Introduction to Communication Networks and Distributed Systems

Unit 2 - Reference Models and Inter Process Communications -

Fachgebiet Telekommunikationsnetze
Prof. Dr. A. Zubow
zubow@tkn.tu-berlin.de

Technische Universität Berlin

The Issue of Network Software

- First computer networks were designed with hardware in focus & software as some add-on
- Over time the complexity of software increased dramatically making **software structuring** essential:
  - Most networks are organized as a **stack of layers** or levels
  - Each layer offers a certain service to higher layers
  - Divide & conquer approach known from Software Engineering (OO/ADT)

E.g. Linux kernel 5.3 networking subsystem (./net)

Language	Files	Lines	Code	Comments	Blanks
ASN.1	1	177	125	1	51
Assembly	1	7	7	0	0
C	1366	1017006	746104	113980	156922
C Header	234	45255	31101	7694	6460
HEX	1	86	86	0	0
Makefile	86	1805	1173	351	281
Total	1689	1064336	778596	122026	163714

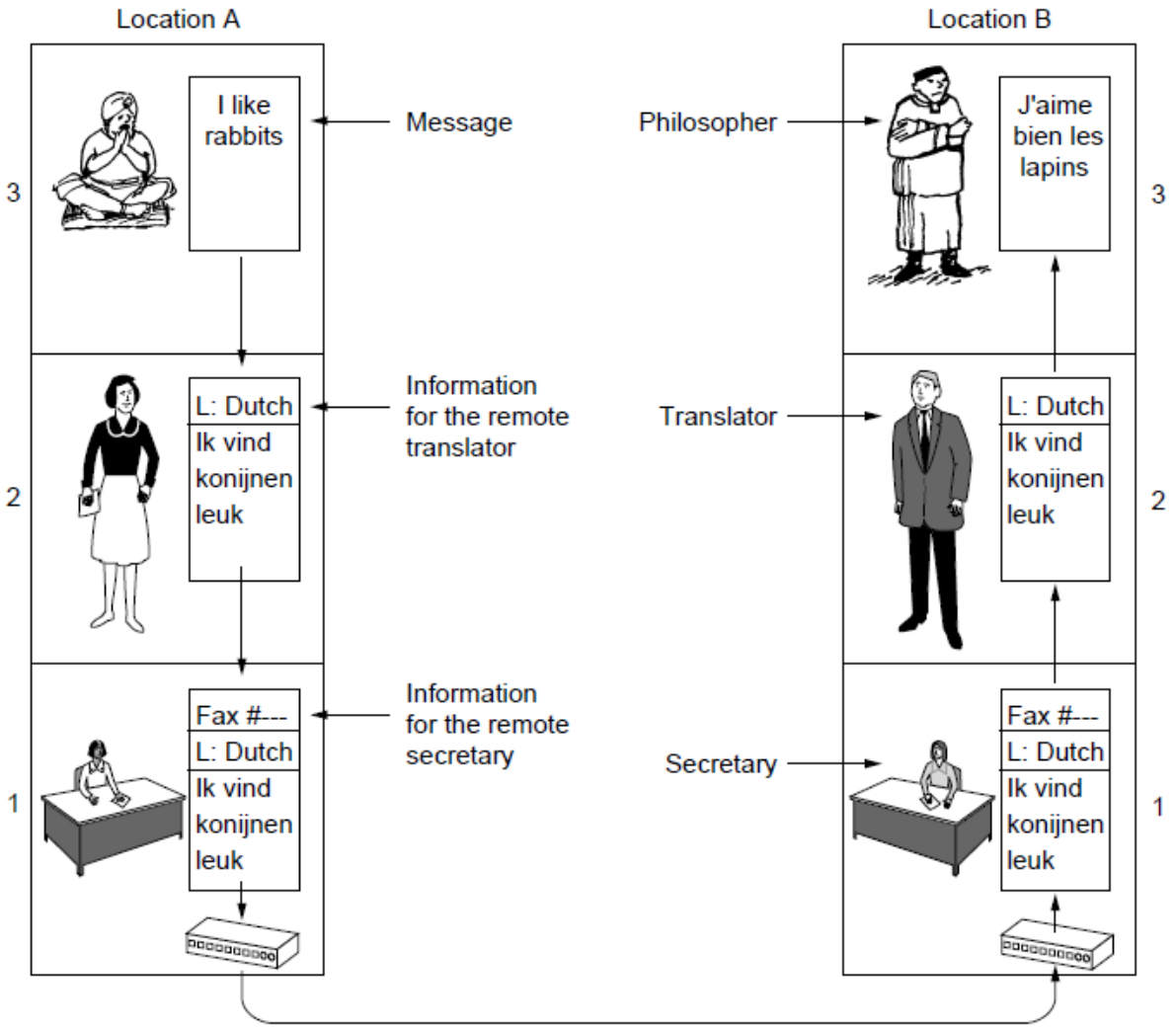
~ 1 million lines of C code

Analogy: The philosopher-translator-secretary architecture

Philosophers

Assistants

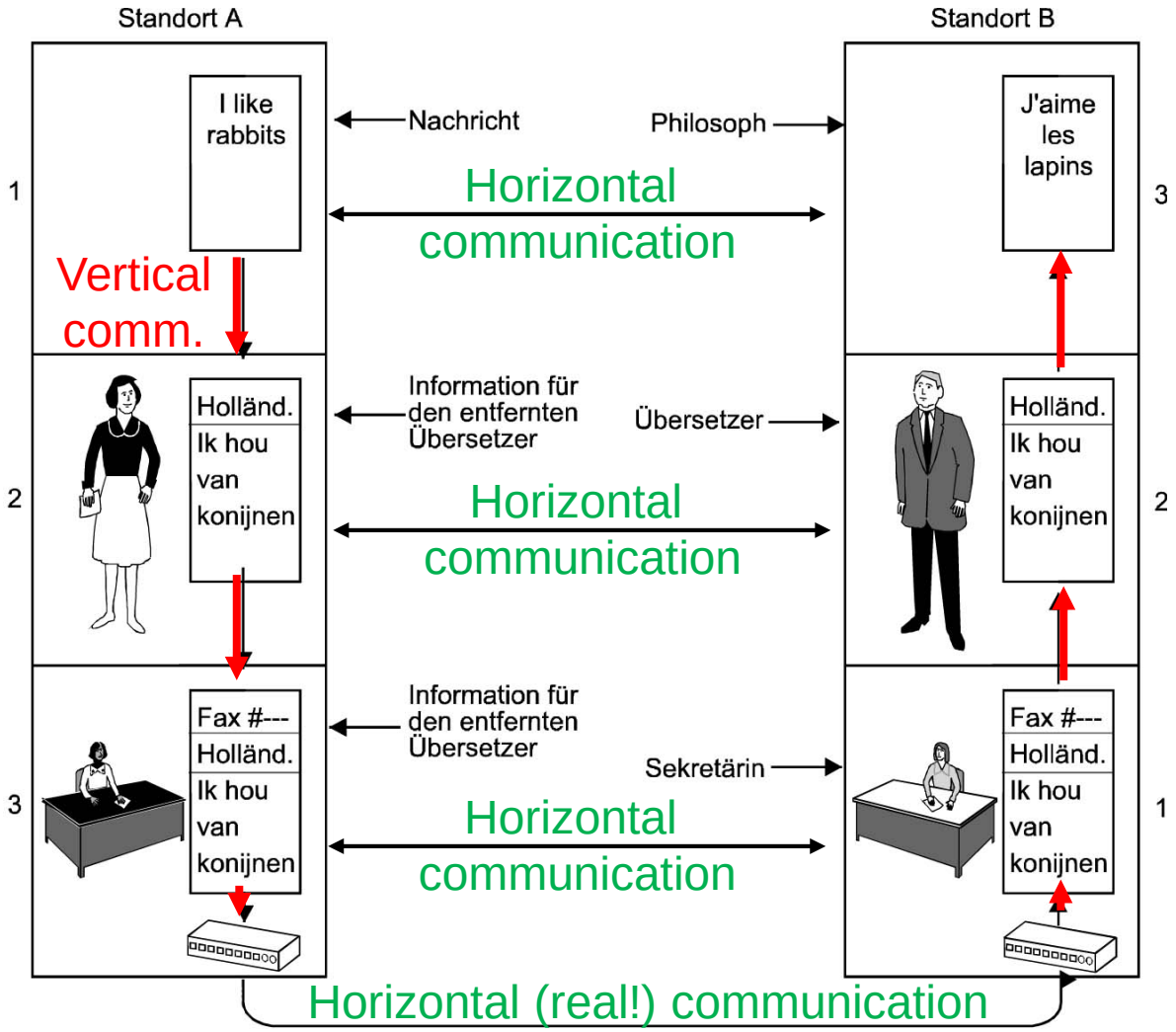
Office clerks



[Tanenbaum., op. cit.]

Analogy: Nested Layers as nested Translations

- Vertical vs. horizontal communication
  - Vertical:
    always real
  - Horizontal:
    may be real or virtual
- **Note**: protocols interchangeable as long as the interface remains unchanged, e.g.:
  - layer 2: Dutch => Russian
  - Layer 3: Fax => E-Mail

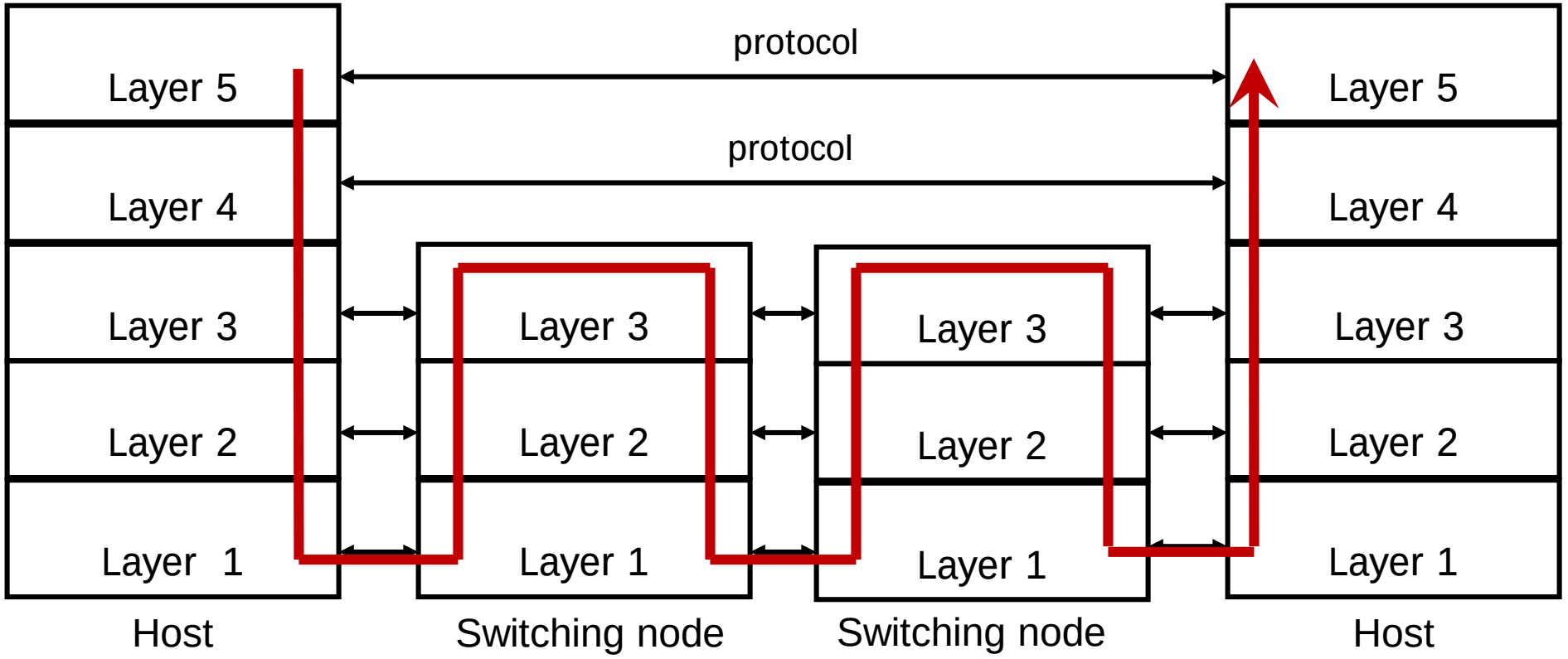


The Reference Model

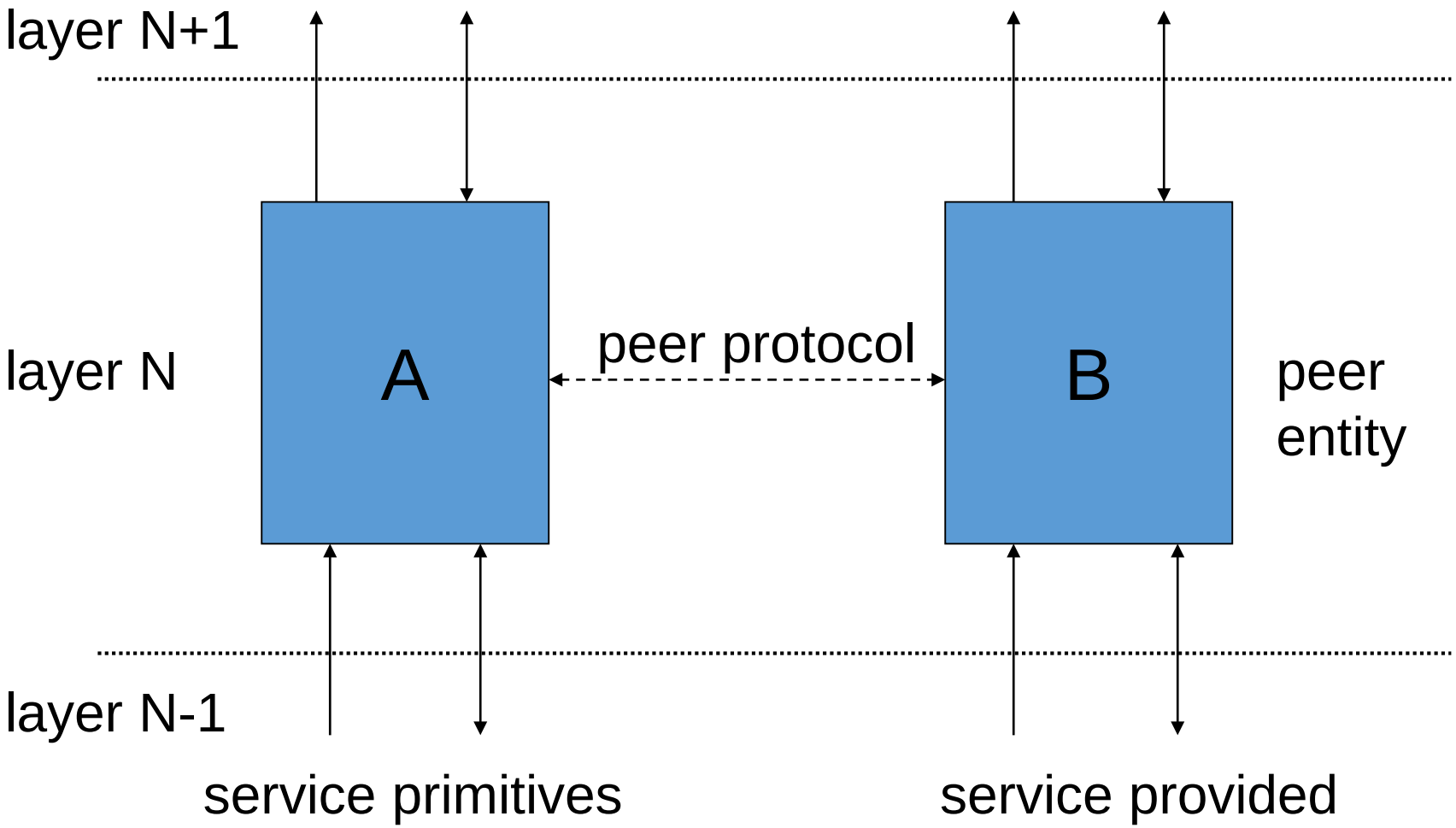
- To keep **complexity** of communication systems tractable:
  - Division in subsystems with clearly assigned responsibilities – layering
- Each layer offers a particular **service**
  - More abstract and more powerful the higher up in the layering hierarchy
- To provide a service, a layer has to be **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** (alternative: cross-layering)

Multi-layer Architecture

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

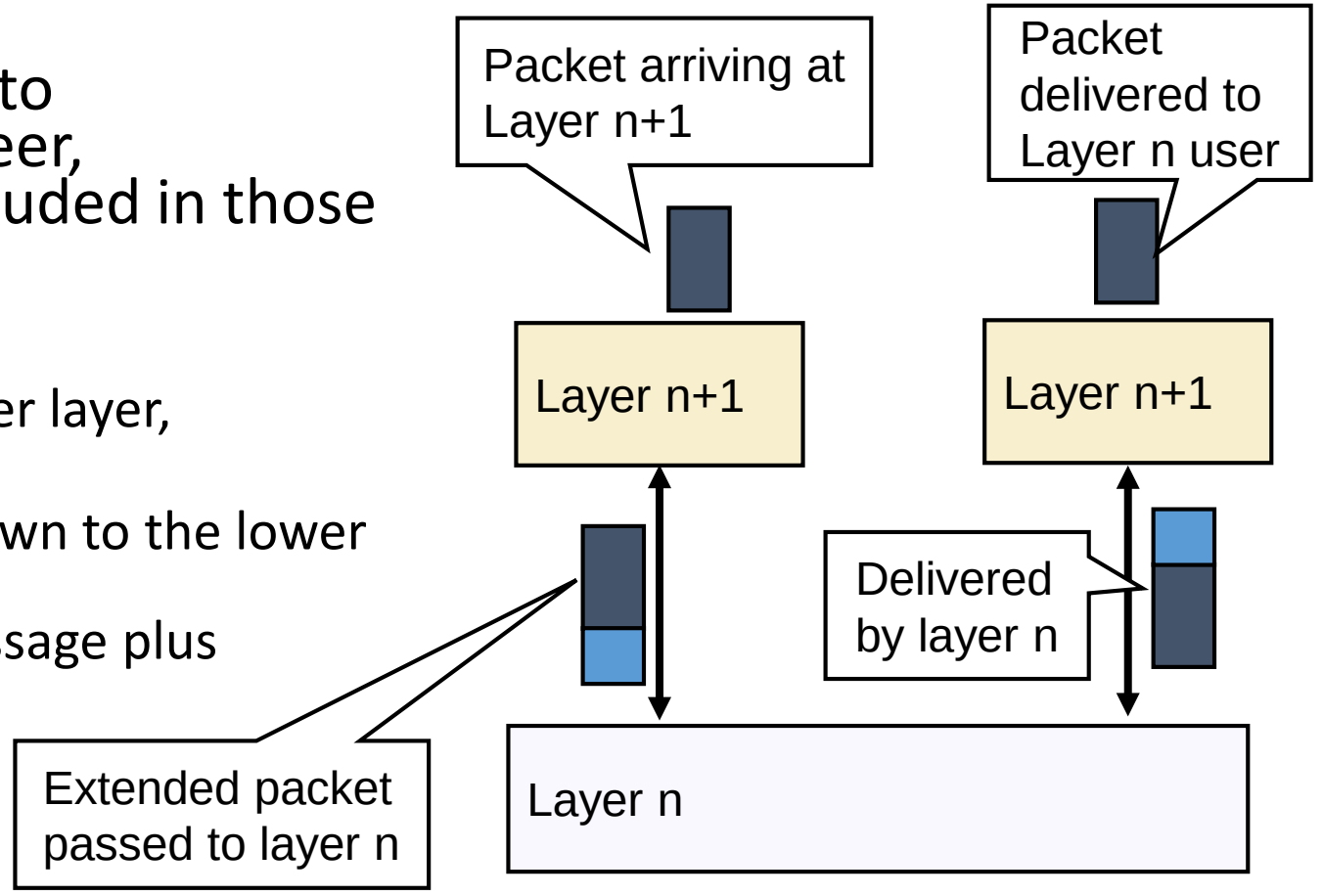


Multi-layer Architecture (II)



Protocols and Messages

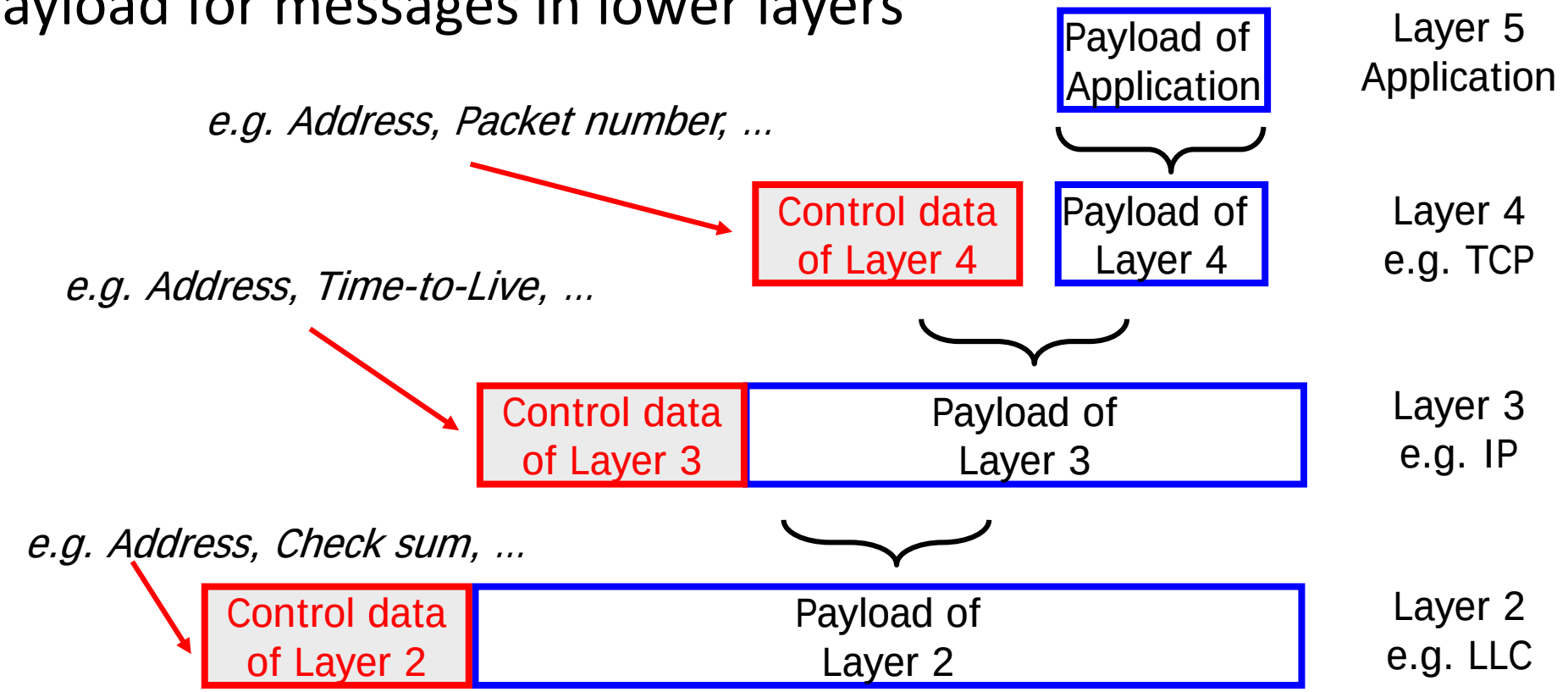
- When using lower-layer services to communicate with the remote peer, administrative data is usually included in those messages
- Typical example
  1. Protocol receives data from higher layer,
  2. Adds own administrative data,
  3. Passes the extended message down to the lower layer,
  4. Receiver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



Embedding Messages

src: wiki

- 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)

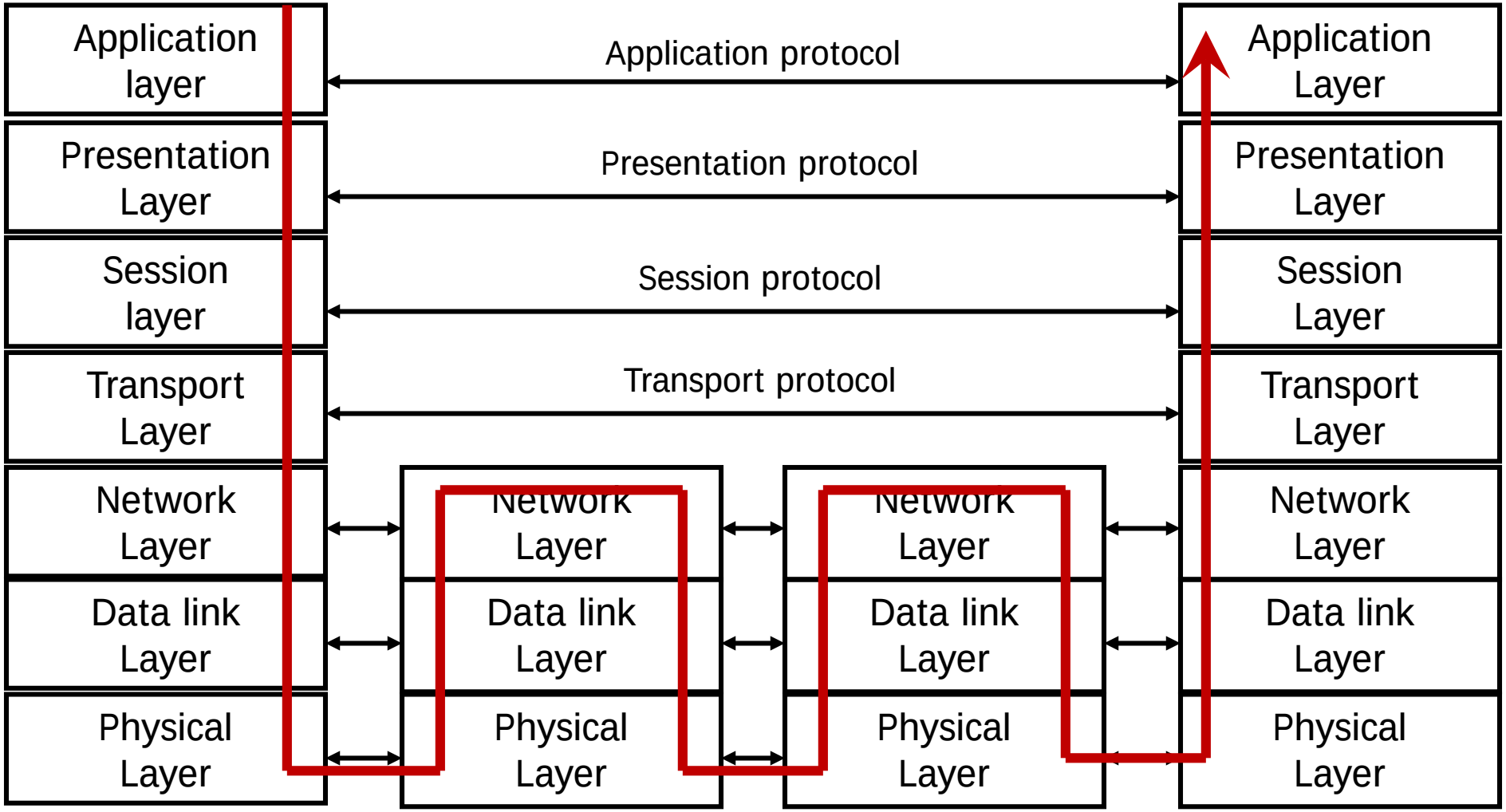
Standardization

- 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** – defines the layers 1 & 2 (ISO/OSI reference model)
- **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
- More information: Tanenbaum, chapter 1.4

ISO/OSI Model

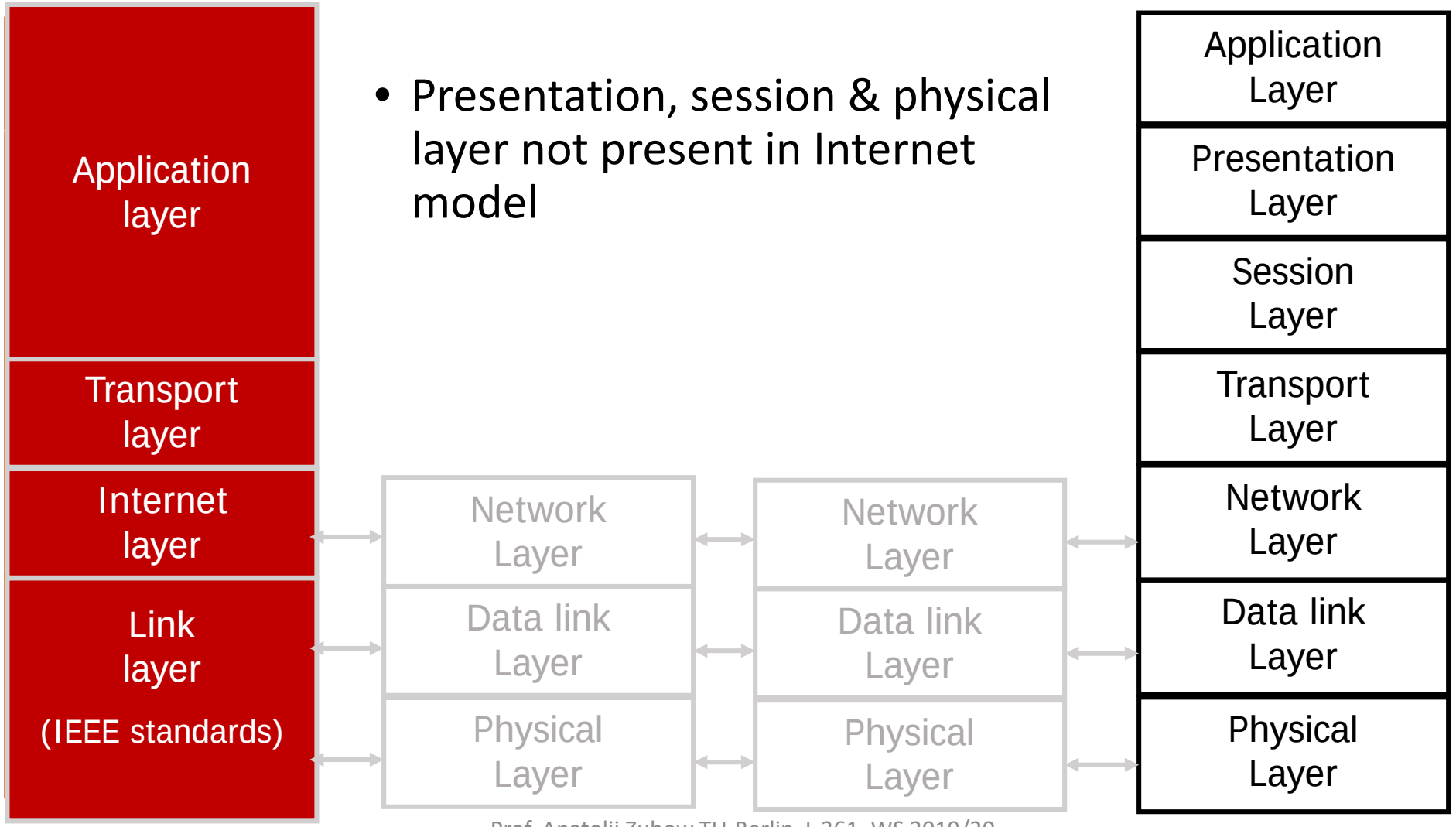


Brief Overview of the 7 Layers

- **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 (in red) vs. ISO/ OSI



- Presentation, session & physical layer not present in Internet model

Architecture, 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 Internet reference model
  - Data link layer: Ethernet & CSMA/CD (defined in IEEE standard)
  - 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

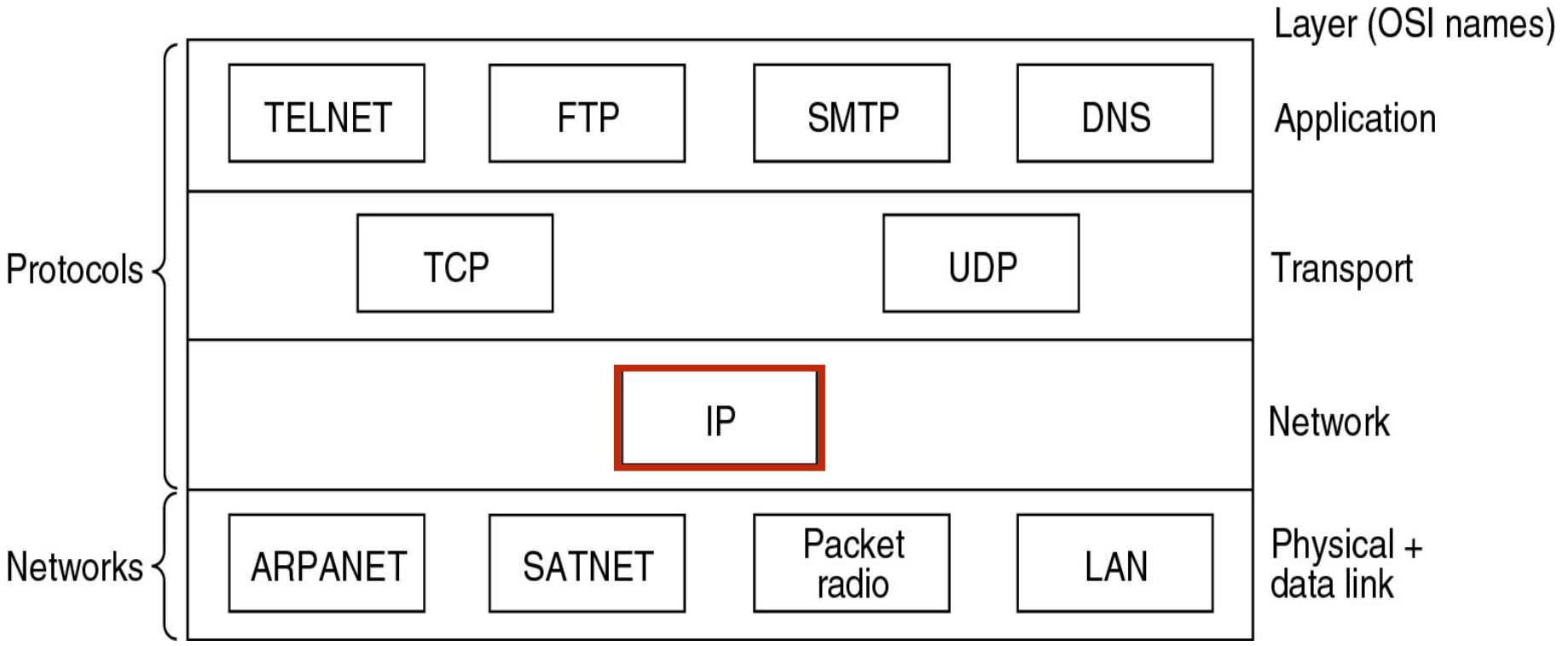
  → **Anything over Internet**
- New applications do not need any changes in the network
  - Compare with the telephone network

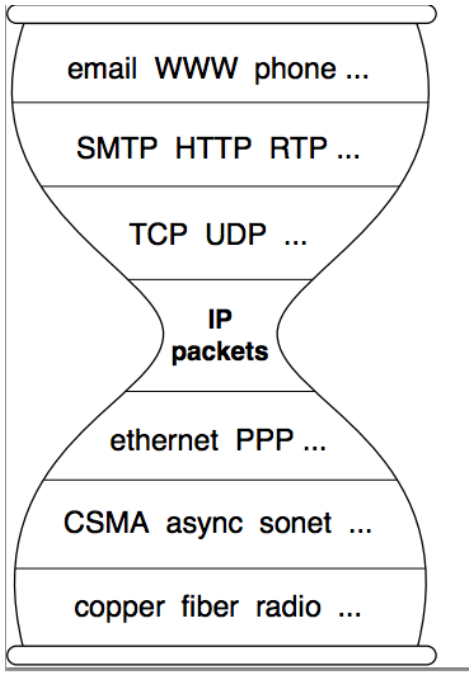
The Internet Suite of Protocols

- Over time, suite of protocols evolved around core TCP/IP protocols

→Internet Protocol Suite is also refereed to as TCP/IP Protocol Suite

→“**hourglass model**”: thin waist of the protocol stack at IP, above technological layers





Naming & Addressing in the Internet Stack

- **Names**: Data to **identify** an entity exist on different levels
  - Alphanumerical names for resources: e.g. saturn.tkn.tu-berlin.de, www.tkn.tu-berlin.de
- **Address**: Data how/where to **find** an entity
  1. 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)
  2. 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 (LAN): a MAC address
  - 48 bits, hexadecimal notation, example: 08:00:20:ae:fd:7e

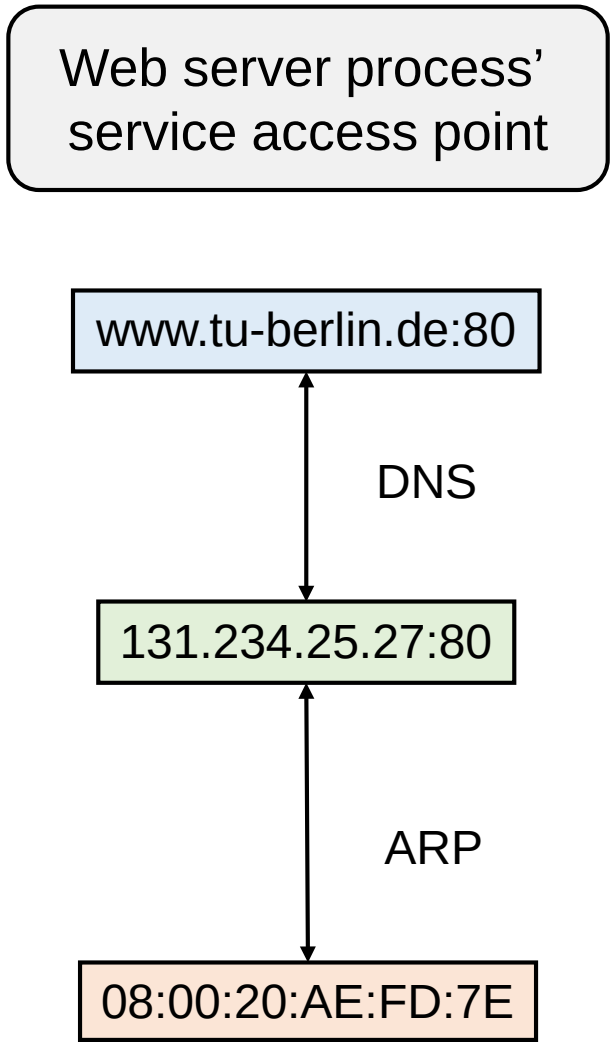
Mapping

- Needed: mapping from name to address

→ realized by separate protocols

- From alphanumerical name to IP address: **Domain Name System** (DNS)

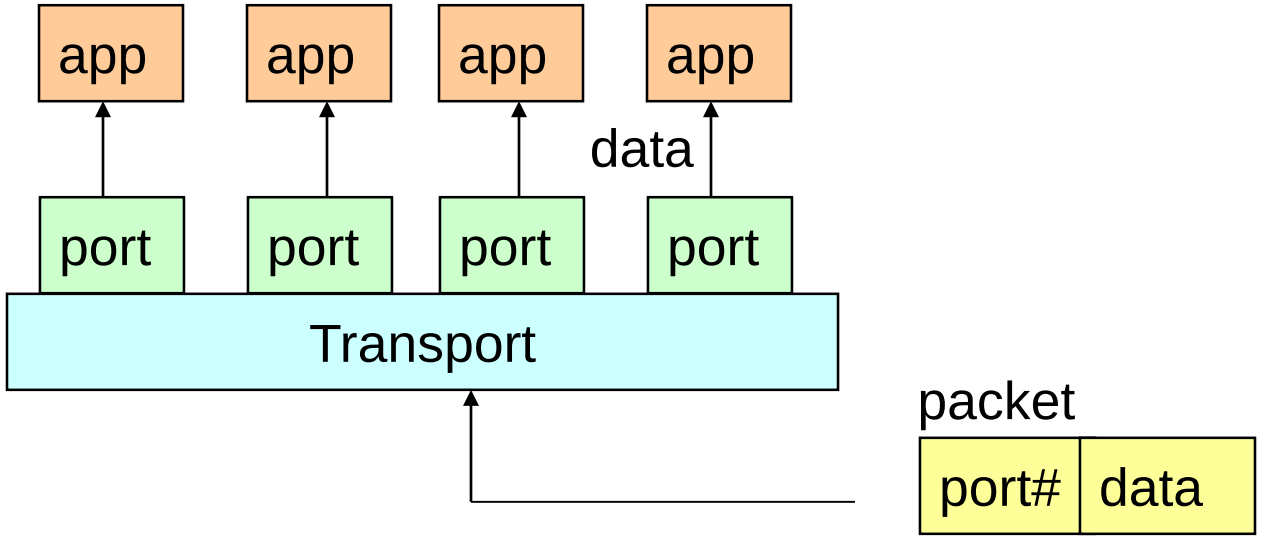
- Often also needed: mapping from IP address to MAC address: **Address Resolution Protocol** (ARP)



Understanding Ports

[Buya, op. cit.]

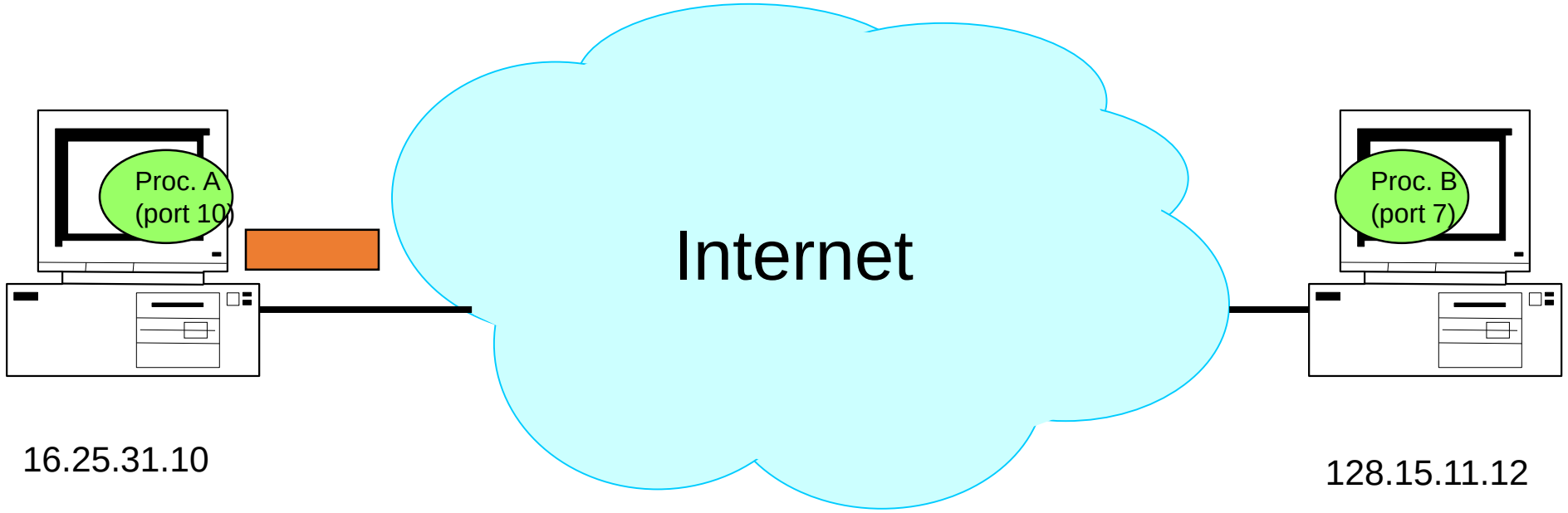
- … to distinguish between individual processes
- Port is represented by a positive (16-bit) integer value
- Some ports have been reserved to support common/well known services: http 80/tcp; ftp 21/tcp; telnet 23/tcp; smtp 25/tcp;
- User level process/services generally use port number value >= 1024



Internet End-to-End View

[Stoica, op. cit.]

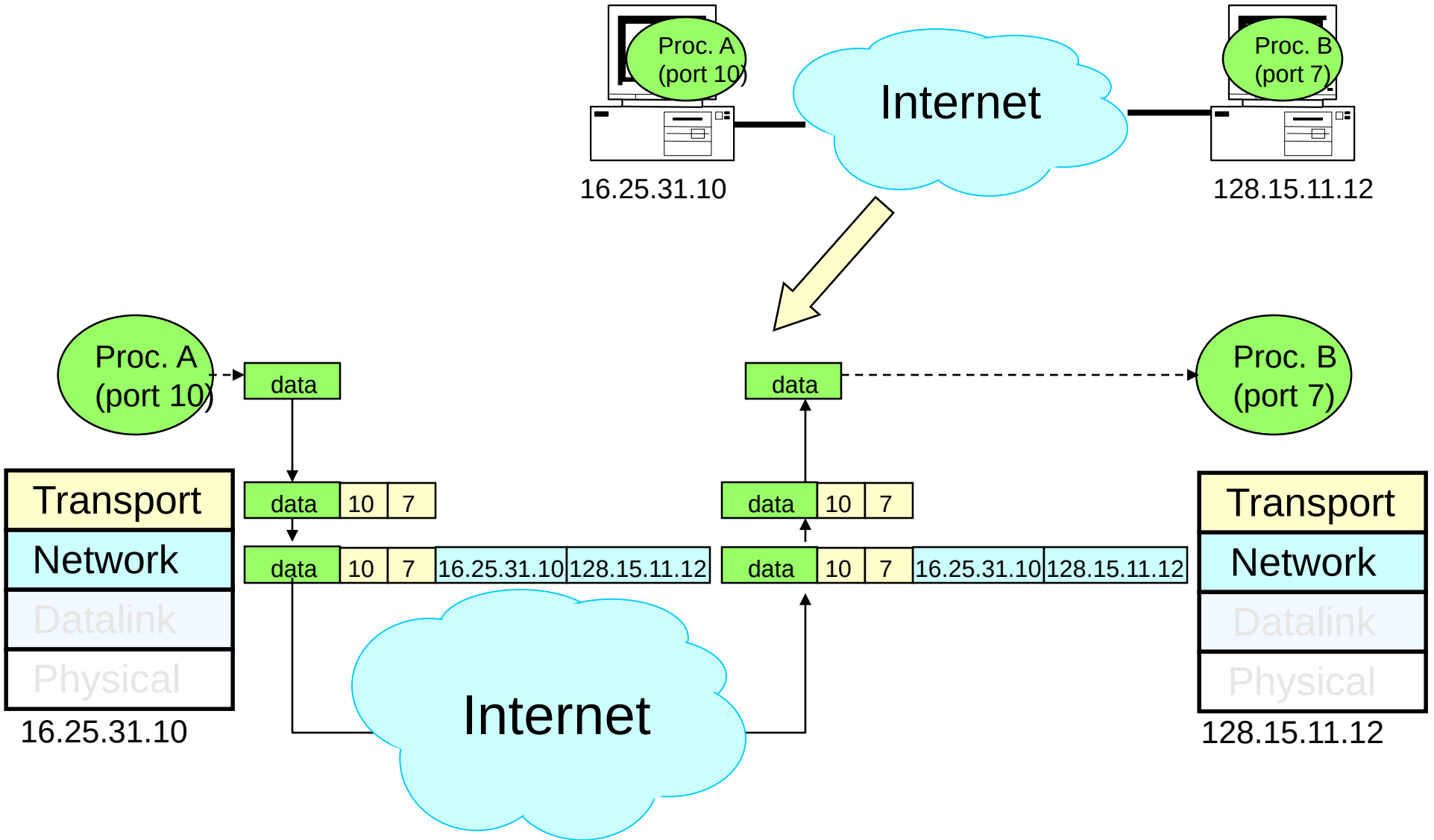
- Process A sends a packet to process B



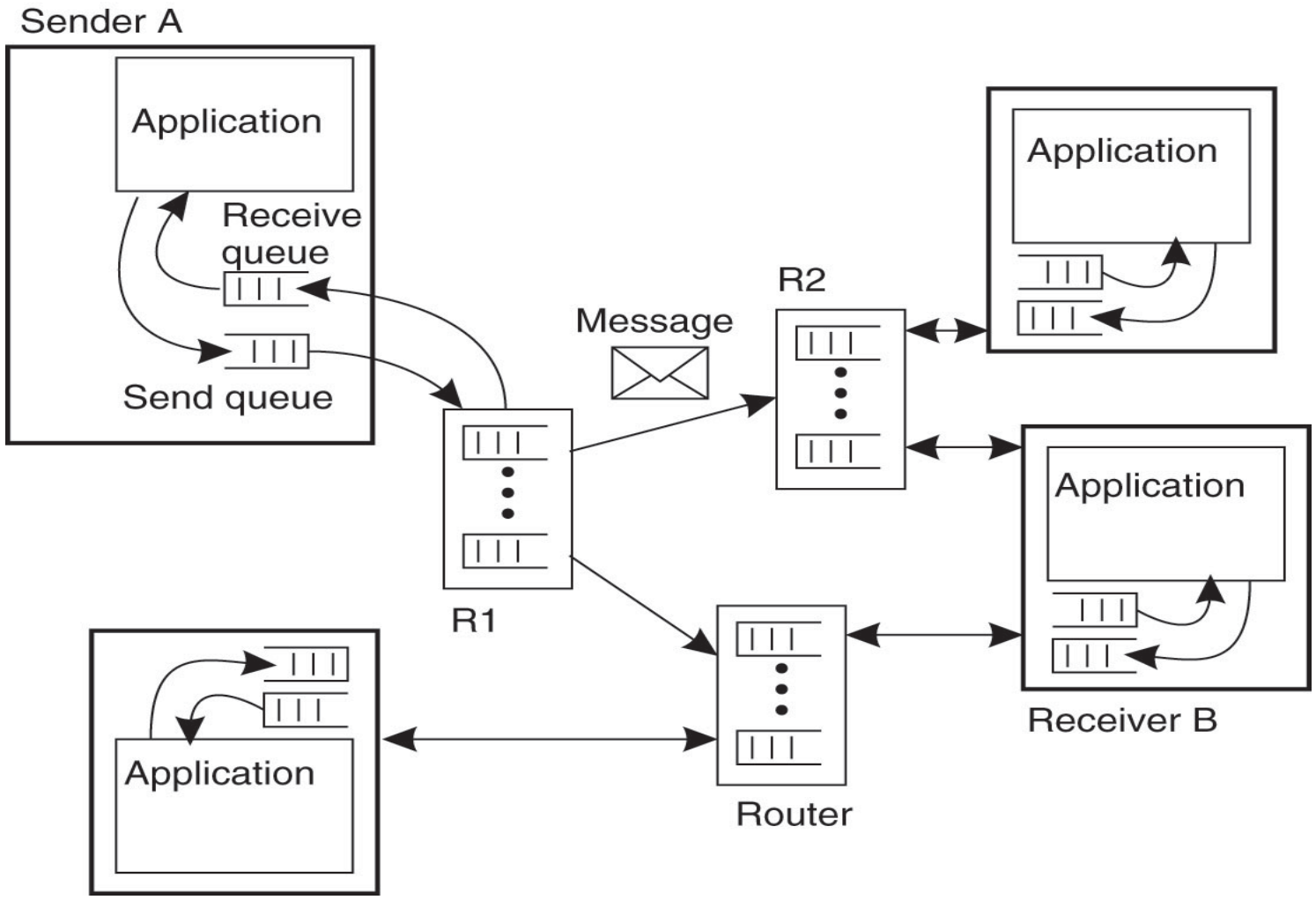
IP address:
A four-part “number” used by Network Layer to route a packet from one computer to another

End-to-End Layering View

[Stoica, op. cit.]



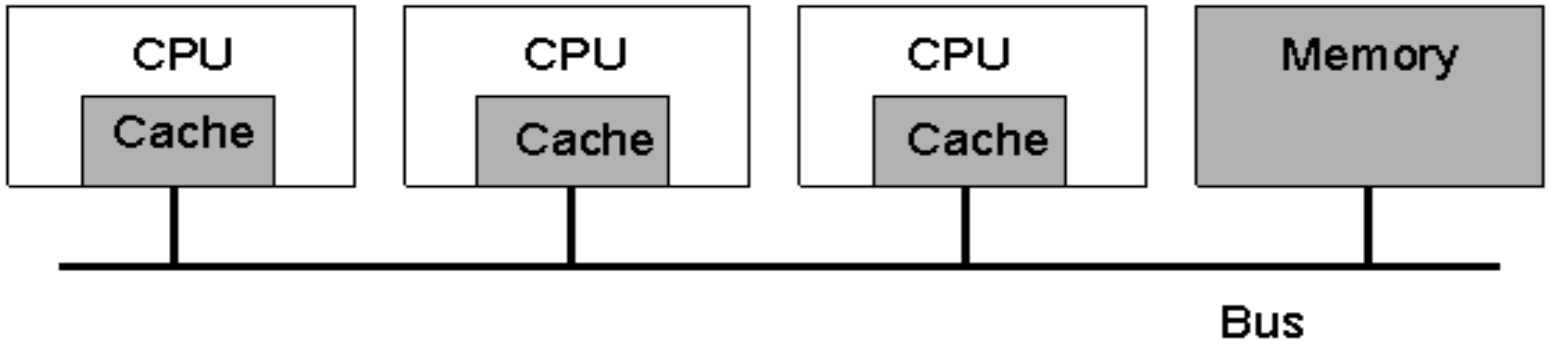
General Architecture of a Message Passing System



[Tanenbaum, op. cit.]

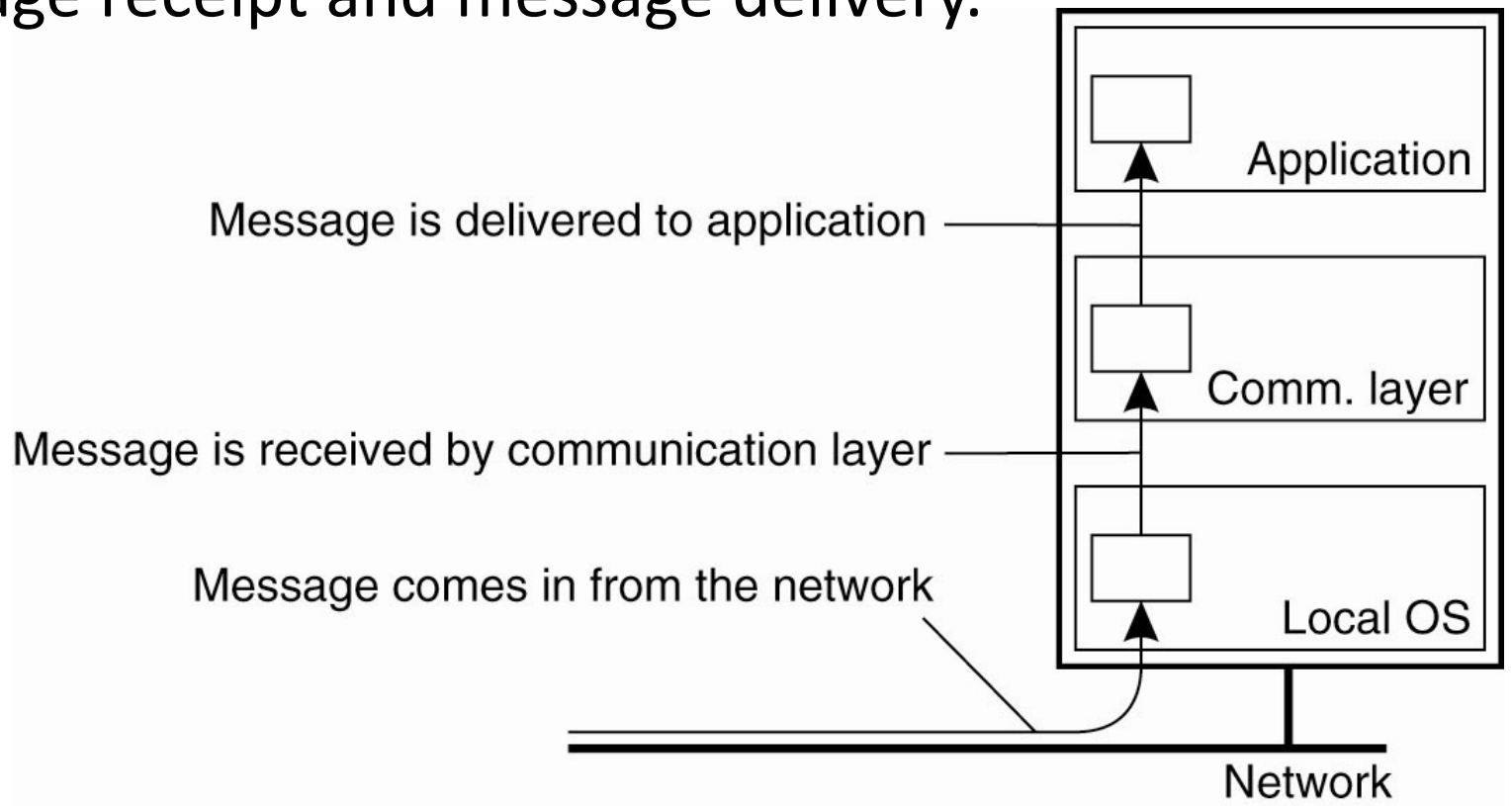
In contrast to a Multiprocessor

- A bus-based multiprocessor with typical features:
  - Physical access to a common memory
  - Resources under same management
  - Very fast (bus, switching matrix) based local communication



Message Receipt vs. Message Delivery [Tanenbaum, op. cit.]

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

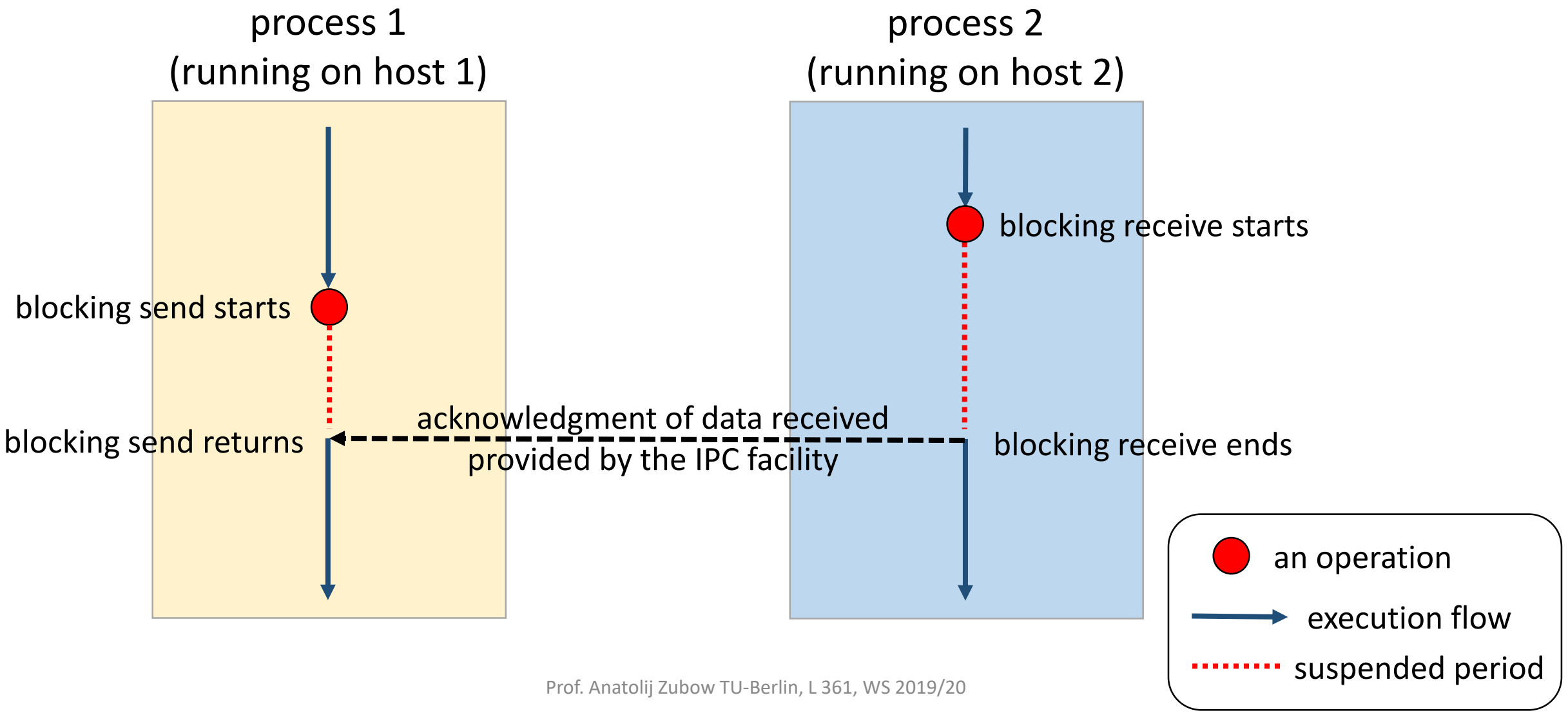


Interaction Principles: Synchronous Interaction

- Blocking send
  - Blocks until message is transmitted
  - Blocks until message acknowledged
- Blocking receive
  - Waits for message to be received

- **You should known**: upper/lower bounds on execution speeds, message transmission delays and clock drift rates

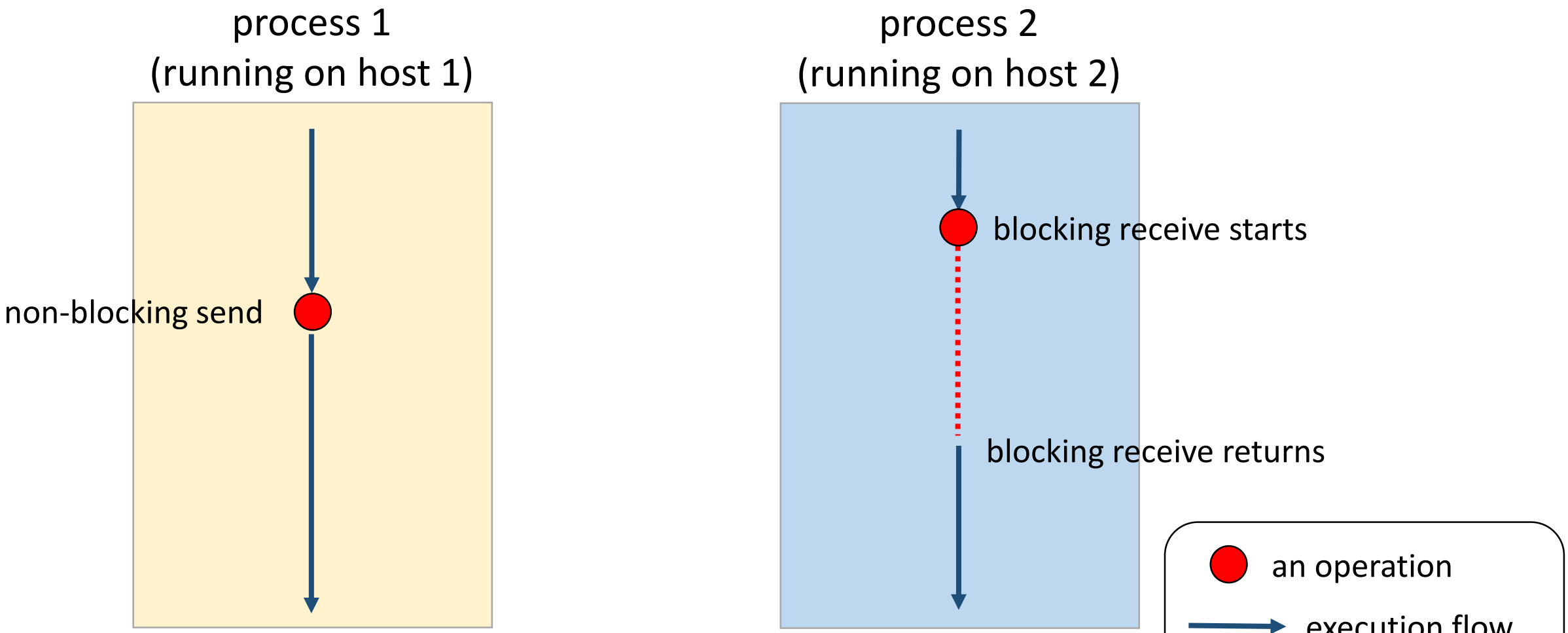
Synchronous Send & Receive



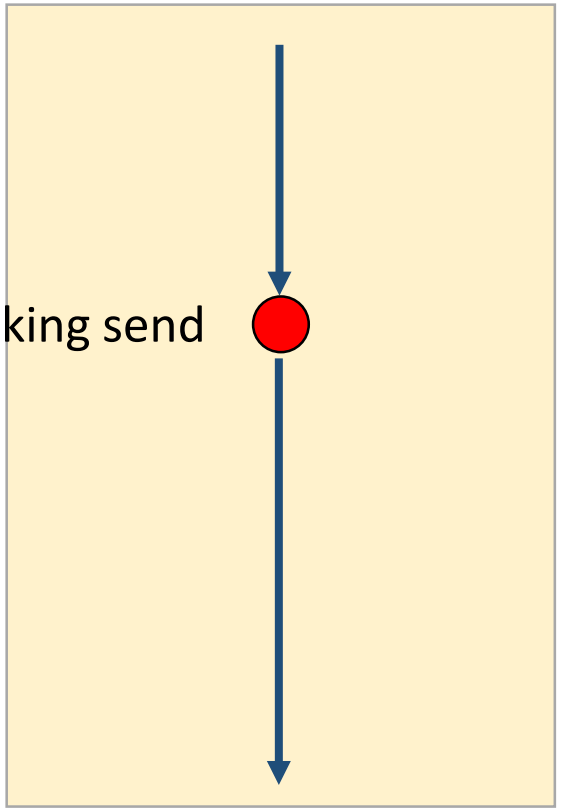
Interaction Principles: Asynchronous Interaction

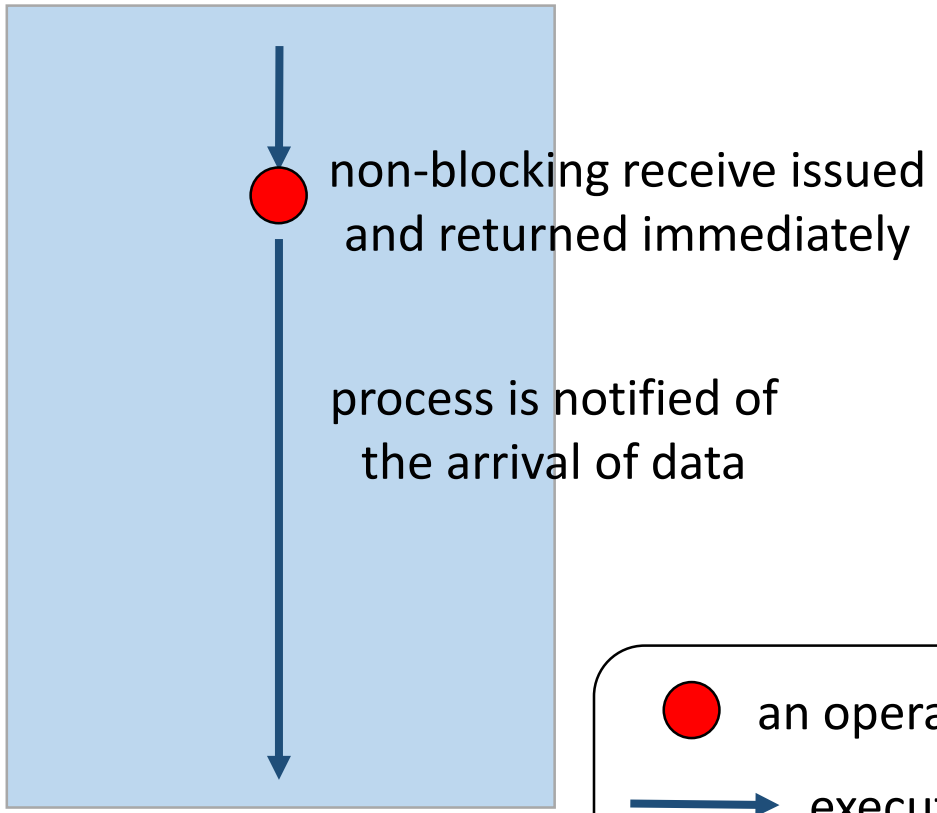
- Non-blocking send: sending process continues as soon message is **queued**
- Blocking or non-blocking receive:
  - Blocking:
    - Timeout
    - Threads
  - Non-blocking: proceeds while waiting for message
    - Message is queued upon arrival
    - Process needs to poll or be interrupted
- **Advantage**: arbitrary process execution speeds, message transmission delays and clock drift rates
- Some problems impossible to solve (e.g. agreement – see Two Army problem)

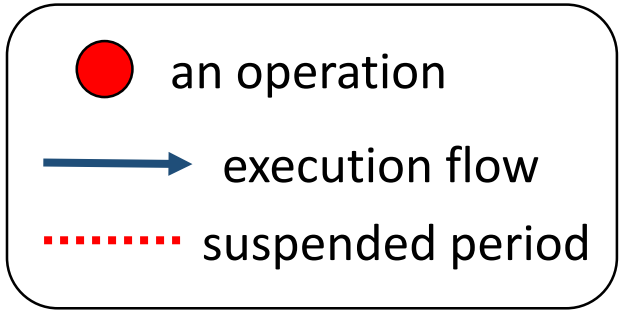
Asynchronous Send & Synchronous Receive



Asynchronous Send & Asynchronous Receive

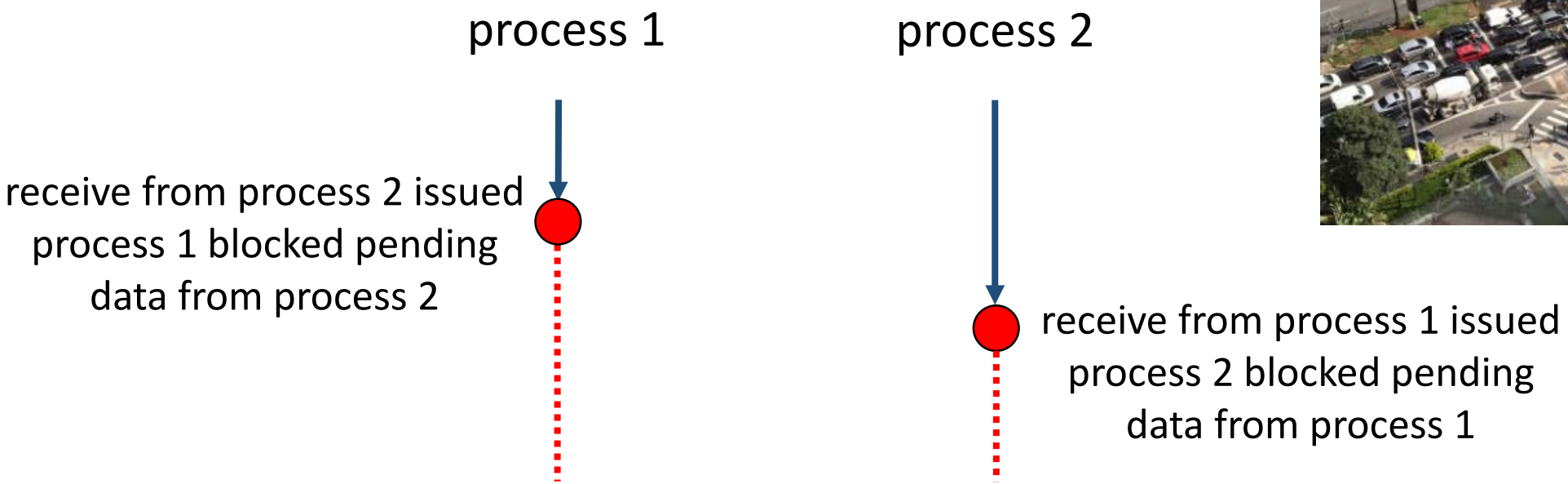






Blocking, Deadlock, and Timeouts

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



Intuitive Message-Passing Primitives [Tanenbaum, op. cit.]

… there is a great variety …

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_isend	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 are none
MPI_irecv	Check if there is an incoming message, but do not block

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

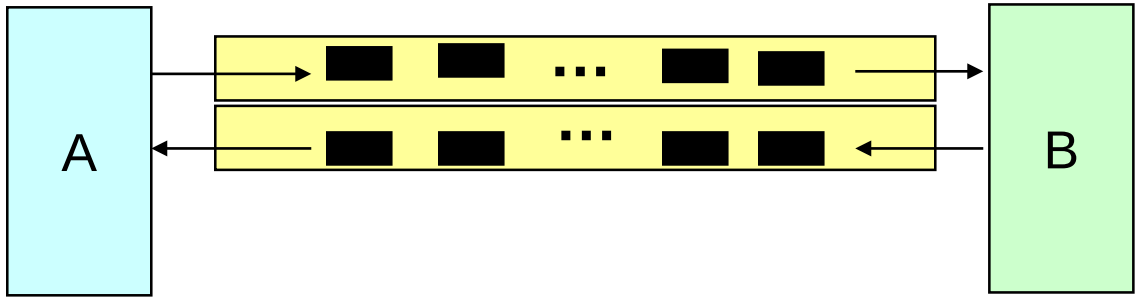
  → **connection-less service**
- **Note**: Both are possible on top of either circuit or packet switching

Two Communication Models

[Buya, op. cit.]

Connection-Oriented Communication

- ***Reliable byte stream*** *(connection oriented)*
  - Byte stream (or word stream, or X stream)
  - Correct, complete, in order, confirmed:
    - Processes have a guarantee that messages will be delivered (sender can check that!)
    - It is possible to build reliability atop unreliable service, e.g. TCP atop IP
  - Sometimes, but not always secure/dependable



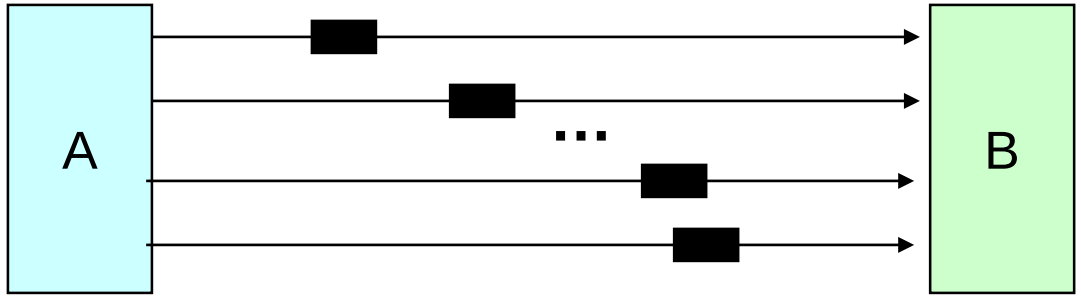
Connection-oriented Services

- 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

Two Communication Models

[Buya, op. cit.]

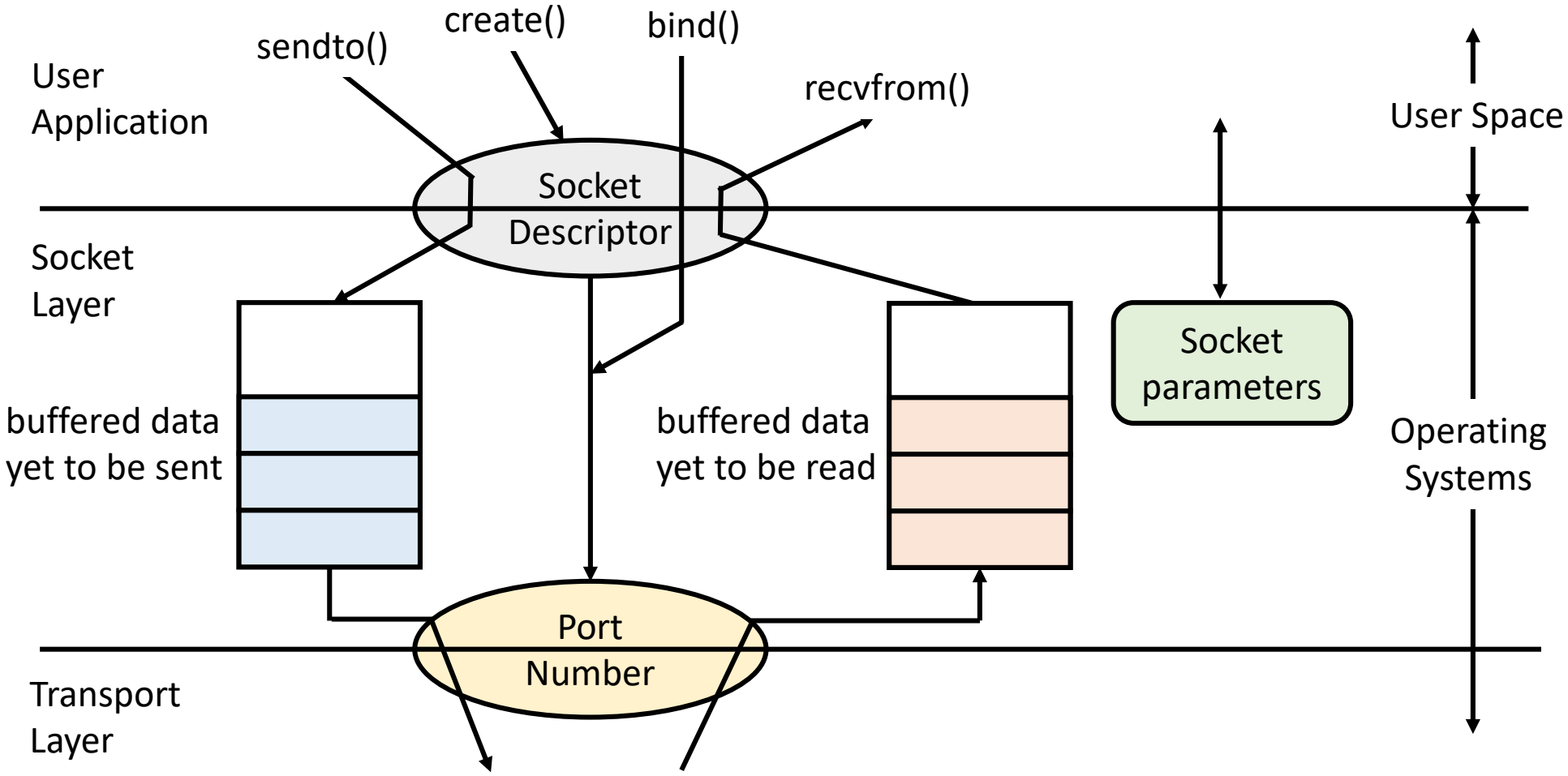
- Connectionless Communication
  - Unit of data are messages (limited length)
  - Correct, but not necessarily complete or in order
  - Usually insecure/not dependable, not confirmed
  - Application must provide its own reliability
  - Sometimes a better option, e.g. stock quote service, as latency matters



Sockets: the Transport Layer API

- Sockets provide an **API for programming networks** at the transport layer.
- 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.
- 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.

Socket Layer



Caution: Sockets support multiple Domains

- Domain defined while socket is created!
  - *int socket(int **domain**, int **type**, int **protocol**)*
- ***domain*** is PF_UNIX, PF_INET, PF_OSI, etc.
  - PF_INET is for communication on the Internet to IP addresses
- ***type*** is either SOCK_STREAM (connection oriented, reliable), SOCK_DGRAM (connectionless) 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)
- While nowadays INET domain/protocols prevail, the approach is more general.

Berkeley Sockets

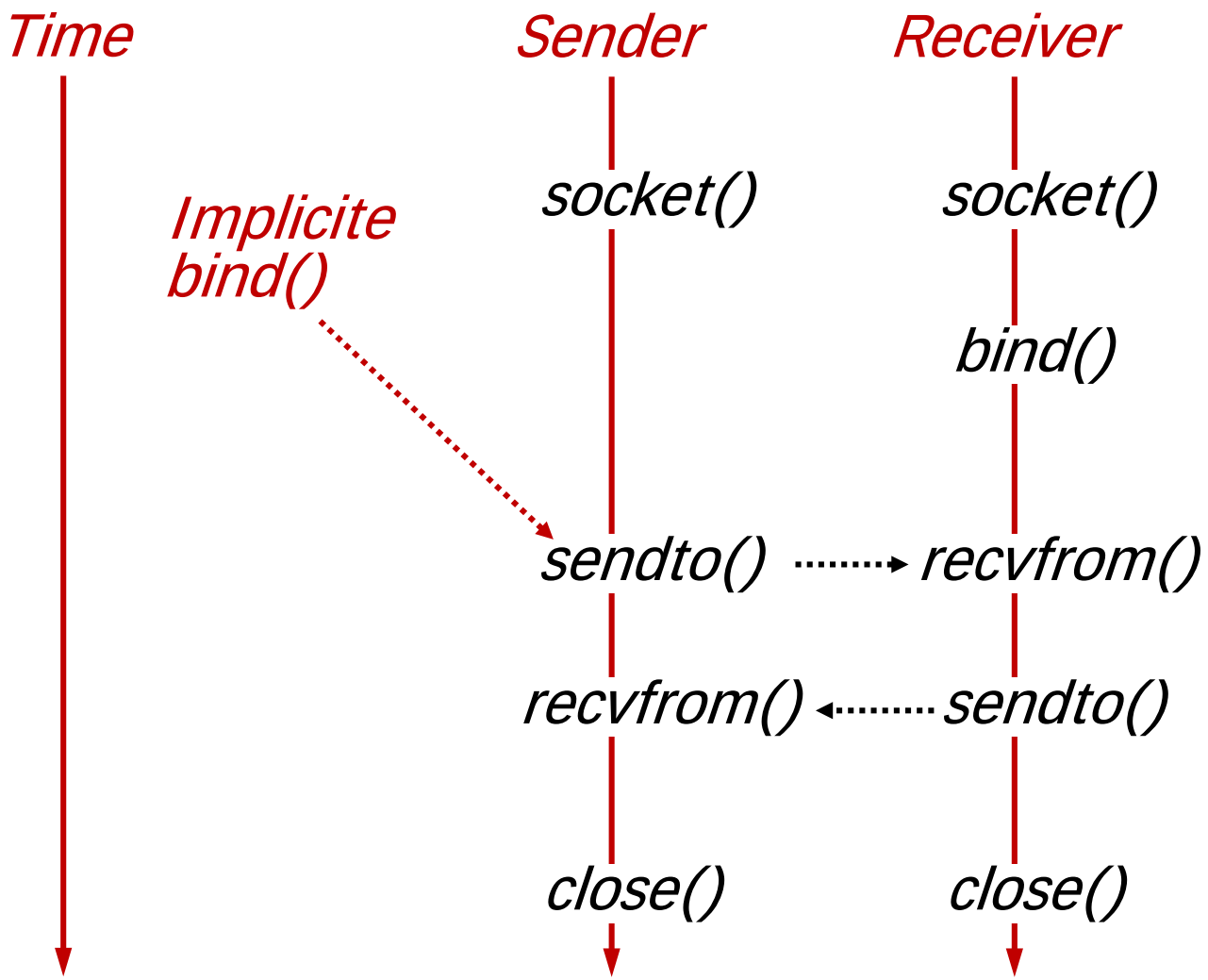
… is a UNIX application programming interface (API) for Internet sockets

… implemented as library

Calls	Meaning
SOCKET	Create a new communication endpoint
BIND	Associate a local address with a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Passively establish an incoming connection
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

- Good source of information: http://beej.us/guide/bgnet/

Datagram Sockets (connectionless)



Datagram Communication over Sockets

[Karl, op. cit.]

- 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`

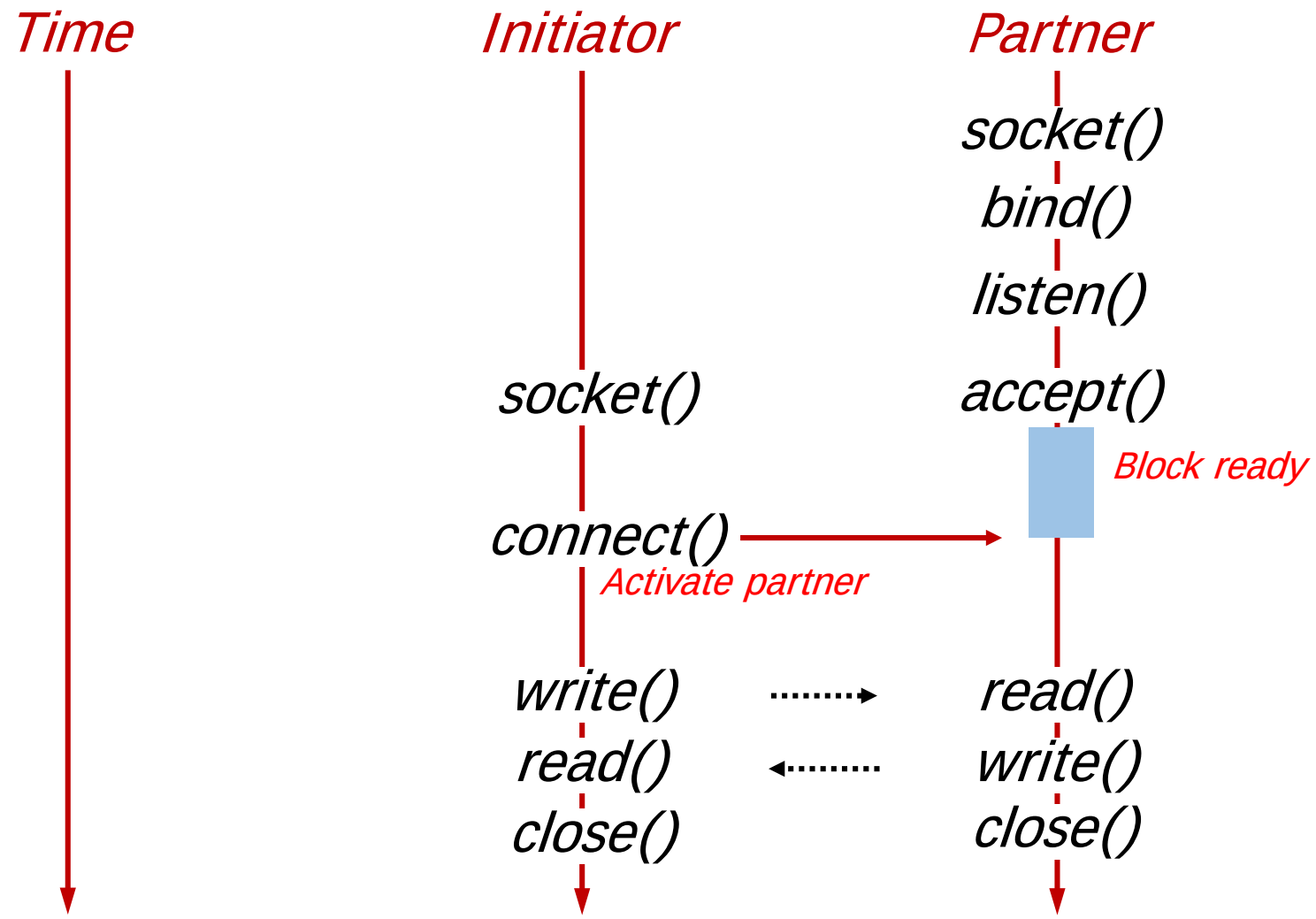
  Timer might be added!

Java API for Datagram Comm.

[wonkwang, op. cit.]

- 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 *InterruptedIOException*)
  - *Connect*
    (throws *SocketException* if port unknown or in use)
  - *close DatagramSocket*

Stream Sockets (connection-oriented)



Byte Streams over Connection-oriented Socket [Karl, op. cit.]

- 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

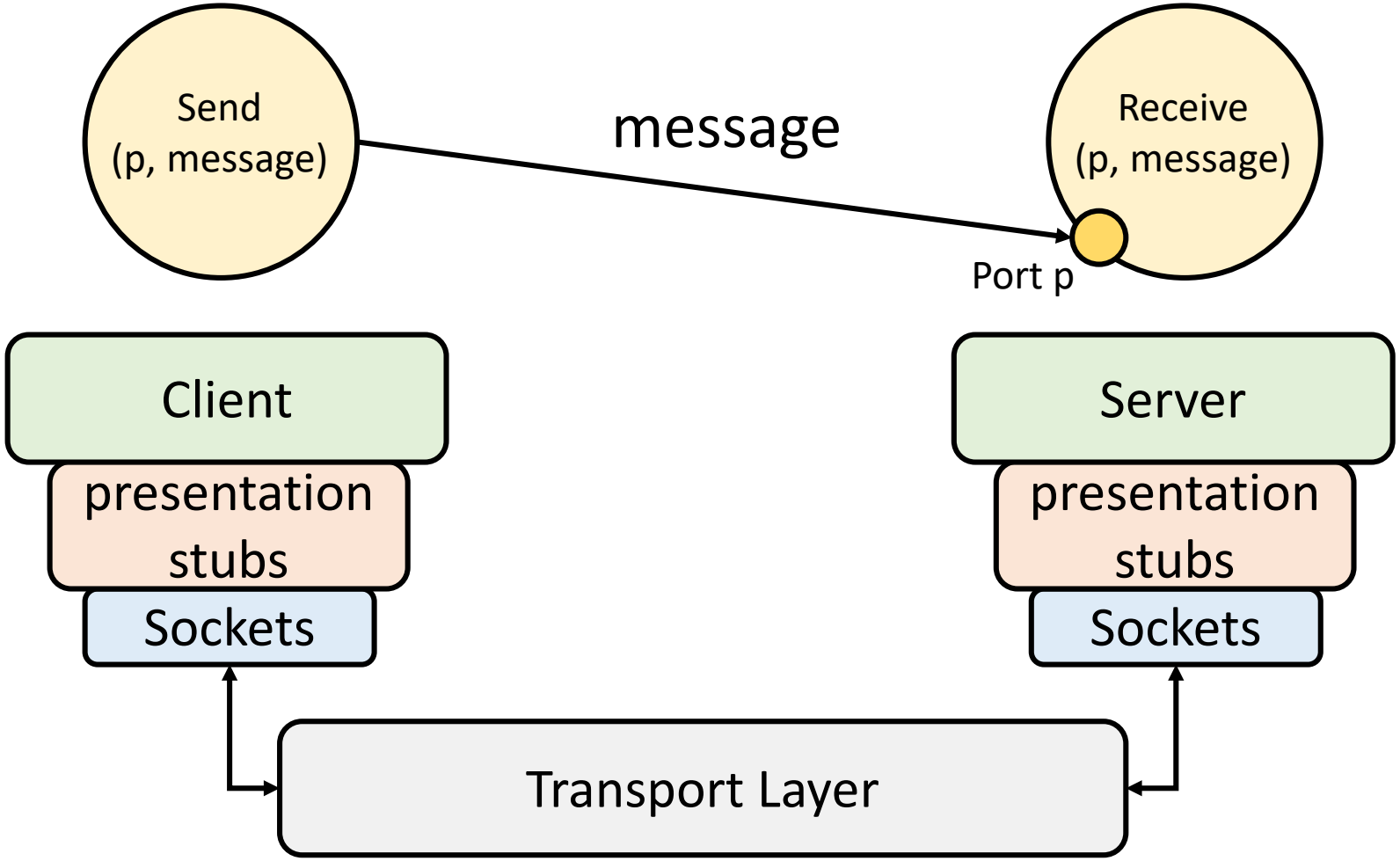
Partner *(server)*

- `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

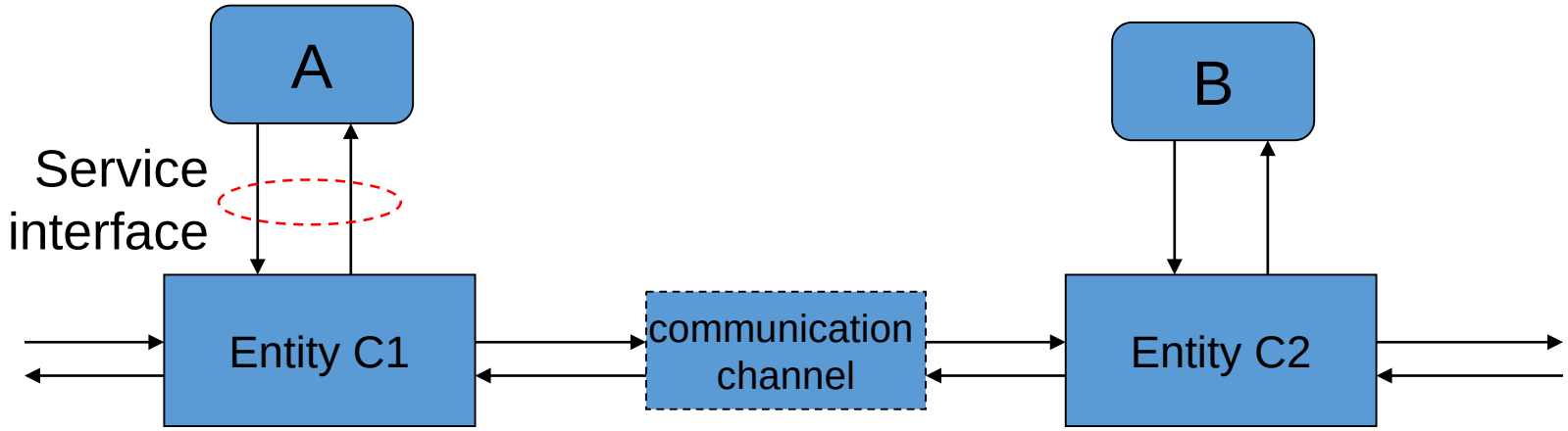
- Class *ServerSocket*
  - *socket constructor* (for listening at a server port)
  - *getInputStream, getOutputStream*
  - *DataInputStream, DataOutputStream*
    (automatic marshaling/unmarshaling)
  - *close* to close a socket
    (raises *UnknownHost, IOException*, etc)

Inter-Process Communication



Communication: a Service offered to Users

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



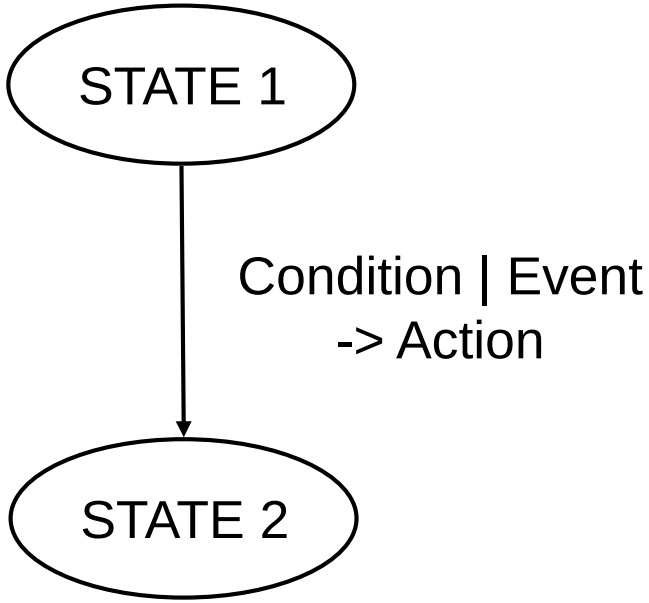
- Entities exchange messages:
  - A message contains "an **envelope**" (aka **header** – an organization part)
  - And, mostly, "the **payload**" (aka **user's data**)

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

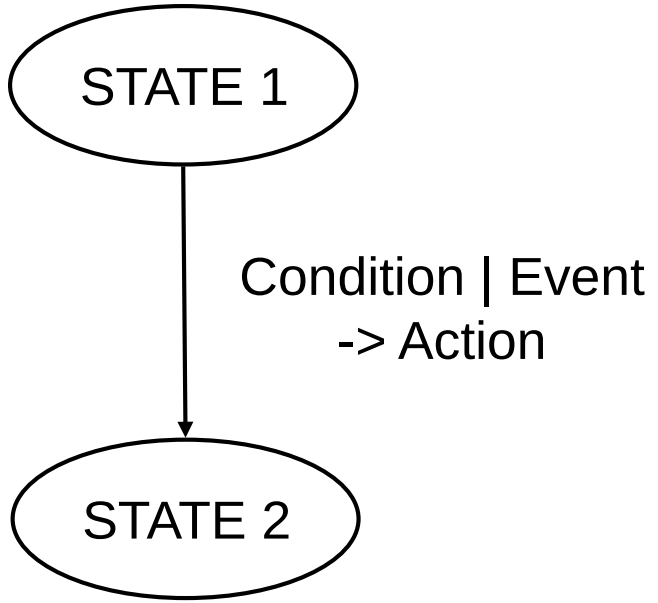
- Formal behavior
  - Rules which constitute the protocol have to be precisely specified
- One popular method: (Extended) Finite State Machine (FSM)
  - Protocol instance/protocol engine at each entity with several states
    - E.g. connected, waiting, …
  - Events/transitions between states
    - Message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



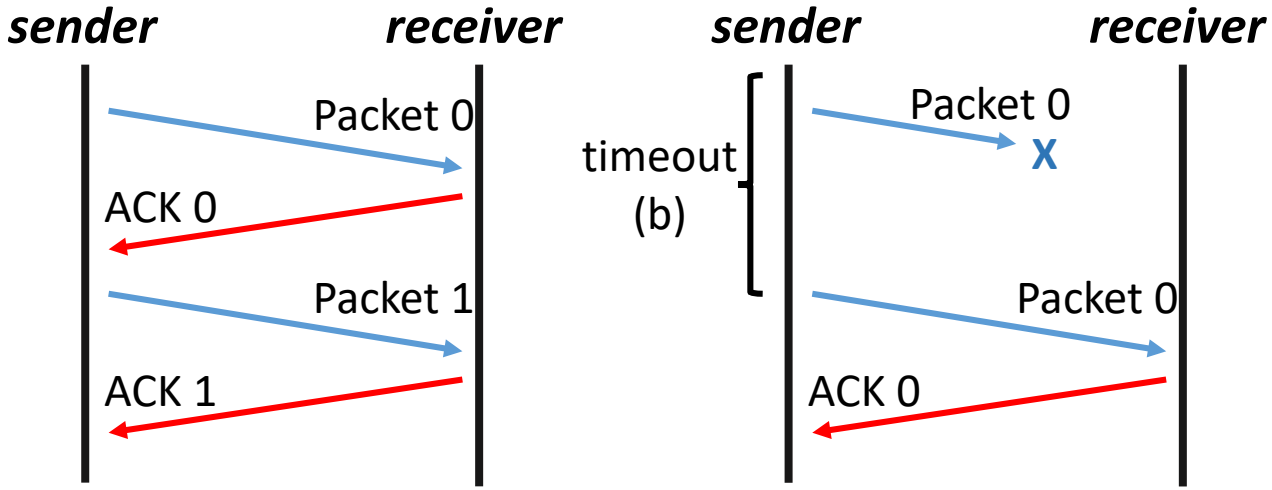
Protocol Specifications

- Finite State Machines (FSM)
  - $\sum = (S,E,A,D,S_0)$
  - $S = \{s^j\}$: countable state space
  - $E = \{e^j\}$: countable set of events
  - $A = \{a^j\}$: countable set of actions
  - D: transition function, D: S x E -> S x A

    ϕ: undefined transition (zero element)
  - S_0: initial state



Ex.: Specification of Send-and-Wait Protocol



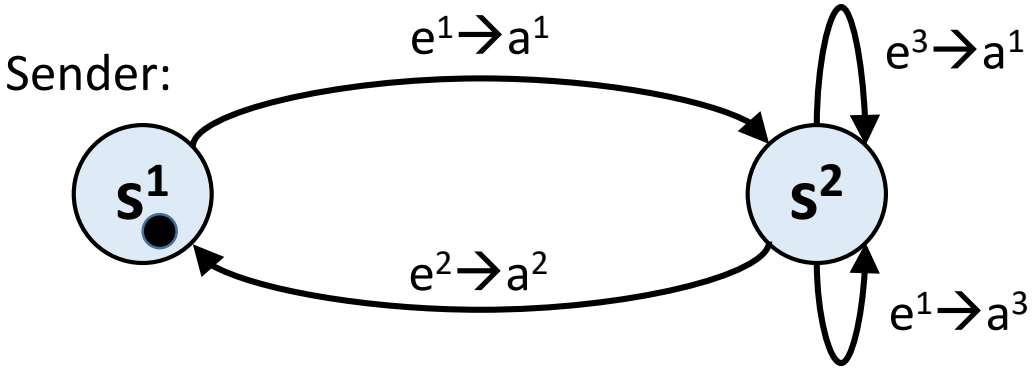
- **Sender** specification: state space, events, actions
  - s^1 - idle — e^1 - data to send (+ Tx **Req**)
  - s^2 - waiting — e^2 - get acknowledge
  - $s^0 = s^1$ — e^3 - timer expired
  - a^1 = <send data, set timer>
  - a^2 = <acknowledge transfer, clear timer> (**Conf**)
  - a^3 = <response: busy> (+ Transmission **Confirmation**)

Transition function:

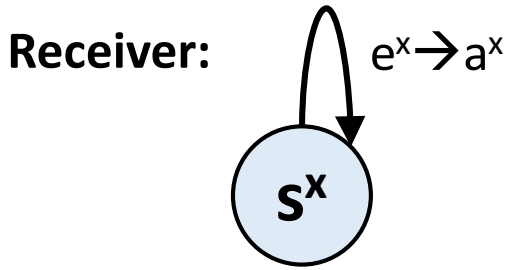
	State	
Event	s^1	s^2
e^1	a^1, s^2	a^3, s^2
e^2	ϕ	a^2, s^1
e^3	ϕ	a^1, s^2

Send-and-Wait Specification (2)

FSM visualized as graph:



- 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

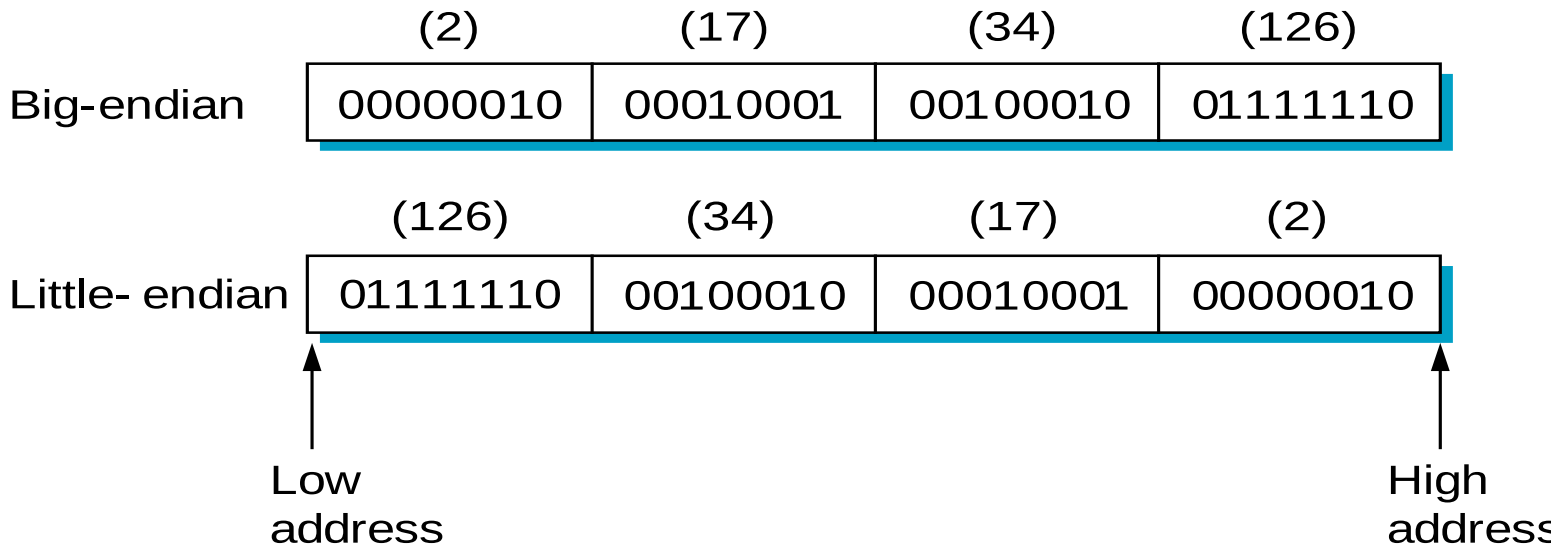


Some remaining questions

- Streams of bits/bytes can be transmitted: so what?
- How do we know what is the **information** inside?

Simple Example

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



Ex.:

on a Motorola 680x0 CPU, the 32 bit integer number 255 is stored as:

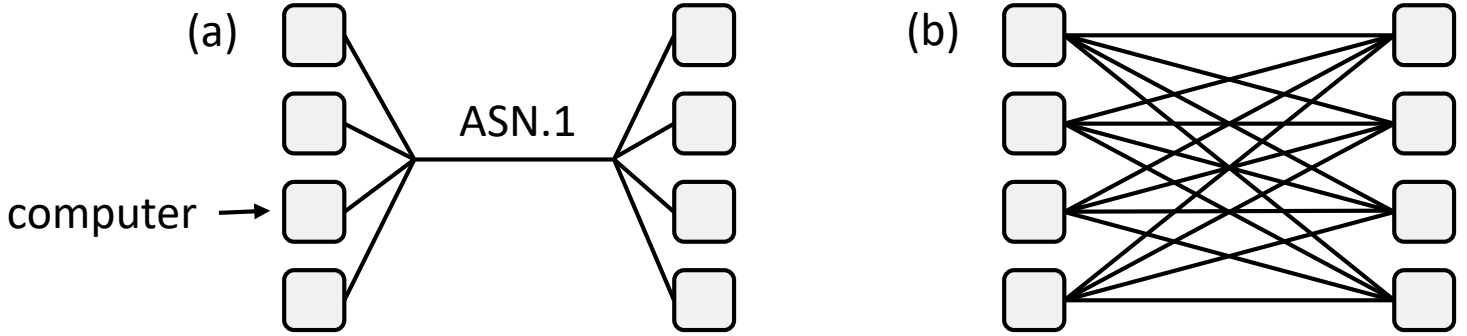
00000000 00000000 00000000 11111111

but an Intel 80x86-CPU stores this as:

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 (serialize)

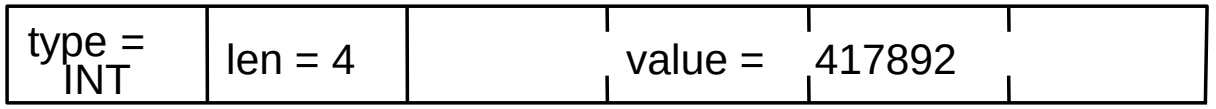


- Conversion Strategy
  - canonical intermediate form
  - receiver-makes-right (an *N* x *N* solution)

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 on the “wire”.

- Tagged versus untagged data



Abstract Syntax Notation One - ASN.1

- A standard **interface description language** for defining data structures that can be serialized & deserialized in a cross-platform way.
- 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).

Abstract Syntax Notation One - ASN.1 (II)

- A data type definition is called „abstract syntax“; it uses a Pascal-like syntax.
- Lexical rules:
  - E.g. lowercase & uppercase letters are different, a type identifier must start with an uppercase letter, …
- 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)

Abstract Syntax Notation One - ASN.1 (III)

- Examples:

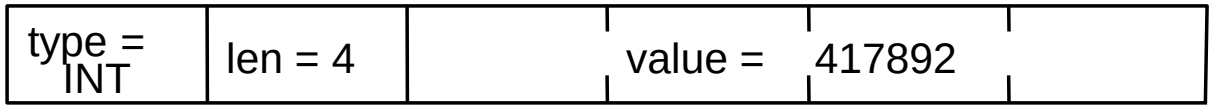
```
MonthsPerYear ::= INTEGER
MonthsPerYear ::= INTEGER (1..12)
Answer ::= ENUMERATED (correct(0), wrong(1),noAnswer(3))
```

- 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

Abstract Syntax Notation One - ASN.1 (III)

- 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.

E.g.:



- The coding of data depends on their type:
  - integer numbers are transmitted in Big-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