP-Client sendet Nachricht an den Server und erhaelt eine Antwort
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPCClient{
7     public static void main(String args[]){
8         // args give message contents and server hostname
9         try {
10             DatagramSocket aSocket = new DatagramSocket();
11             byte[] m = args[0].getBytes();
12             InetAddress aHost = InetAddress.getByName(args[1]);
13             int serverPort = 6789;
14             DatagramPacket request = new DatagramPacket(m, args[0].length(), aHost, serverPort);
15             aSocket.send(request);
16             byte[] buffer = new byte[1000];
17             DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
18             aSocket.receive(reply);
19             System.out.println("Reply: " + new String(reply.getData()));
20             aSocket.close();
21         } catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
22         } catch (IOException e){System.out.println("IO: " + e.getMessage());}
23     }
24 }
```

Beispiel: UDP-Server

- Erzeugung eines `DatagramSocket`s mit expliziter Übergabe von Parameter (Portnummer) an den Konstruktor
- Initialisierung des Eingangspuffers `buffer` als Byte-Array
- Schleife zum Warten auf eine Nachricht
- Bereitstellung des zu empfangenden `DatagramPackets` aus dem Eingangspuffer
- Empfangen eines potenziellen `DatagramPacktes` am Socket `aSocket`
- Initialisierung eines `DatagramPackets` als Antwort mit dem Inhalt der empfangenen Nachricht
- Versand des `DatagramPackets` über Socket

```
1 // vgl. Coulouris et al. [2002:166]
2 // Abb. 4.4: UDP-Server empfaengt wiederholt eine Anforderung und sendet sie an den Client zurueck
3
4 import java.net.*;
5 import java.io.*;
6 public class UDPServer{
7     public static void main(String args[]){
8         try{
9             DatagramSocket aSocket = new DatagramSocket(6789);
10            byte[] buffer = new byte[1000];
11            while(true){
12                DatagramPacket request = new DatagramPacket(buffer, buffer.length);
13                aSocket.receive(request);
14                DatagramPacket reply = new DatagramPacket(request.getData(),
15                    request.getLength(), request.getAddress(), request.getPort());
16                aSocket.send(reply);
17            }
18        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());}
19        }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
20    }
21 }
```