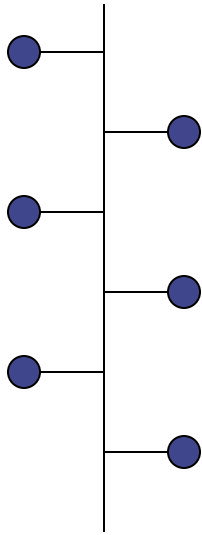


## Introduction to Communication Networks and Distributed Systems



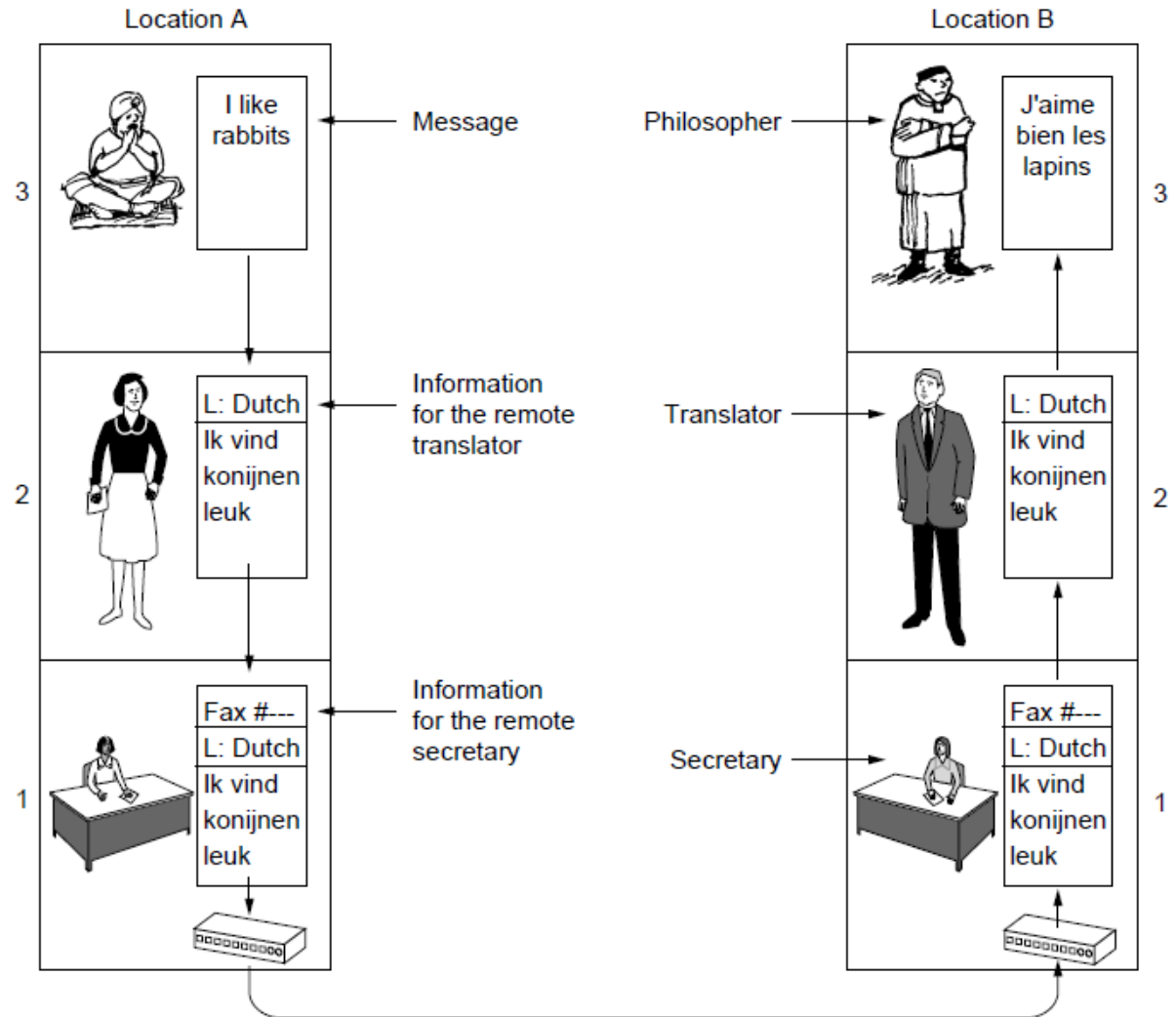
### ***Unit 2: Reference Models and Inter Process Communications***

# Networking...The reality is even more complex...

Philosophers

Assistants

Office clerks



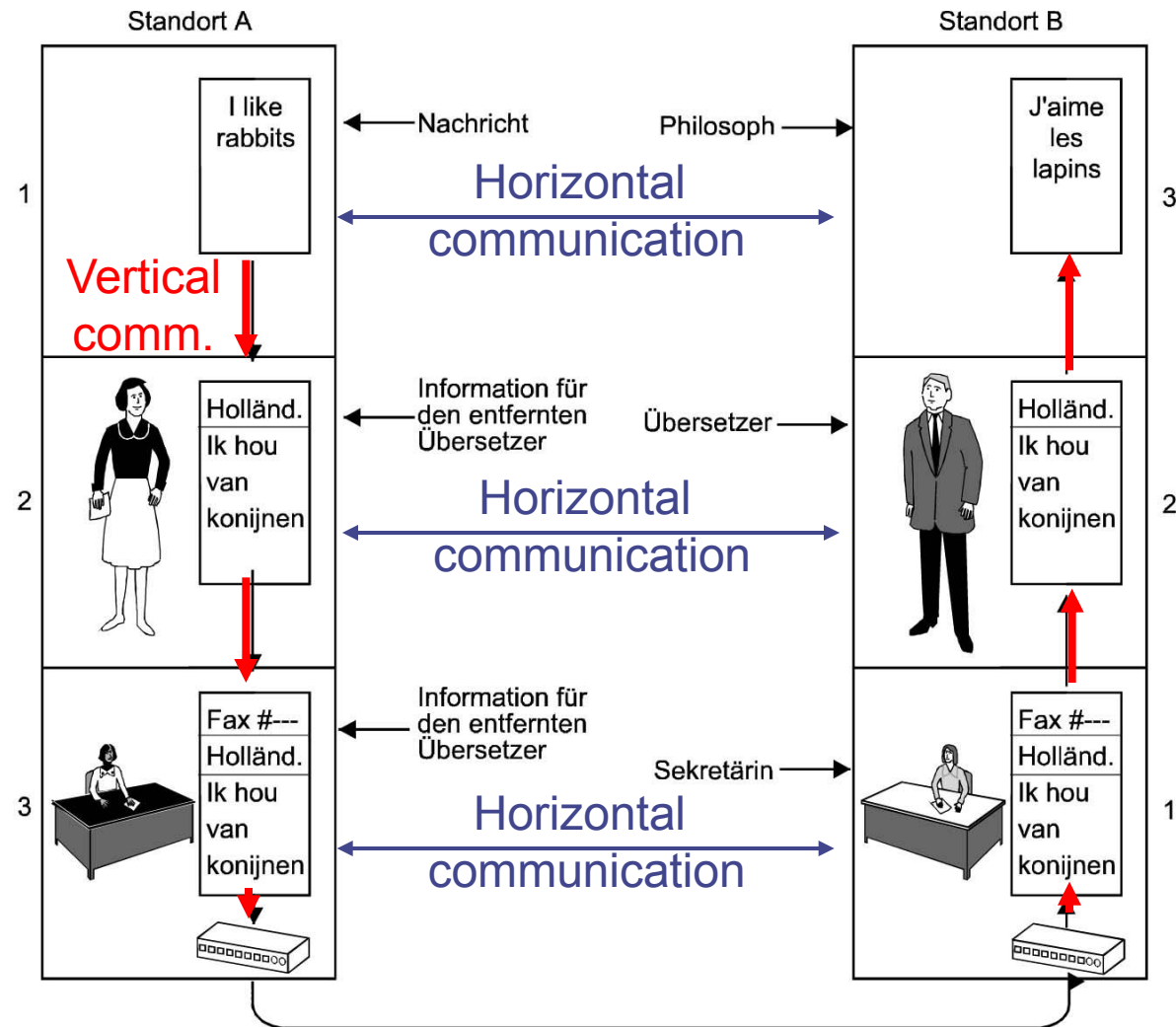
# The reference model

- To keep complexity of communication systems tractable:
  - division in subsystems with clearly assigned responsibilities – layering
- Each layer offers a particular service
  - more abstract and more powerful the higher up in the layering hierarchy
- To provide a service, a layer has to be distributed over remote devices
- Remote parts of a layer use a protocol to cooperate
  - Make use of service of the underlying layer to exchange data
  - Protocol is a horizontal relationship, service a vertical relationship
- Layers/protocols are arranged as a (protocol) stack
  - One atop the other, only using services from directly beneath
  - ⇒ Strict layering

# Analogy: Nested layers as nested translations

- Vertical vs. horizontal communication

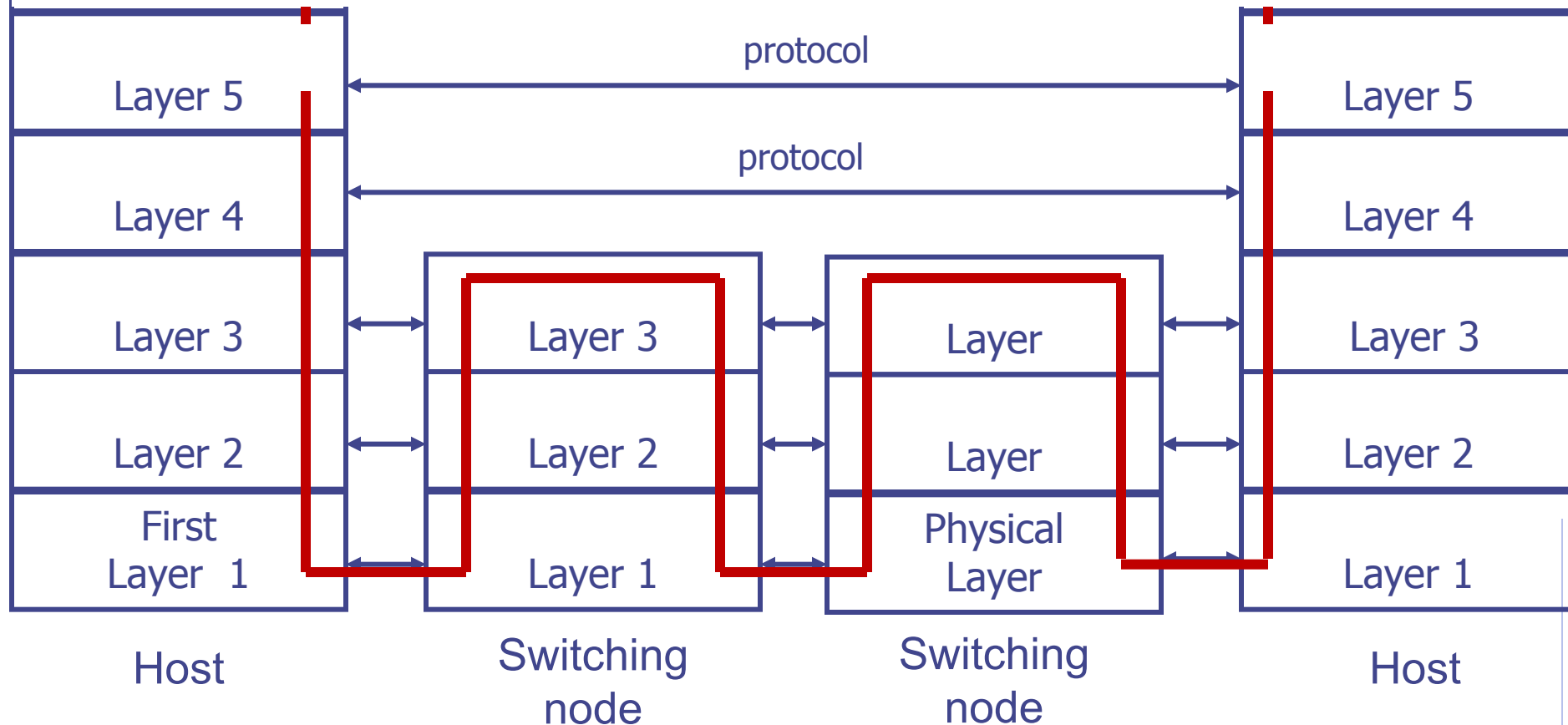
- Vertical: always real
- Horizontal: may be real or virtual



Horizontal (real!) communication

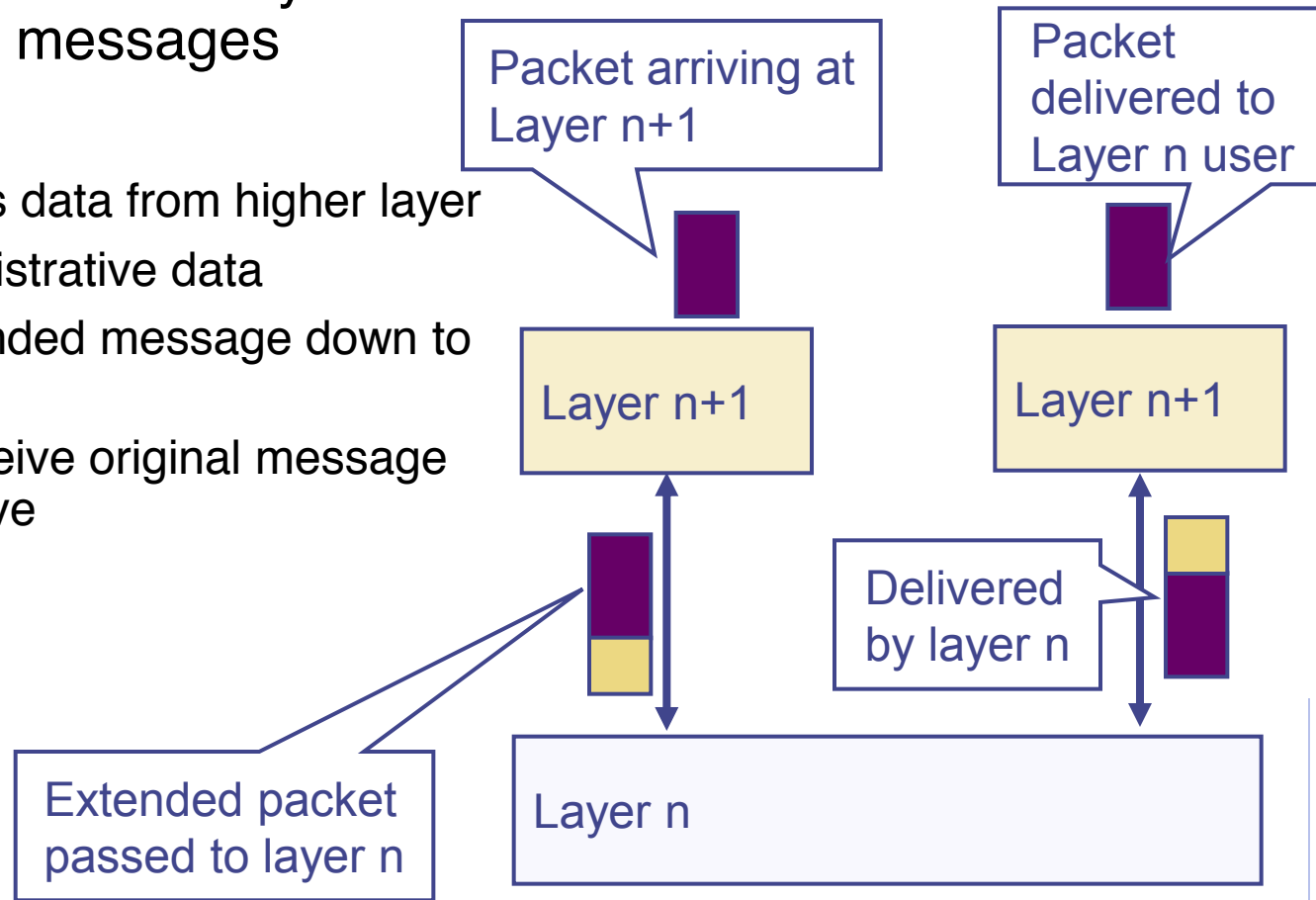
# Multi-layer Architecture

- Number of Layers, and  
{ services, naming and addressing conventions } / Layer
- Functions to be executed in each layer
- Protocols: (host-to-host, node-to-node, host-to-node)



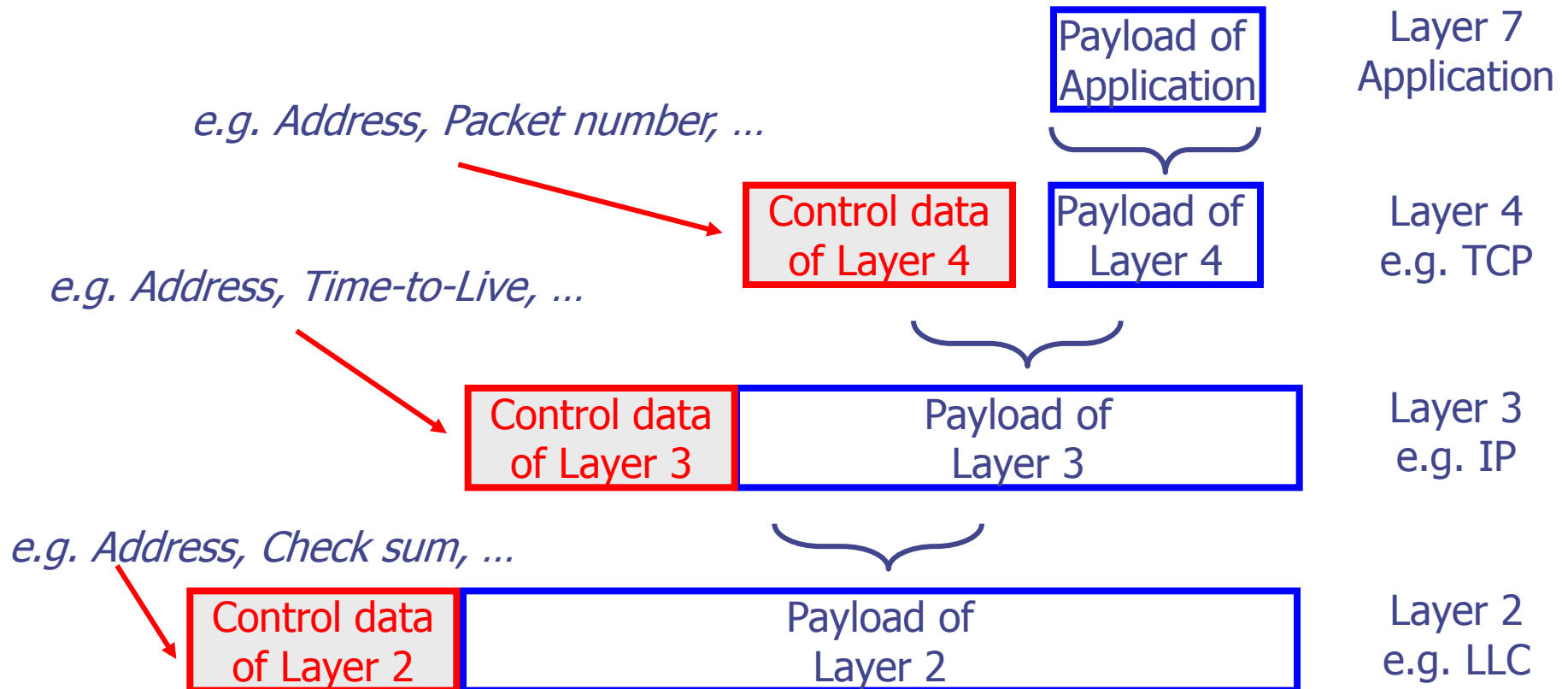
# Protocols and messages

- When using lower-layer services to communicate with the remote peer, administrative data is usually included in those messages
- Typical example
  - Protocol receives data from higher layer
  - Adds own administrative data
  - Passes the extended message down to the lower layer
  - Receiver will receive original message plus administrative data
- Encapsulating
  - Header or trailer



# Embedding messages

- Messages from upper layers are used as payload for messages in lower layers



# How to structure functions/layers?

- Many functions have to be realized
- Not each function is necessary in each Layer..
- How to actually assign them into layers to obtain a real, working communication system?
  - This is the role of a specific reference model
- Two main reference models exist
  - ISO/OSI reference model (International Standards Organization Open Systems Interconnection)
  - TCP/IP reference model (by IETF – Internet Engineering Taskforce)



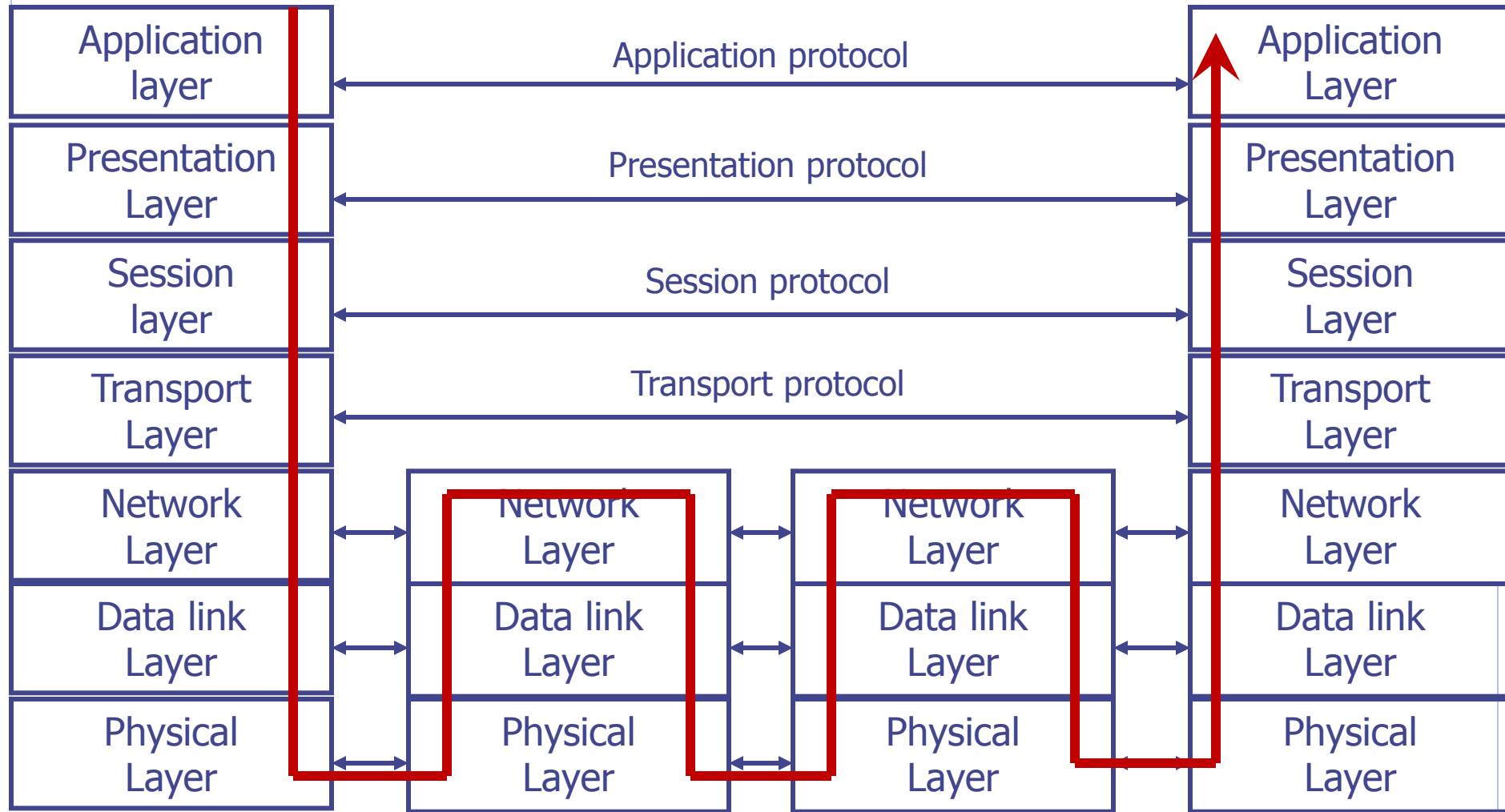
# Standardization

- To build large networks, standardization is necessary
- Traditional organization
  - **ISO**- Int. Standardization Organization , **ITU** - (Int. Telecomm. Union)
  - world-wide, group national bodies, relatively slow “time to market”
- Internet
  - Mostly centered around the **Internet Engineering Task Force (IETF)** with associated bodies (Internet Architectural Board, Internet Research Task Force, Internet Engineering Steering Group)
  - Consensus oriented, focus on working implementations
  - Hope is quick time to market, but has slowed down considerably in recent years
- **IEEE Committee 802** – driving the Link Layer!
- **Manufacturer bodies** – defining de-facto standards and profiles for the IEEE/INTERNET/ ...

# ISO/OSI reference model

- Basic design principles
  - One layer per abstraction of the “set of duties”
  - Choose layer boundaries such that information flow across the boundary is minimized (minimize inter-layer interaction)
  - Enough layers to keep separate things separate, few enough to keep architecture manageable
- Result: 7-layer model
  - Not strictly speaking an architecture, because
  - Precise interfaces are not specified (nor protocol details !)
  - Only general duties of each layer are defined

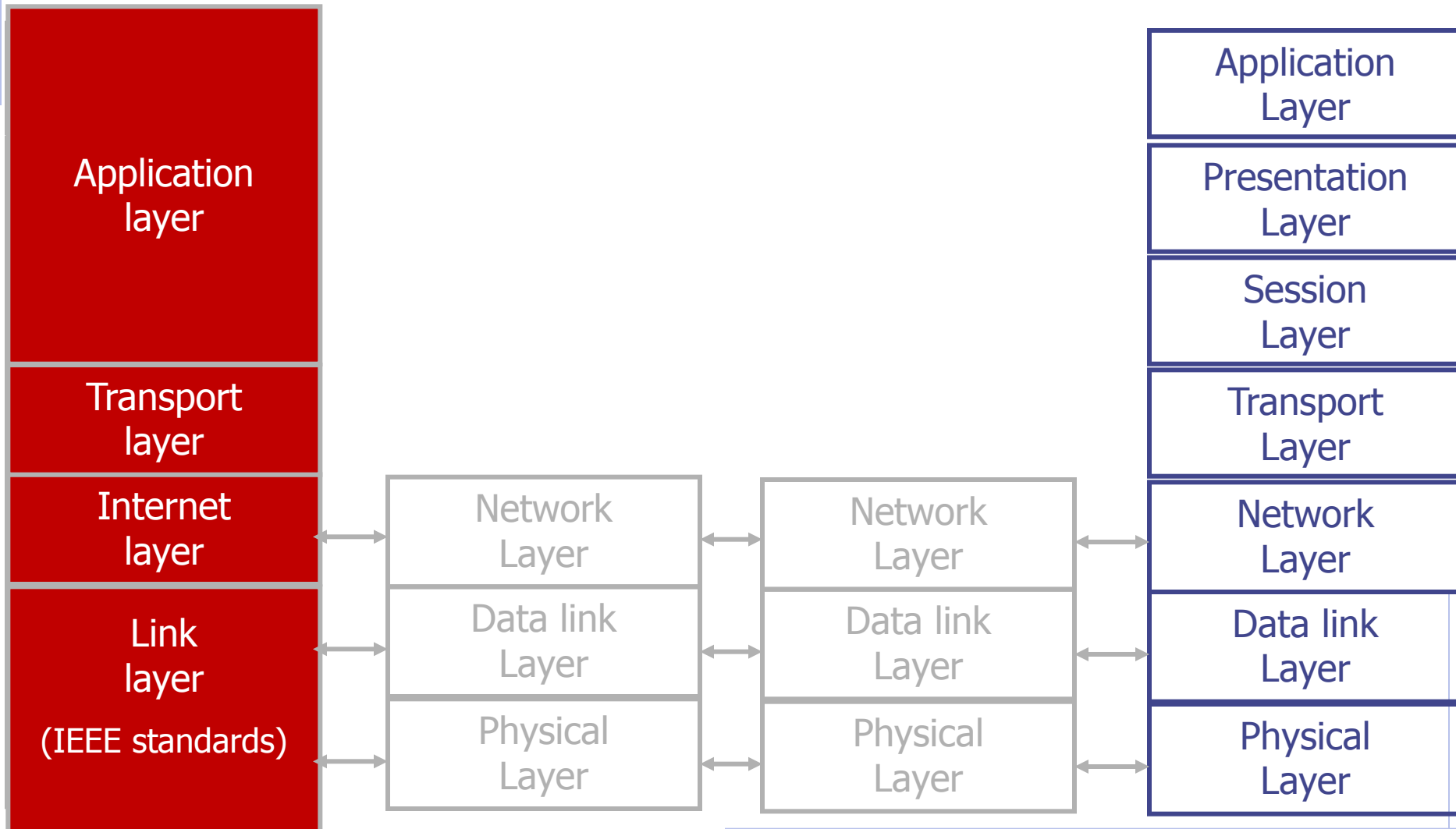
# ISO/OSI model



# 7 layers in brief

- Physical layer: Transmit raw bits over a physical medium
- Data Link layer: Provide a (more or less) error-free transmission service for data frames - also over a shared medium!
- Network layer: Solve the forwarding and routing problem for a network- bring data to a **desired host**
- **Transport layer**: Provide (possibly reliable, in order) end-to-end communication, overload protection, fragmentation to **processes**  
*“Bringing data from process A to B with sufficient quality”*
- Session layer: Group communication into sessions which can be synchronized, checkpointed, ...
- Presentation layer: Ensure that syntax and semantic of data is uniform between all types of terminals
- Application layer: Actual application, e.g., protocols to transport Web pages

# Internet Model model (in red) vs. ISO/ OSI



# Some example protocols

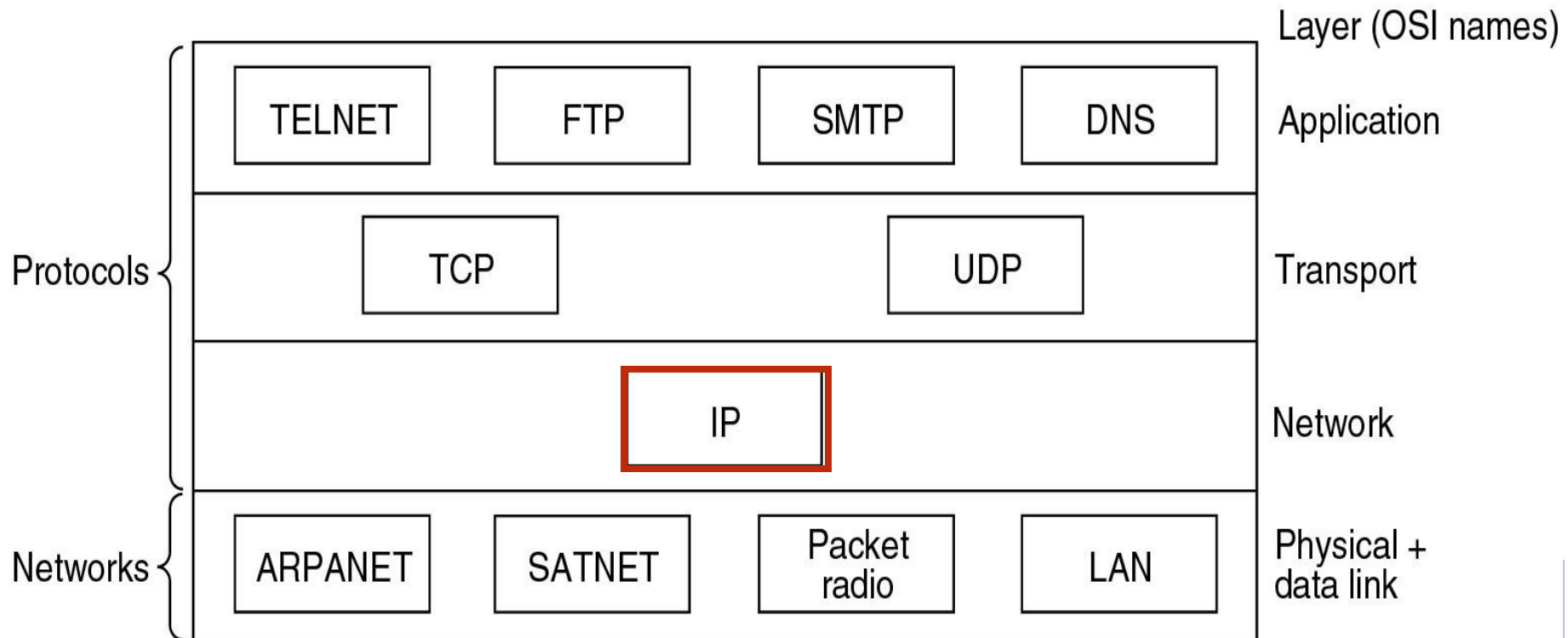
- A communication architectures needs standard protocols in addition to a layering structure
- And: some generic rules & principles which are not really a protocol but needed nonetheless
  - Example principle: end-to-end
  - Example rule: Naming & addressing scheme
- Popular protocols of the 5-layer reference model
  - Data link layer: Ethernet & CSMA/CD
  - Network layer: Internet Protocol (IP)
  - Transport layer: Transmission Control Protocol (TCP)

# Internet reference model

- Historically based on ARPANET, evolving to the Internet
  - Started out as little university networks, which had to be interconnected
- Some generic rules & principles
  - Internet connects **networks**
    - Minimum functionality assumed (just unreliable packet delivery!)
    - Internet layer (IP): packet switching, addressing, routing & forwarding
      - ➔ ***Internet over everything***
  - End-to-end
    - Any functionality should be pushed to the instance needing it!
  - Fate sharing
- In effect only two layers really defined... Internet and Transport Layer - Lower and higher layers not really defined
  - ➔ *Anything over internet*
- New Applications do NOT need any changes in the NETWORK!
  - Compare with the telephone network!!!

# The Internet Suite of Protocols

- Over time, suite of protocols evolved around core TCP/IP protocols  
➔ *Internet Protocol Suite is also refereed to as TCP/IP Protocol Suite*



So-called “**hourglass model**”: Thin waist of the protocol stack at IP, above the technological layers



# Naming & addressing in the Internet Stack

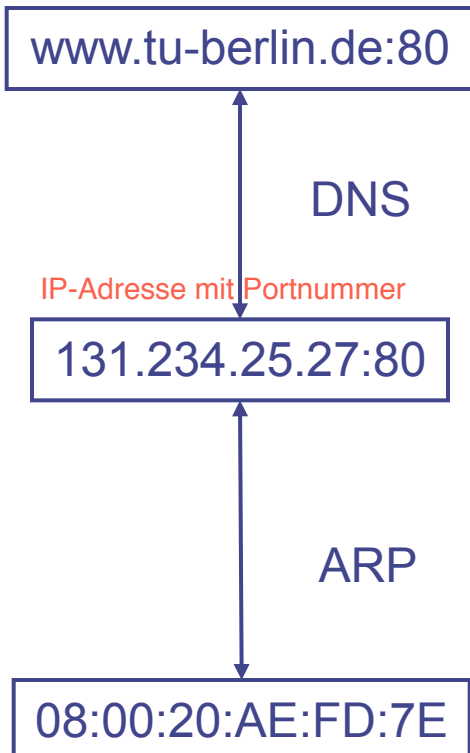
- Names: Data to identify an entity exist on different levels
  - Alphanumerical names for machines: [www.tu-berlin.de](http://www.tu-berlin.de)
- Address: Data how/where to find an entity
  - Address of a network device in an IP network: An IP address
    - IPv4: 32 bits, structured into 4x8 bits
    - Example: 131.234.20.99 (dotted decimal notation)
  - Address of a network: Some of the initial bits of an IP address
- Address of a networked device in the LOCAL AREA (IEEE 802 standardized) network...
  - 48 bits, hexadecimal notation, example: 08:00:20:ae:fd:7e

# Mapping

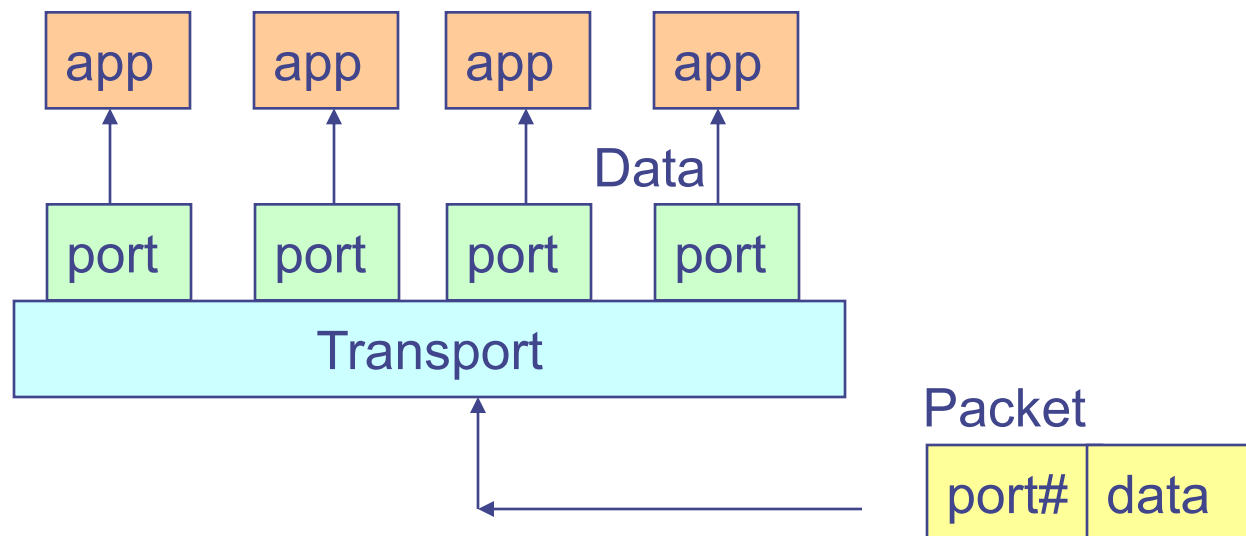
TCP/IP verwendet Adressen, keine Namen

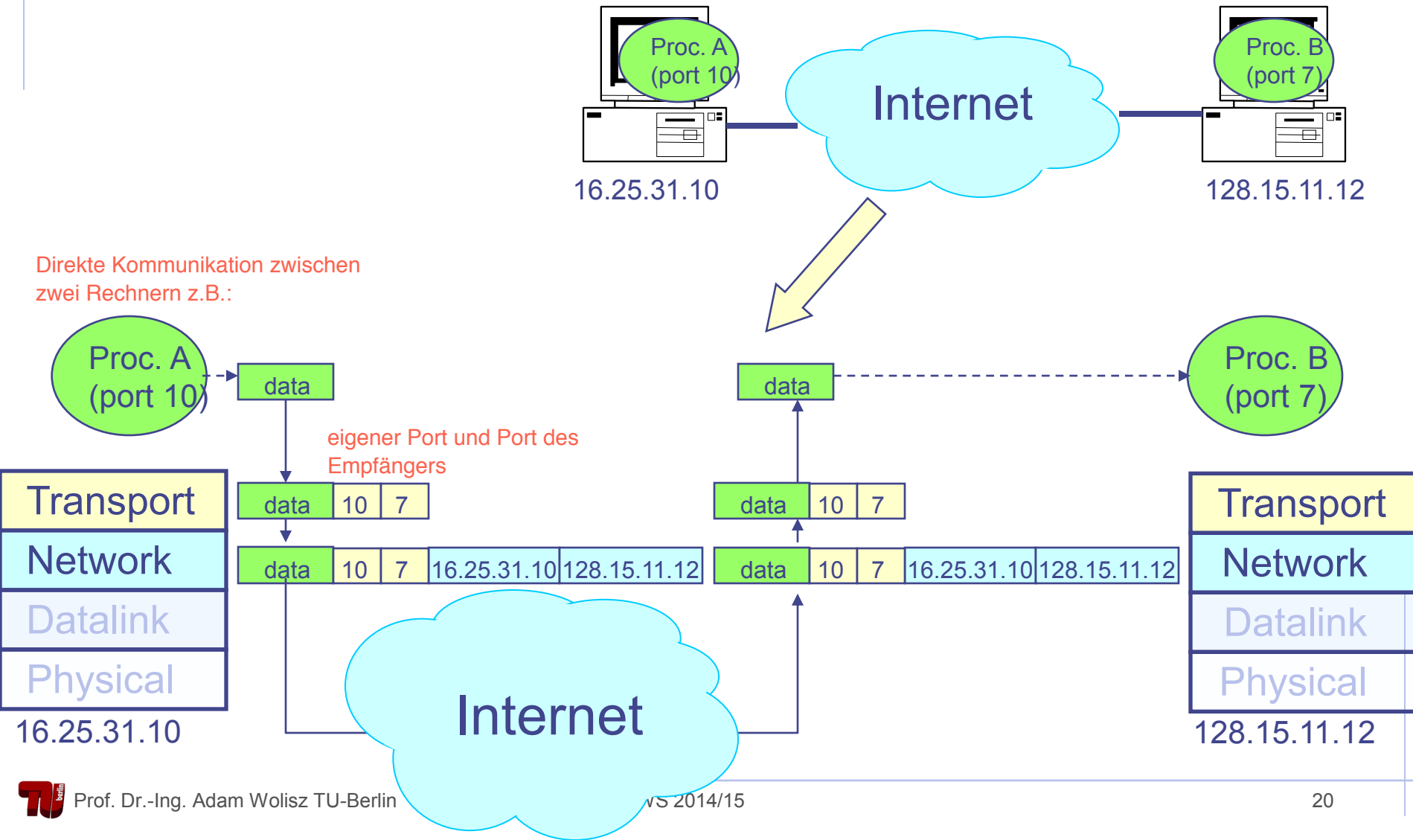
- Needed: Mapping from name to address
  - ➔ Realized by separate protocols
- From alphanumerical name to IP address: Domain Name System (DNS)
- Often also useful: Mapping from IP address to MAC name/address: Address Resolution Protocol (ARP)

Web server process'  
service access point

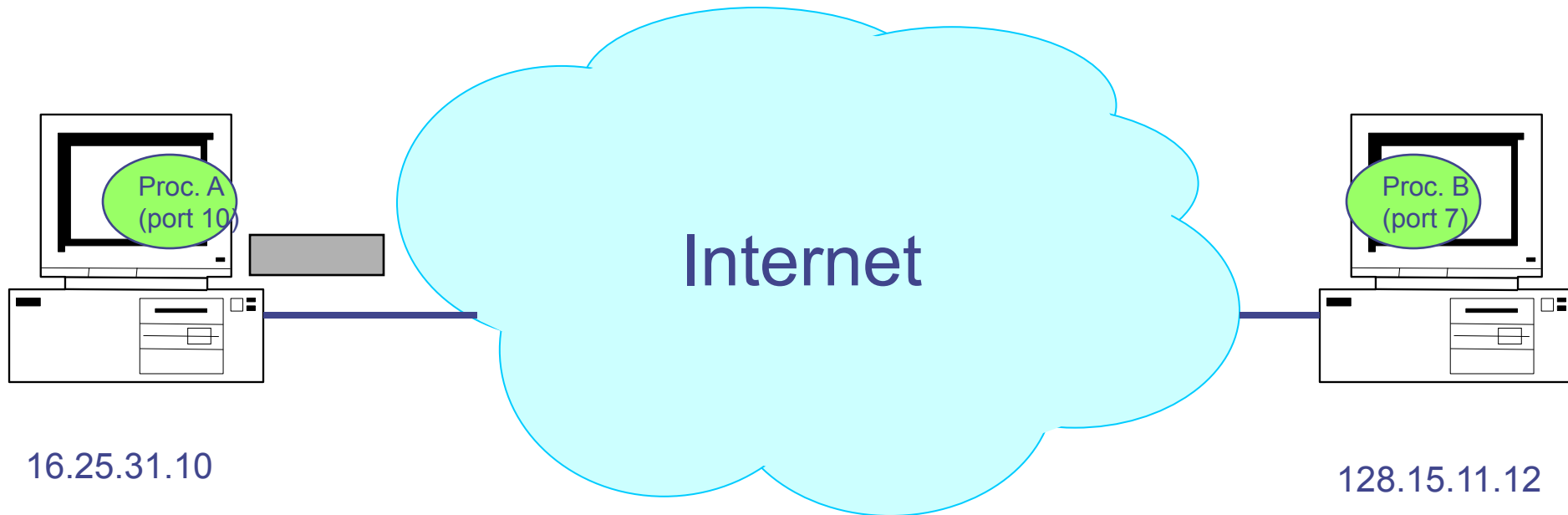


- Port is represented by a positive (16-bit) integer value
- Some ports have been reserved to support common/well known services: http 80/tcp; ftp 21/tcp; telnet 23/tcp; smtp 25/tcp;
- User level process/services generally use port number value  $\geq 1024$



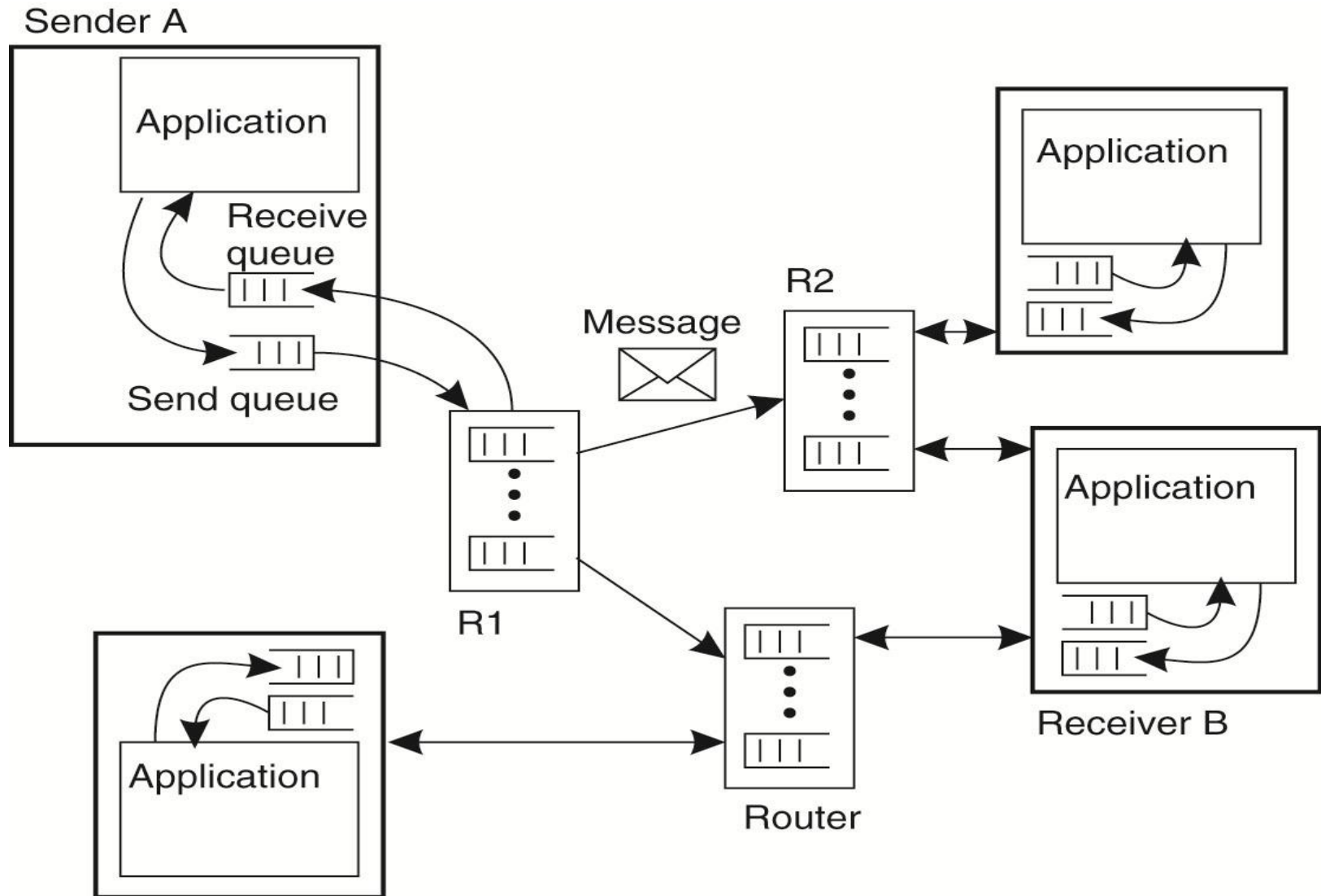


- Process A sends a packet to process B



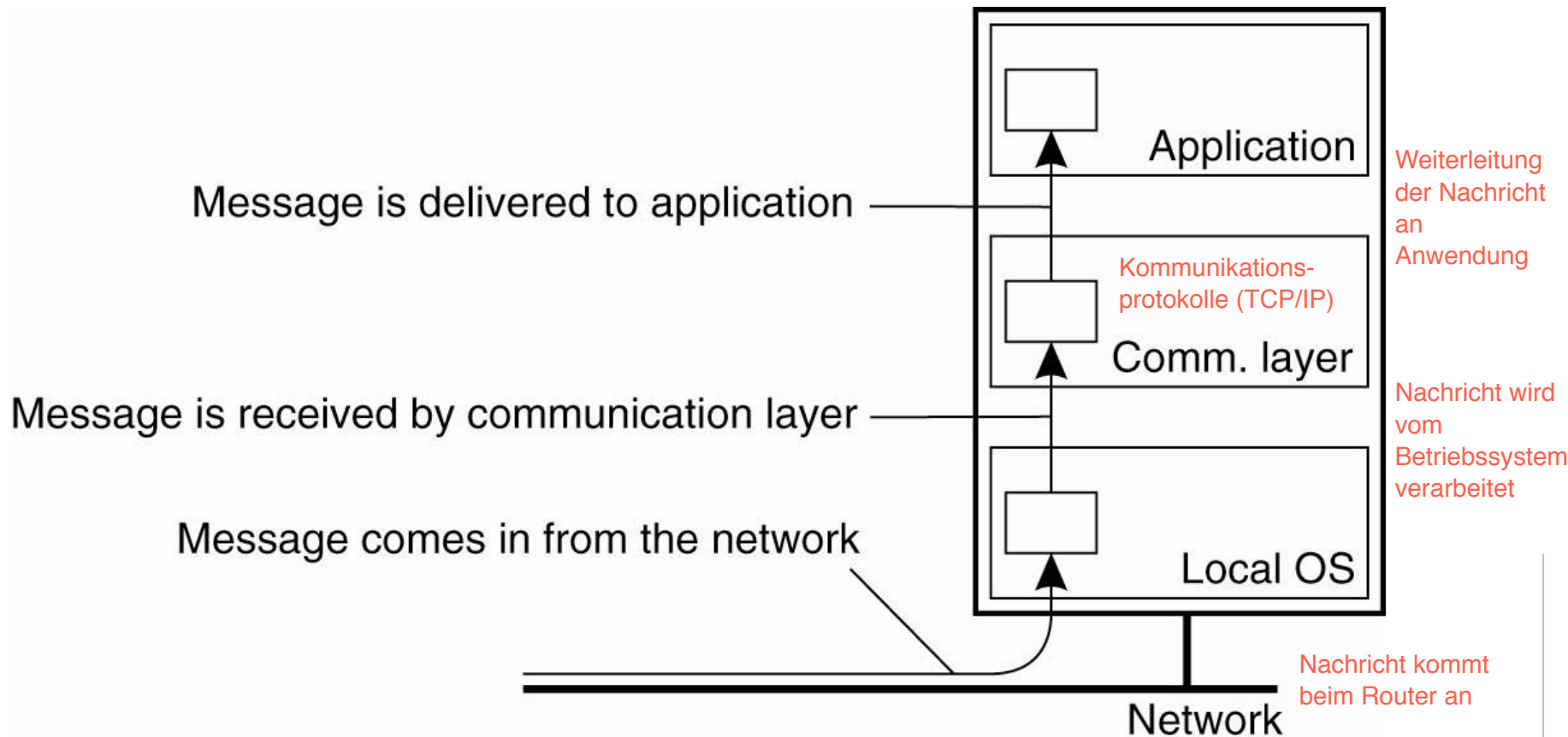
## *IP Address:*

A four-part “number” used by *Network Layer* to route a packet from one computer to another



# Message Receipt vs. Message Delivery [Tanenbaum]

- Figure 8-12. The logical organization of a distributed system to distinguish between message receipt and message delivery.



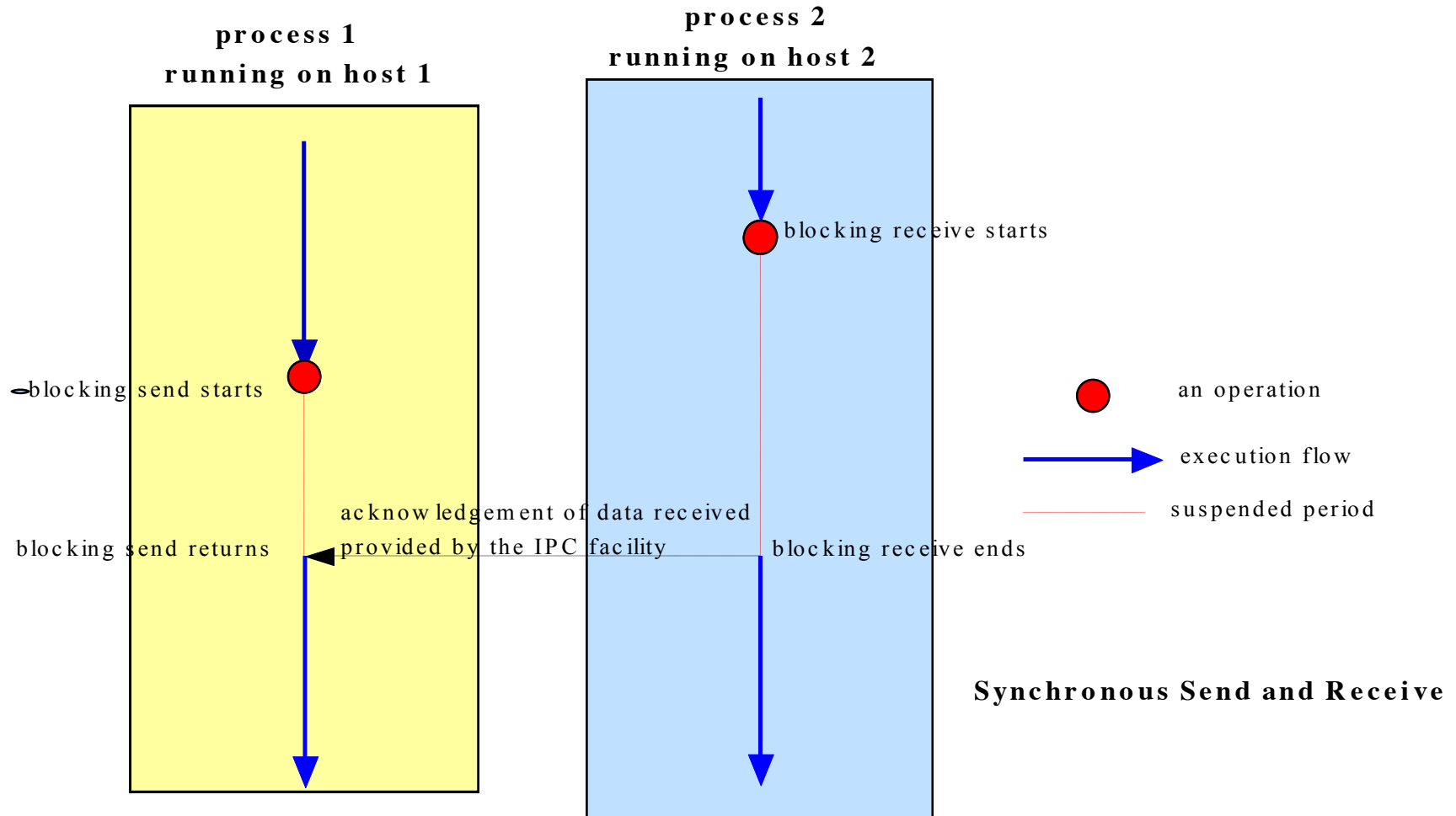
# Synchronous Interaction

“Blockierend” <-> nicht blockierend

- Blocking send
  - Blocks until message is transmitted
  - Blocks until message acknowledged
- Blocking receive
  - Waits for message to be received
- Known upper/lower bounds on execution speeds, message transmission delays and clock drift rates



# Synchronous send and receive

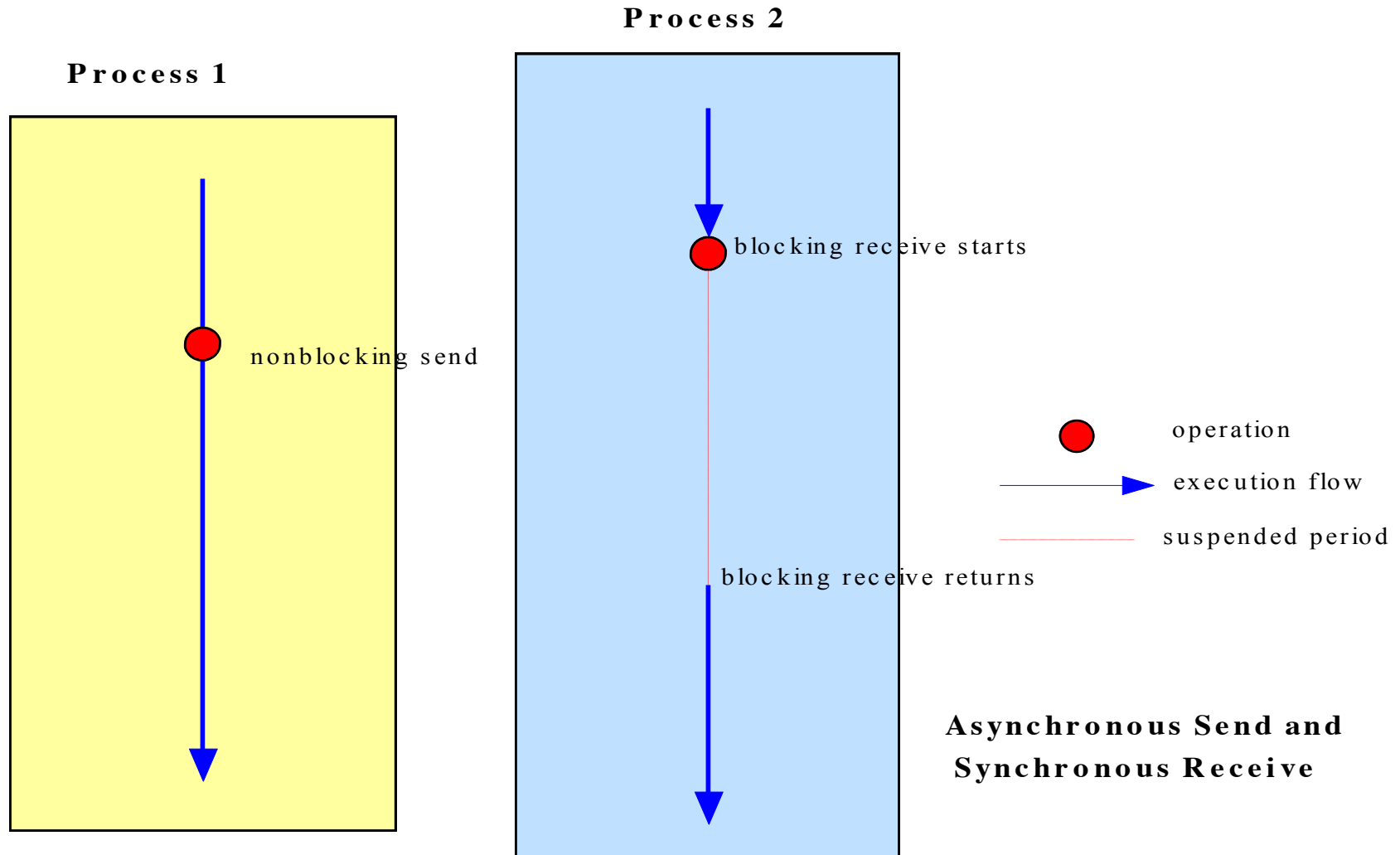


“keine gute Idee”: Jeder an das Tempo des anderen gebunden

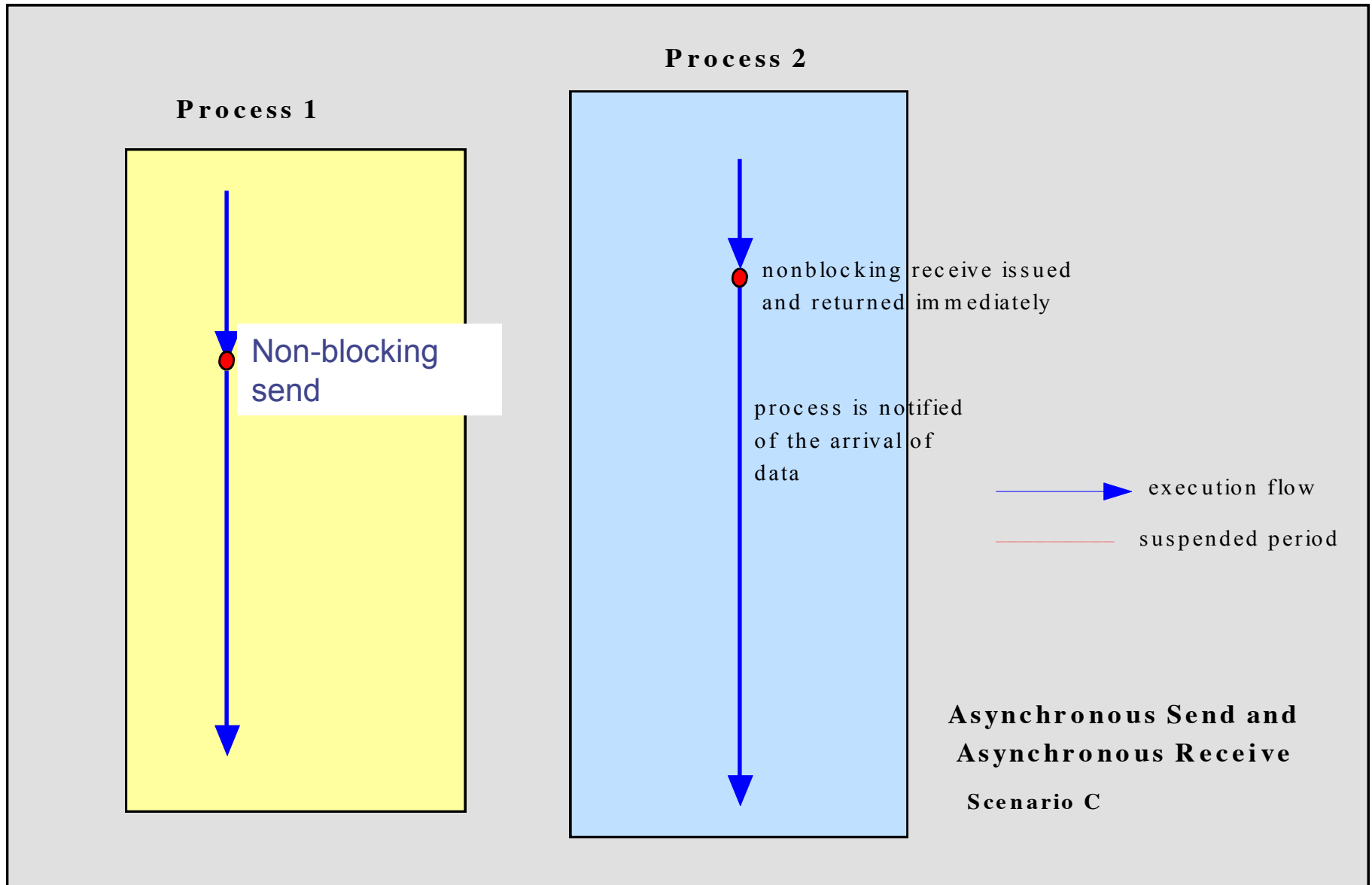
# Asynchronous Interaction

- Non-blocking send: sending process continues as soon message is queued. Puffer verwenden: Weiter machen, sobald Nachricht abgesendet und im Puffer
- Blocking or non-blocking receive:
  - Blocking:
    - Timeout. Blockierend: Benachrichtigung bei Empfang einer Nachricht
    - Threads.
  - Non-blocking: proceeds while waiting for message. Genauso funktioniert Empfang: Nachricht liegt im Puffer bis sie abgeholt wird
    - Message is queued upon arrival.
    - Process needs to poll or be interrupted.
- Arbitrary processes execution speeds, message transmission delays and clock drift rates
- Some problems impossible to solve (e.g. agreement)

# Asynchronous send and synchronous receive

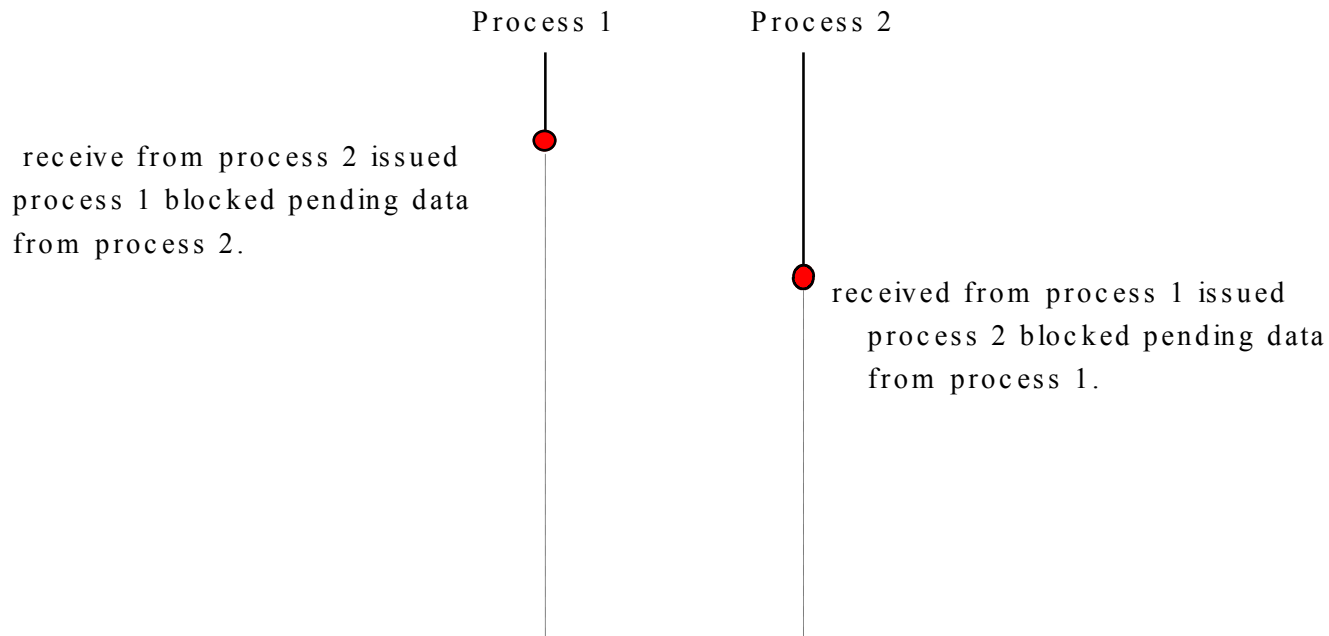


# Asynchronous send and Asynchronous receive



# Blocking, deadlock, and timeouts

- Blocking operations issued in the wrong sequence can cause deadlocks.
- Deadlocks should be avoided. Alternatively, timeout can be used to detect deadlocks.



(vorstellbare) Basis-Operationen für Nachrichtenübertragung (real nicht immer alle verwendet/verfügbar)

Primitive	Meaning
MPI_bsend	Append outgoing message to a local send buffer
MPI_send	Send a message and wait until copied to local or remote buffer
MPI_ssend	Send a message and wait until receipt starts
MPI_sendrecv	Send a message and wait for reply
MPI_isead	Pass reference to outgoing message, and continue
MPI_issend	Pass reference to outgoing message, and wait until receipt starts
MPI_recv	Receive a message; block if there is none
MPI_irecv	Check if there is an incoming message, but do not block

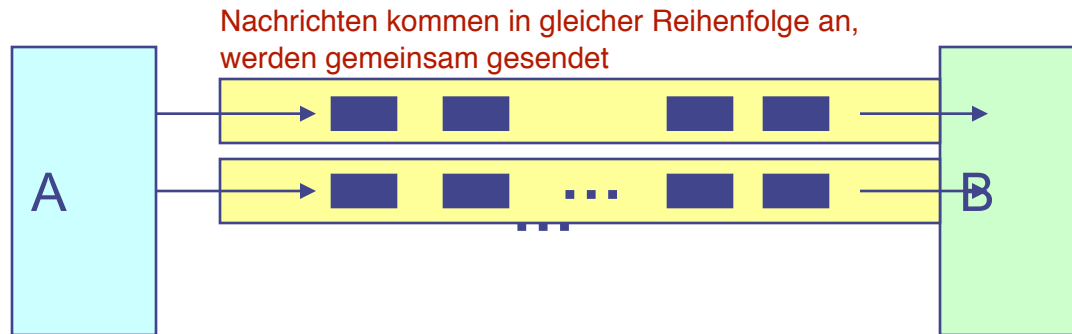
mehrere Möglichkeiten eine Nachricht zu senden, weniger Alternativen beim Empfang (nur blockierend, nicht blockierend)

Viele Varianten von "Nachricht abgesendet" -> heißt nicht unbedingt auch zuverlässig zugestellt -> in der Praxis: Dokumentation genau lesen, wie "send" implementiert ist

# Connection-oriented vs. connection-less service

- Recall telephony vs. postal service
  - Service can require a preliminary setup phase
    - ➔ **connection-oriented service**
    - Three phases: connect, data exchange, release connection
    - During connect: attention of the partner assured, resources reserved....
  - Alternative: Invocation of a service primitive at any time, with all necessary information included in the invocation
    - ➔ **connection-less service**
- **Note:** *Both are possible on top of either circuit or packet switching!*  
auch ein paketvermittelndes System kann eine Verbindung aufbauen
- Connection-oriented services - primitives to handle connection
  - CONNECT – setup a connection to the communication partner
  - LISTEN – wait for incoming connection requests
  - INCOMING\_CONN – indicate an incoming connection request
  - ACCEPT – accept a connection
  - DISCONNECT – terminate a connection

längere  
Kommunikation mit  
dem gleichen Partner



## ■ Connection-Oriented Communication verbindungsorientierter Dienst

einzelne Nachrichten,  
verschiedene Partner



## ■ Connectionless Communication verbindungsloses Kommunikation



# Typical examples of services

- ***Datagram service (Connection –less)***

- Unit of data are messages (limited length)
- Correct, but not necessarily complete or in order –
- Usually insecure/not dependable, not confirmed

Application must provide its own reliability“

- ***Reliable byte stream (connection oriented)***

- Byte stream
- Correct, complete, in order, confirmed - Processes have a guarantee that messages will be delivered (sender can check that!)
  - Possible to build reliability atop unreliable service .
- Sometimes, but not always secure/dependable

# Sockets: the Transport Layer API

Historisch: zahlreiche Betriebssysteme, bis UNIX:  
vereinheitlichtes Dateisystem -> Rechner wurden verbunden  
Übertragung der Idee von UNIX auf Kommunikation in Berkeley  
(Berkeley-Sockets)

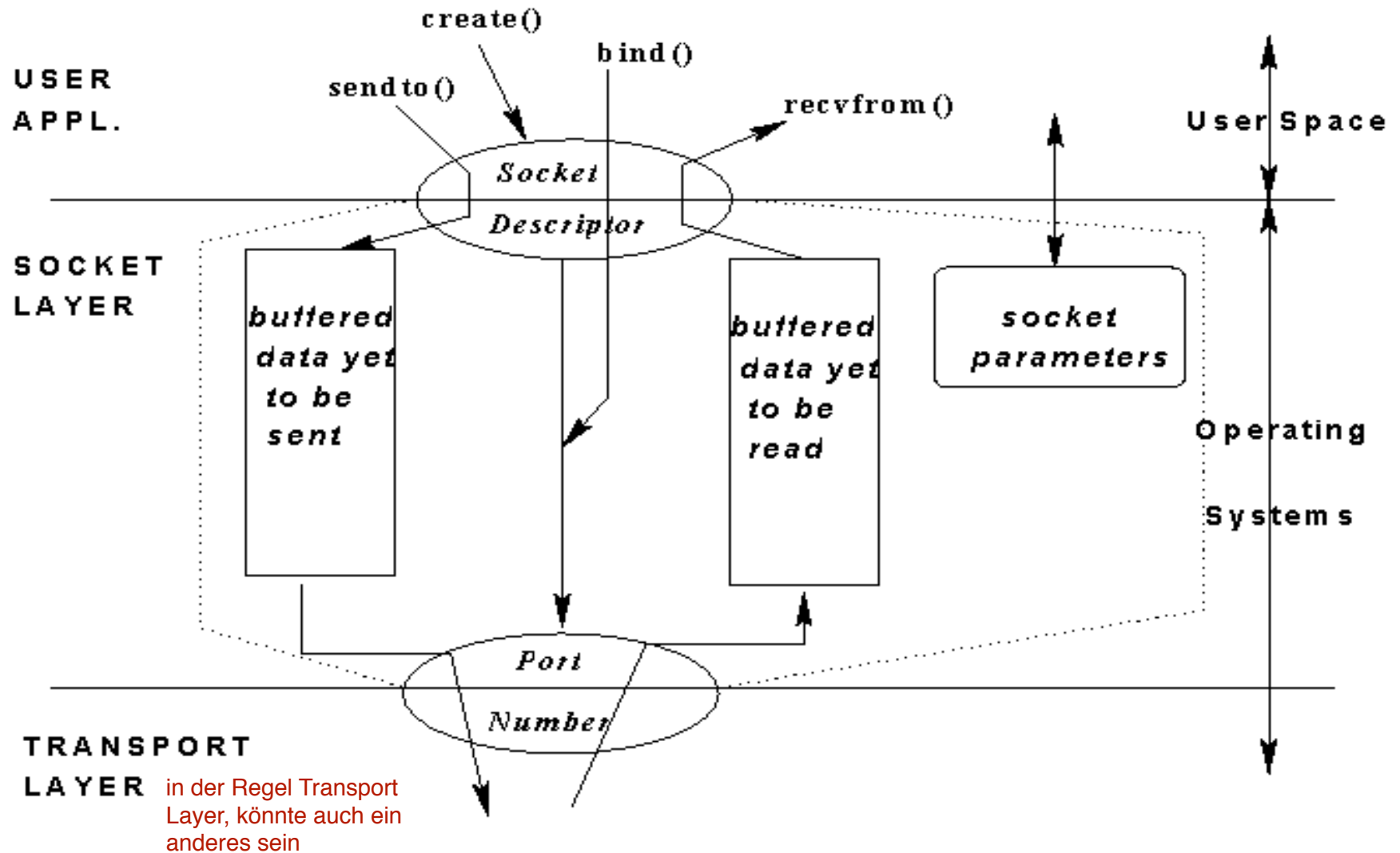
- Sockets provide an API (Application Programming Interface) for programming networks at the transport layer.

allgemeine Schnittstelle (nicht nur für Internet!) für Kommunikation

- A socket is an endpoint of a two-way communication link between two processes located on the same machine - or located on different machines connected by a network.

Endpunkte für Kommunikation in beide Richtungen

- Network communication using Sockets is similar to file I/O
  - Socket handle is treated like file handle.
  - The streams used in file I/O operation are also applicable to socket-based I/O
- Socket-based communication principles are programming language independent!
- The success of this API is based on its abstraction of all possible used network protocols/underlying network topology.
  - A socket is bound to a port number so that the transport protocol can identify the application that data destined to be sent.



# Caution: Sockets support multiple domains!!!

- Domain defined while socket is created!
  - `int socket(int domain, int type, int protocol)`
- *domain* is `AF_UNIX`, `AF_INET`, `AF_OSI`, etc.
  - `AF_INET` is for communication on the internet to IP addresses.
- *type* is either `SOCK_STREAM` (connection oriented, reliable), `SOCK_DGRAM` or `SOCK_RAW`
  - Originally more types have been envisioned
- *protocol* specifies the protocol used. It is usually 0 to say we want to use the default protocol for the chosen domain and type (note IP 4 vs. IP 6)
  - kann sich auf Internet beziehen, aber auch beliebiges anderes möglich  
meist Internet
- While nowadays INET Domain/protocols prevail, the approach is more General.
  - wichtig, sonst kann Adresse nicht korrekt interpretiert werden

# Berkeley Sockets

## Basis Operationen von Sockets

Primitive	Meaning
Socket	Create a new communication end point
Bind	Attach a local address to a socket
Listen	Announce willingness to accept connections
Accept	Block caller until a connection request arrives
Connect	Actively attempt to establish a connection
Send	Send some data over the connection
Receive	Receive some data over the connection
Close	Release the connection

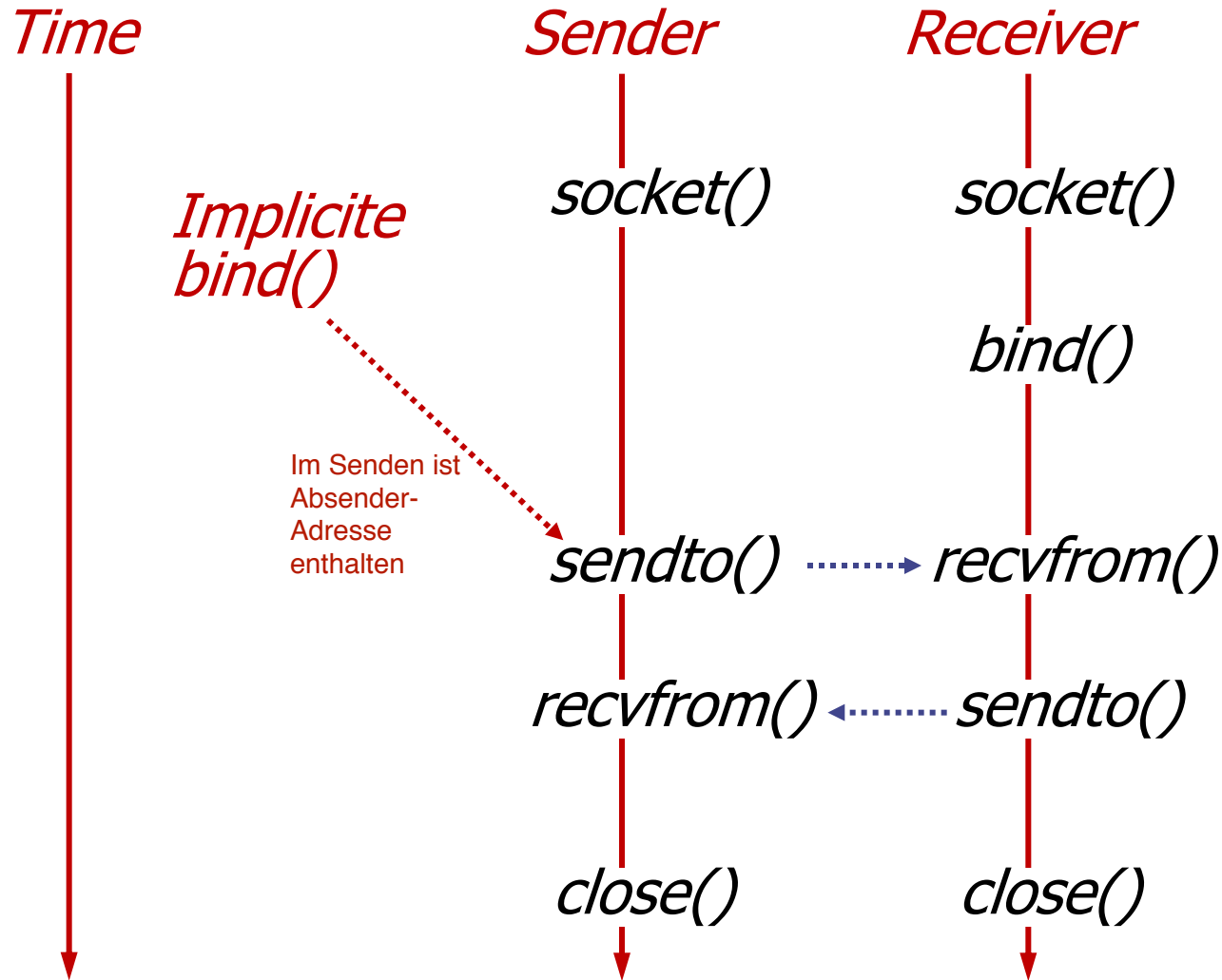
Adresse des Sockets  
festlegen

Bereitschaft  
Verbindungen zu  
empfangen (wie  
viele)  
dann  
Verbindung  
aufbauen

A VERY GOOD source of information about sockets is the Beej's Guide

<http://beej.us/guide/bgnet/> (legally free download!) Buch Tipp: "Das" Handbuch über Sockets

# Datagram sockets



- Simplest possible service: unreliable datagrams

## Sender

- `int s = socket (...);`
- `sendto (s,  
buffer,  
datasize,  
0,  
to_addr,  
addr_length);`
- `to_addr` and `addr_length` specify the destination

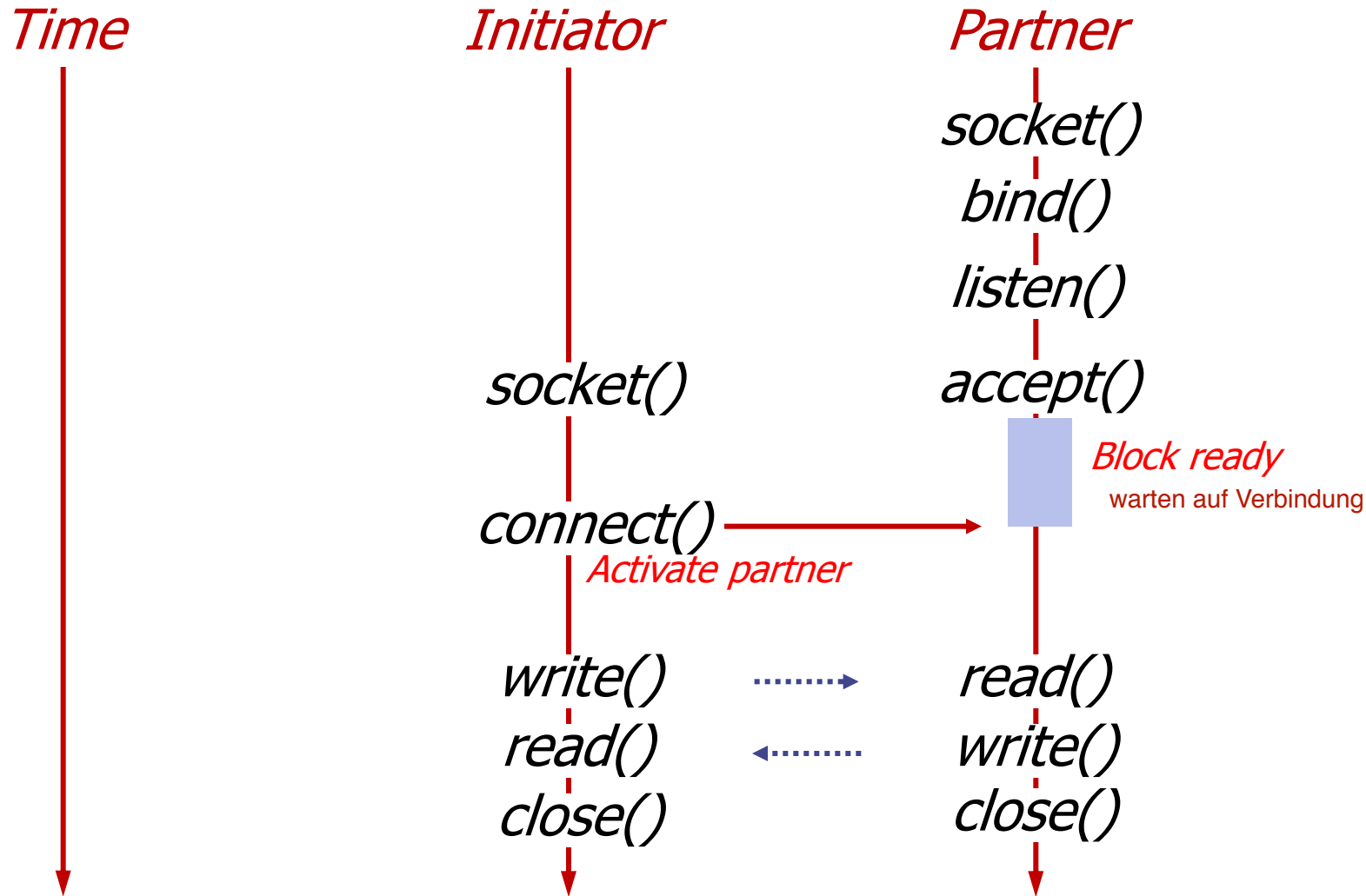
## Receiver

- `int s = socket (...);`
- `bind (s, local_addr,  
...);`
- `recv (s,  
buffer,  
max_buff_length,  
0);`
- Will wait until data is available on socket `s` and put the data into `buffer`

- Class *DatagramSocket*
  - *socket constructor* (returns free port if no arg)
  - *send DatagramPacket* (non-blocking)
  - *receive DatagramPacket* (blocking)
  - *setSoTimeout*  
(receive blocks for time  $T$  and throws *InterruptedException*)
  - *Connect*  
(throws *SocketException* if port unknown or in use)
  - *close DatagramSocket*



# Stream sockets



- For reliable byte streams, sockets have to be connected first
- Receiver has to accept connection

## Initiator (*client*)

- `int s = socket (...);`
- `connect (s,  
destination_addr,  
addr_length);`
- `send (s,buffer,  
datasize, 0);`
- Arbitrary `recv()/send()`
- `close (s);`
- Connected sockets use a `send` without address information

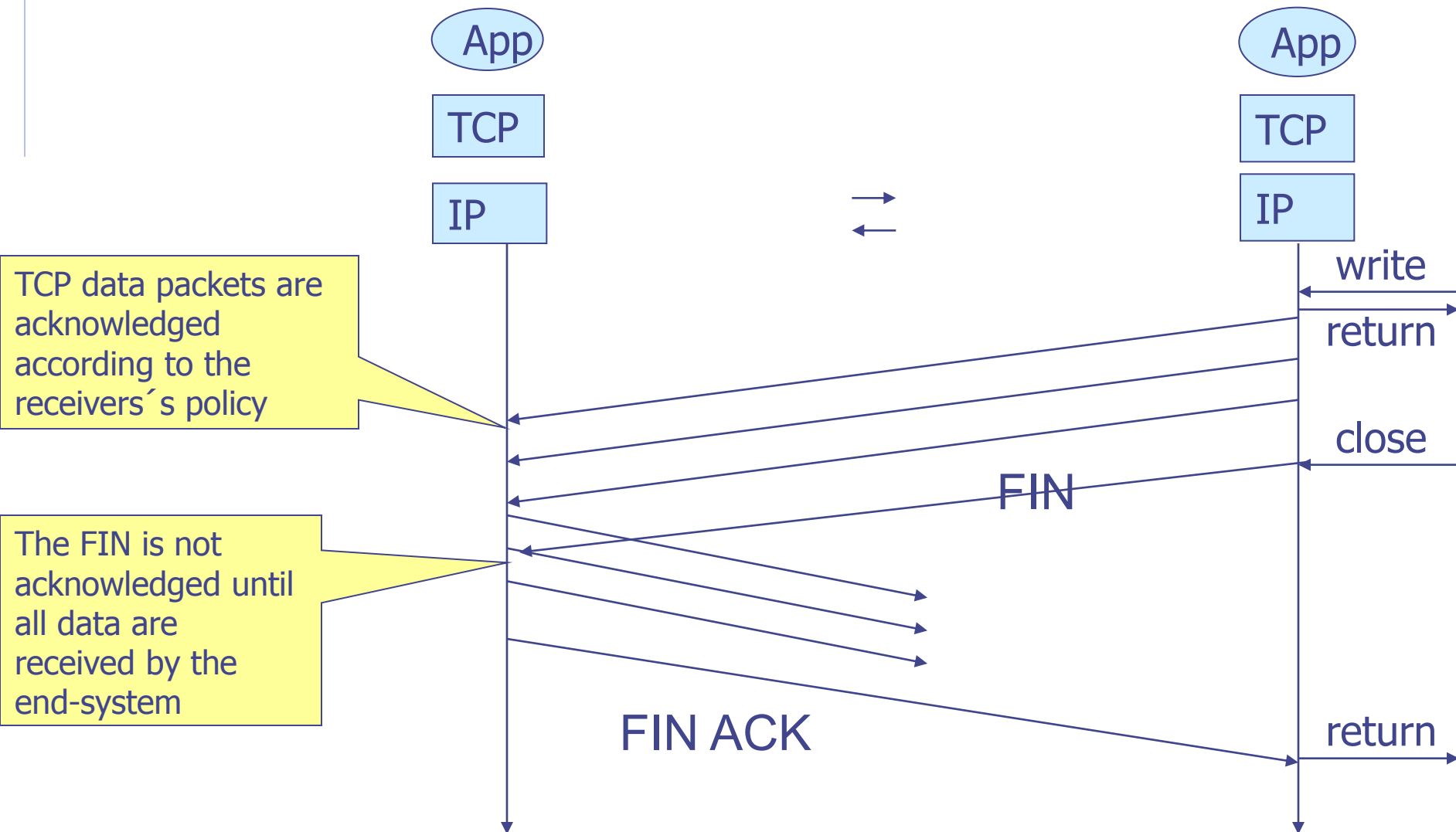
## Partner (*server*)

- `int s = socket (...);`
- `bind (s, local_addr, ...);`
- `listen (s, ...);`
- `int newsock = accept (s,  
*remote_addr, ...);`
- `recv (newsock, buffer,  
max_buff_length, 0);`
- Arbitrary `recv()/send()`
- `close (newsock);`

# Java API for Data Stream Communications

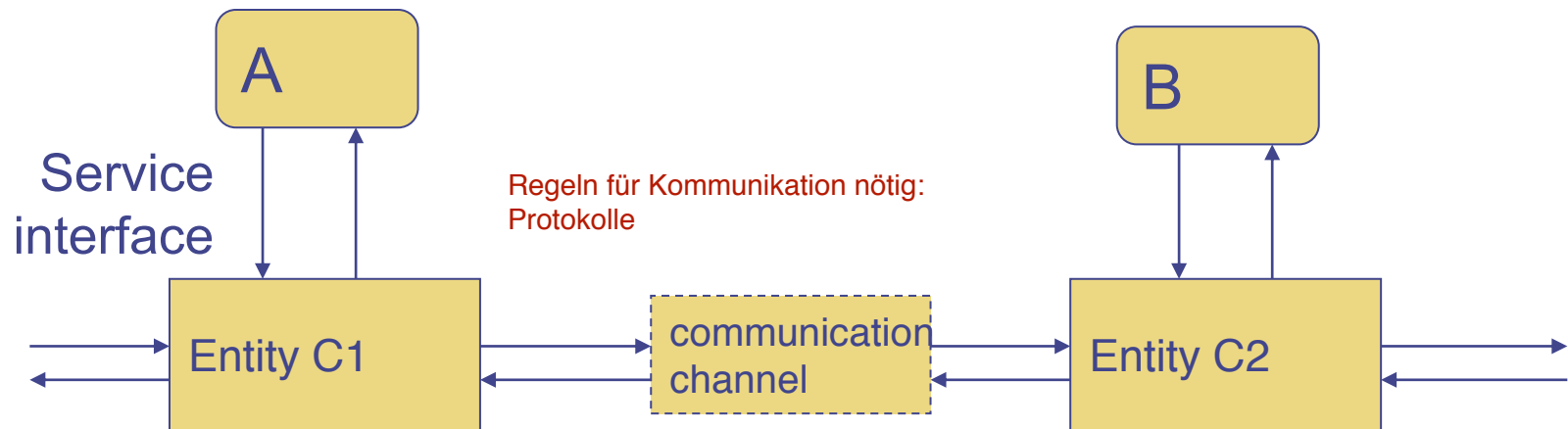
- Class *ServerSocket*
  - *socket constructor* (for listening at a server port)
  - *getInputStream, getOutputStream*
  - *DataInputStream, DataOutputStream*  
(automatic marshaling/unmarshaling)
  - *close* to close a socket  
(raises *UnknownHost, IOException*, etc)

# Reliability of sockets



# Communication: a service offered to the users...

- A, B use a **communication service** provided by a pair of Communicating Entities (short: **entities**)



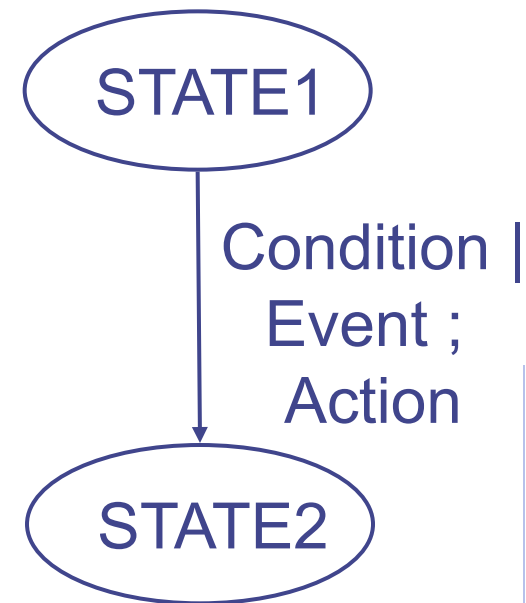
- Entities exchange messages
  - A message contains “**an envelope**” (a.k.a HEADER – an organization part)
  - **and –mostly - “the payload” (a.k.a user’s data!)**

# Protocols

- Cooperation between the entities needs rules
  - How can we ensure that each participant is able to communicate with each other participant?
    - Standardized, overreaching protocols after a general discussion open for everyone
    - Manufacturers must implement the protocol for their device, operating system, application, ...
- Transmission protocols = Rules for communication in network
- Analogy: Hand shake

# Protocol Specification

- Formal behavior
  - Rules which constitute the protocol have to be precisely specified
- Methode zur Beschreibung von Regeln des Protokolls:*
- One popular method: (Extended) Finite State Machine (FSM)
  - Protocol instance/protocol engine at each entity with several states
    - Connected – waiting...
  - Events/transitions between states
    - Message arrivals
    - Real time / timer events
  - Transition can have conditions
- Actions during transition are
  - Send new message
  - Set timer, delete timer, ...



# Protocol Specifications - FSMs

- Finite State Machines (FSM)

- $\Sigma = (S, E, D, S_0)$

- $S = \{s^j\}$ : countable state space Zustände

- $E = \{e^j\}$ : countable set of events Ereignisse

- $D$ : transition function

- $D: S \times E \rightarrow [S, A]$  Übergangsfunktion: Legt fest aus welchem Zustand bei welchem Ereignis zu welchem Zustand übergegangen wird

- $\phi$ : undefined transition (zero element)

- $A = \{a^j\}$ : countable set of actions

- $S_0$ : initial state

- $E^f(s)$ : feasible event set for state  $S$

**Just for  
those who  
like it a bit  
more  
formal!**



# Send\_and\_Wait Specification als FSM

- Example - Send and Wait

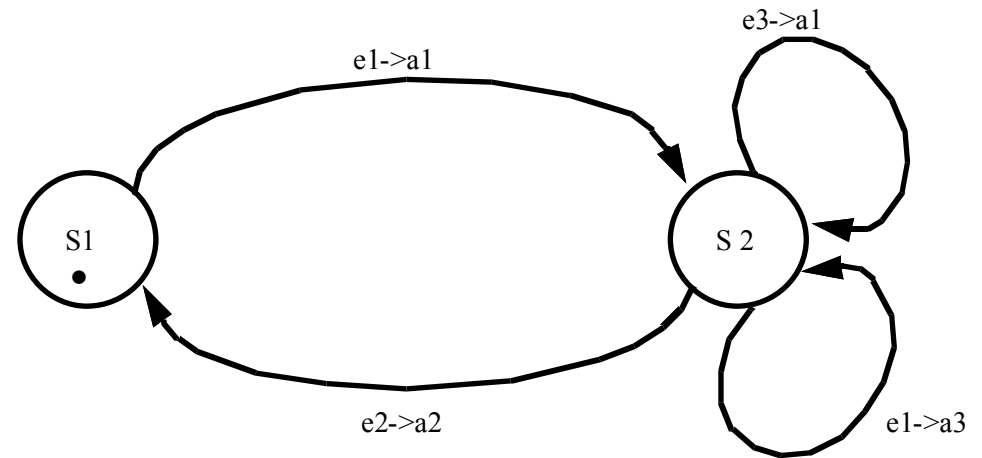
- $s^1$  - idle  $e^1$  - data to send ( **+Transmission Request**)
- $s^2$  - waiting  $e^2$  - get acknowledge
- $s^0 = s^1$   $e^3$  - timer expired
- $a^1 = \langle \text{send data, set timer} \rangle$
- $a^2 = \langle \text{acknowledge transfer, clear timer} \rangle$  (**Conf**)
- $a^3 = \langle \text{response: busy} \rangle$  ( **+Transmission Confirmation**)

Arbeiten mit Zustandstabelle beim Entwurf einer Spezifikation: alle möglichen Ausnahmen/Problemfälle werden gefunden und können berücksichtigt werden

- Transition function

Event	State	
	$s^1$	$s^2$
$e^1$ Ereignis $e^1$ tritt ein ->	$a^1, s^2$ führe Aktion $a^1$ aus und gehe zu Zustand $s^2$	$a^3, s^2$
$e^2$	0 nichts tun, bleibe im gleichen Zustand	$a^2, s^1$
$e^3$	0	$a^1, s^2$

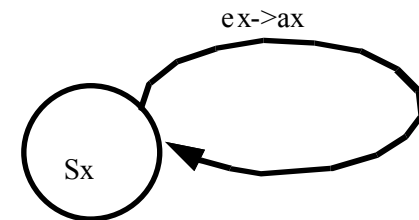
# Protocol Specification: Graph usage



## Receiver

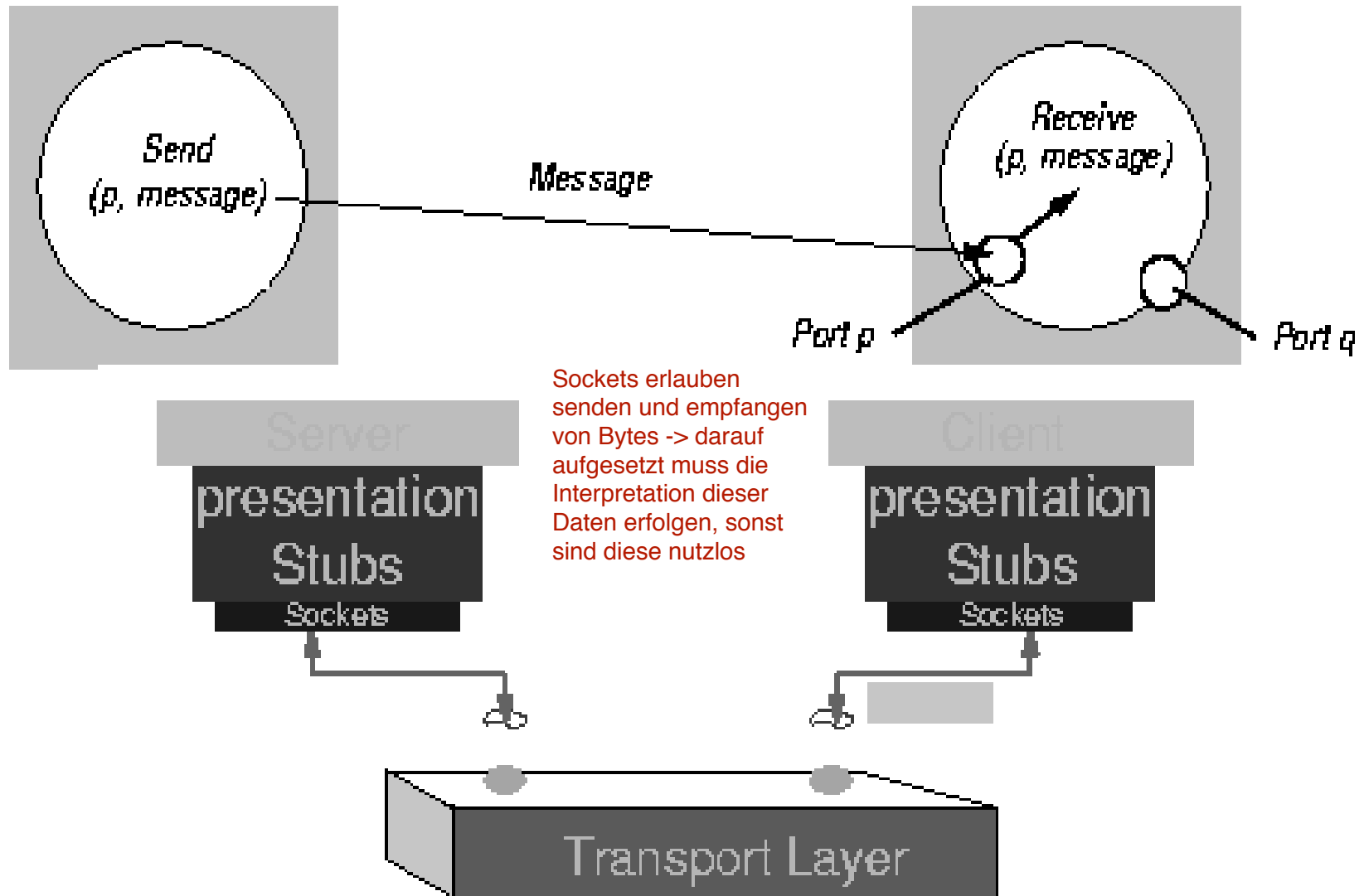
### Receiver side specification

- $S^x$  - idle
- $e^x$  - data arrived
- $a^x$  - acknowledge ( *+Transmission Indication* )
- Note:
  - Timers count is irrelevant
  - Infinite loop if permanent transmission errors!
  - Duplication if acknowledge gets lost



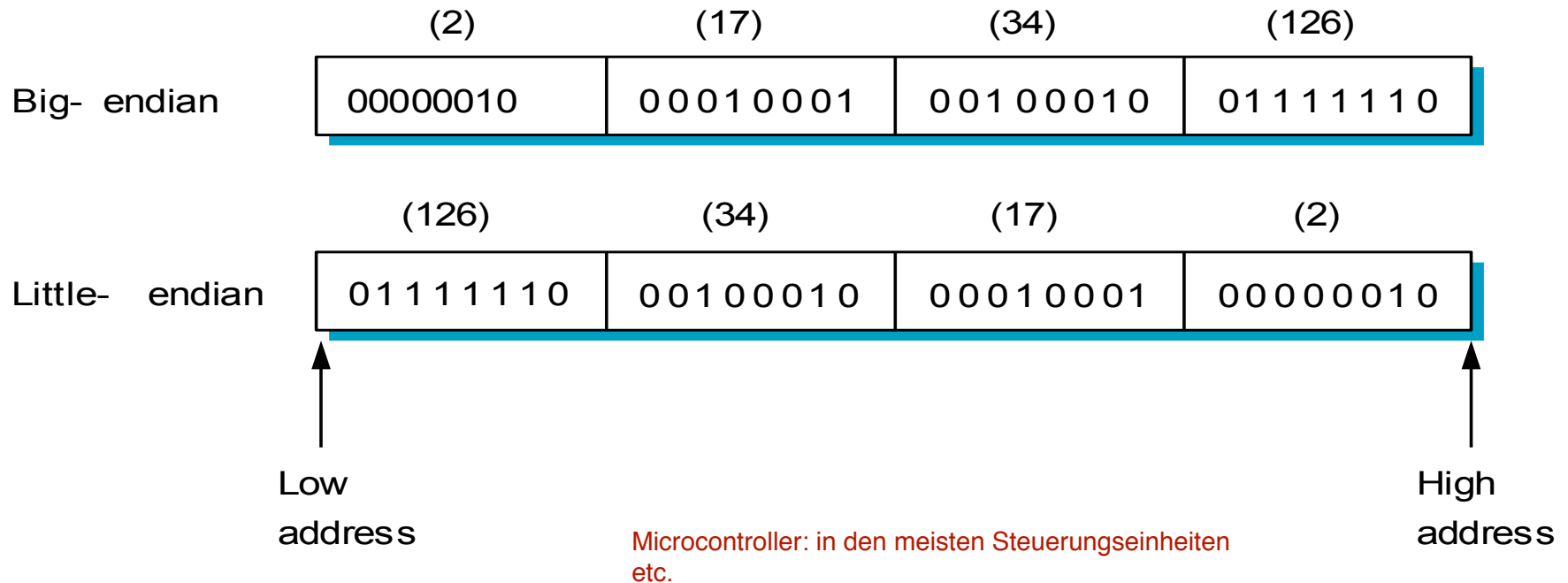
Streams of Bits/bytes can be transmitted: so what?  
How do we know what is the INFORMATION inside?

-> Interpretation der Bytes



# Simple example

- Representation of base types
  - floating point: IEEE 754 versus non-standard
  - integer: big-endian versus little-endian (e.g., 34,677,374)

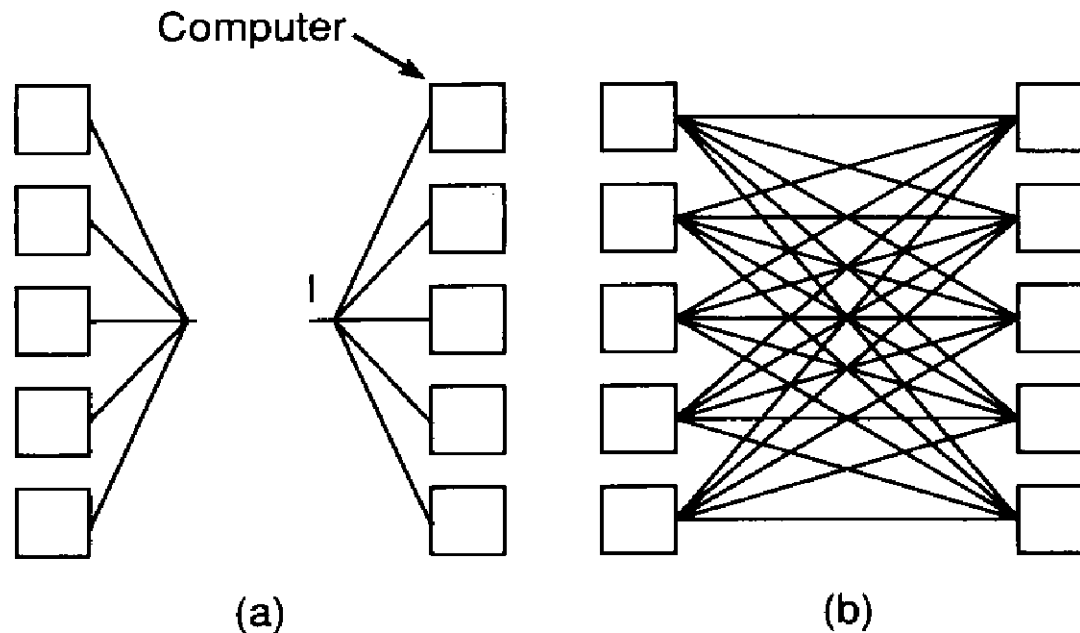


- on a 680x0 CPU, the 32 bit integer number 255 is stored as:  
00000000 00000000 00000000 11111111  
but an Intel 80x86-CPU stores this as: in den meisten PCs  
11111111 00000000 00000000 00000000

# Taxonomy

- Data types

- base types (e.g., ints, floats); must convert
- flat types (e.g., structures, arrays); must pack
- complex types (e.g., pointers); must linearize



- Conversion Strategy

a) – canonical intermediate form

gemeinsamer Standard, alle Sprachen werden nur in den Standard übersetzt (Aufwand nur  $n$  statt  $n \cdot n$ )

b) – receiver-makes-right (an  $N \times N$  solution)

Jede Sprache in jede andere Übersetzen -> großer Aufwand

# Data Conversion

- Two different types of rules are needed:
  - Abstract syntax: a station must define what datatypes are to be transmitted
  - Transfer syntax: it must be defined how these datatypes are transmitted, i.e. which representation has to be used.

Tagged versus untagged data

type = INT	len = 4		value =	417892	
---------------	---------	--	---------	--------	--

Übertragung von Datentypen, nicht nur reinen  
Daten

# Abstract Syntax Notation.1 - ASN.1

- Each transmitted data value belongs to an associated data type.
- For the lower layers of the OSI-RM, only a fixed set of data types is needed (frame formats), for applications with their complex data types ASN.1 provides rules for the definition and usage of data types.
- ASN.1 distinguishes between a data type (as the set of all possible values of this type) and values of this type (e.g. '1' is a value of data type Integer).
- Basic ideas of ASN.1:
  - Every data type has a globally unique name (type identifier)
  - Every data type is stored in a library with its name and a description of its structure (written in ASN.1)
  - A value is transmitted with its type identifier and some additional information (e.g. length of a string).

Einzigtiger Identifier für jeden Datentypen: Jeder Wert wird mit dem Identifier, dem Inhalt und eventuell nötigen Informationen (Länge) versendet



# Definition of Datatypes using ASN.1 (1)

- A data type definition is called „abstract syntax“; it uses a Pascal-like syntax.
- Lexical rules:
  - Lowercase letters and uppercase letters are different
  - A type identifier must start with an uppercase letter
  - Keywords are written in uppercase letters
- ASN.1 offers some predefined simple types:
  - BOOLEAN (Values: True, False)
  - INTEGER (natural numbers without upper bound)
  - ENUMERATED (association between identifier and Integer value)
  - REAL (floating point values without upper or lower bound)
  - BIT STRING (unbounded sequence of bits)
  - OCTET STRING (unbounded sequence of bytes/ octets)
  - NULL (special value denoting absence of a value)
  - OBJECT IDENTIFIER (denoting type names or other ASN.1-objects)

Syntax: bestimmt in welcher Reihenfolge und in welchem Format die Informationen versendet werden

# Definition of Datatypes using ASN.1 (2)

- Examples:
  - MonthsPerYear ::= INTEGER  
MonthsPerYear ::= INTEGER (1..12)  
Answer ::= ENUMERATED (correct(0), wrong(1),noAnswer(3))
- With the following type constructors new types can be built from existing ones:
  - SET: the order of transmission of the elements of a set is not specified. The number of elements is unbounded, their types can differ
  - SET OF: like SET, but all elements are of the same type.
  - SEQUENCE: the elements of a sequence are transmitted in the defined order. They can be of different types. The number of elements is unbounded.
  - SEQUENCE OF: like SEQUENCE, but all elements are of the same type
  - CHOICE: the type of a given value is chosen from a list of types (like a Pascal variant record)
  - ANY: unspecified type

# ASN.1 Transfer Syntax (1)

- Some coding rules (the „transfer syntax“) specifies how a value of a given type is transmitted. A value to be transmitted is coded in four parts:
  - identification (type field or tags)
  - length of data field in bytes
  - data field
  - termination flag, if length is unknown.
- The coding of data depends on their type:
  - integer numbers are transmitted in High-Endian Two's complement representation, using the minimal number of bytes: numbers smaller 128 are encoded in one byte, numbers smaller than 32767 are encoded in two bytes, ...
  - Booleans: 0 is false, every value not equal 0 is true.
  - for a sequence type first a type identification of the sequence itself is transmitted, followed by each member of the sequence.
  - Similar rules apply to the transfer of set types