OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

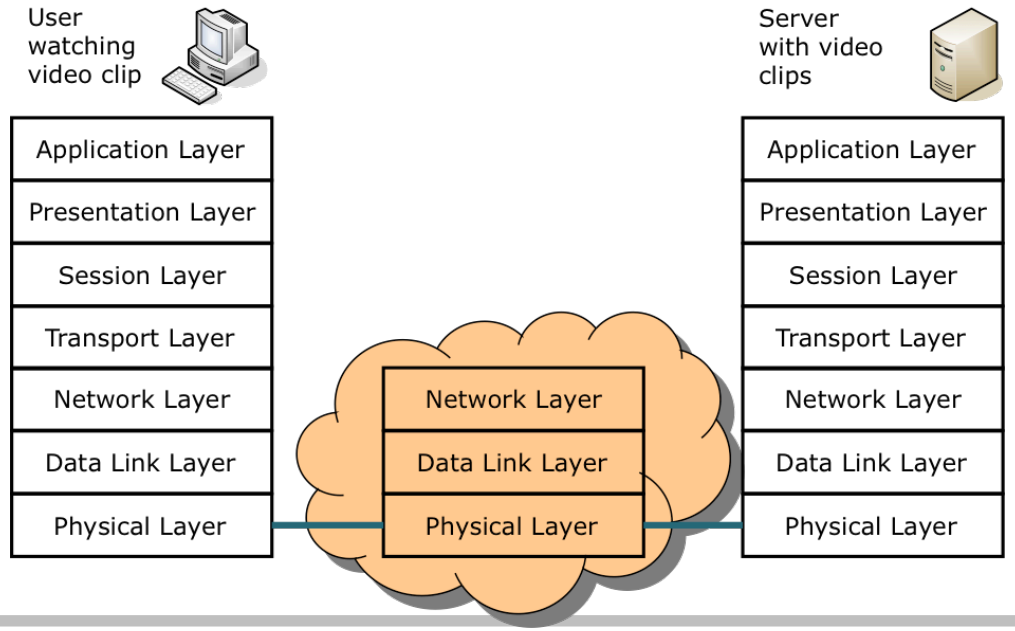
FAKULTÄT FÜR INFORMATIK

Communication and Networked Systems

Computer Networks

Chapter 7: Transport Layer

Prof. Dr. Mesut Güneş
Communication and Networked Systems (ComSys)
OVGU Magdeburg
comsys.ovgu.de | mesut.guenes@ovgu.de



Contents

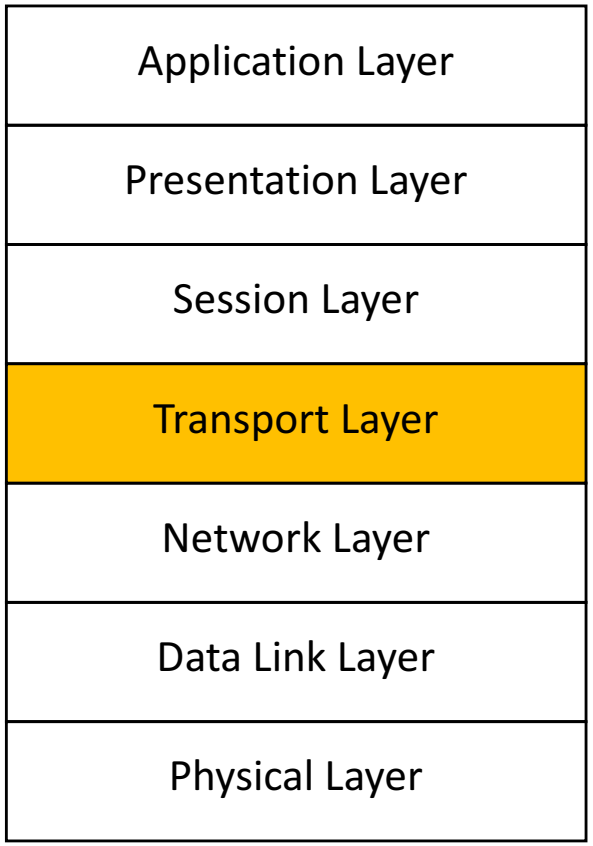
- Design Issues
- User Datagram Protocol (UDP)
- Transport Control Protocol (TCP)
- An example of socket programming

Design Issues

Design Issues

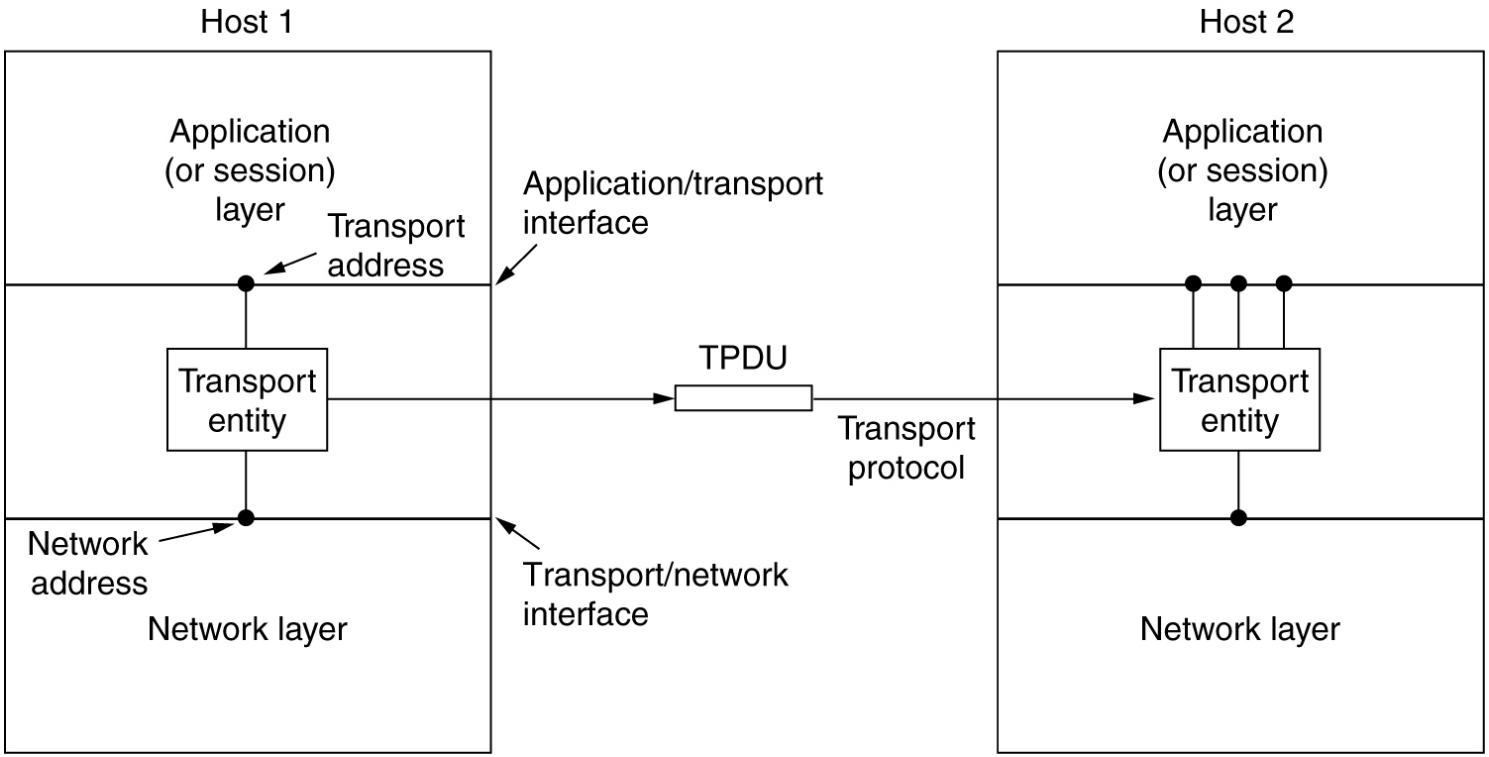
- **Characteristics of the layers below the Transport Layer**
  - Available on the hosts and on the routers
  - Operate in a hop-to-hop fashion
  - Typically unreliable
- **Characteristics of the Transport Layer**
  - Available only on the hosts
  - Operate in a end-to-end fashion
  - Operates like a pipe
- **Services provided to the upper layers**
  - Connection-oriented service
  - Connectionless service
  - Convenient API
    - ➢ Berkeley sockets

OSI Reference Model



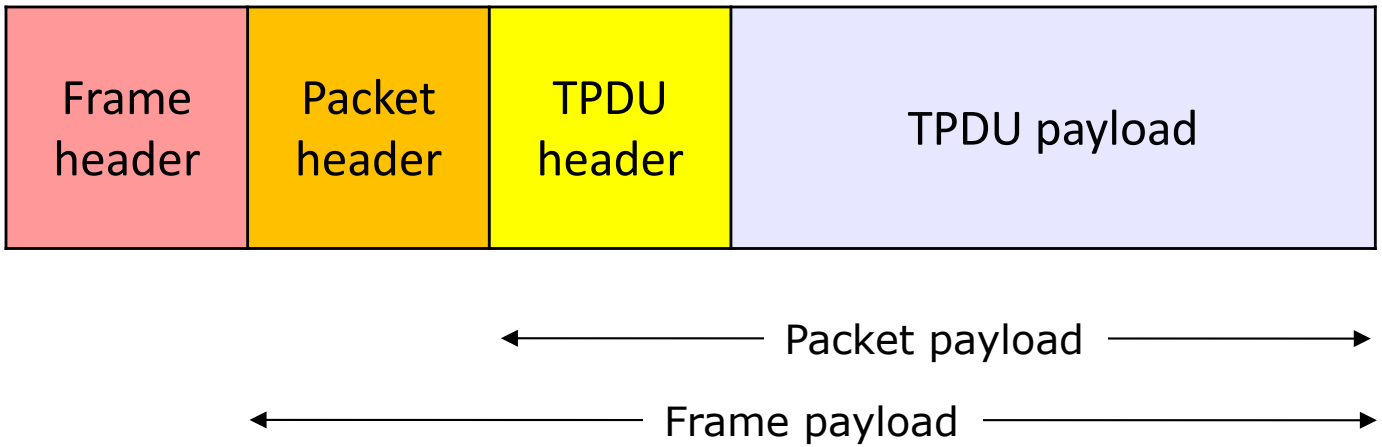
Services Provided to the Upper Layers

- **Goal: provide an efficient, reliable, and cost-effective service**
- **Transport entity is responsible to provide that service**
  - Location: operating system kernel, user process, library package, or network interface card



Transport Service Primitives

- **Some terminology**
  - Messages from transport entities:
    Transport Protocol Data Unit (**TPDU**)
  - TPDUs are contained in network layer packets
  - Packets are contained in data link layer frames

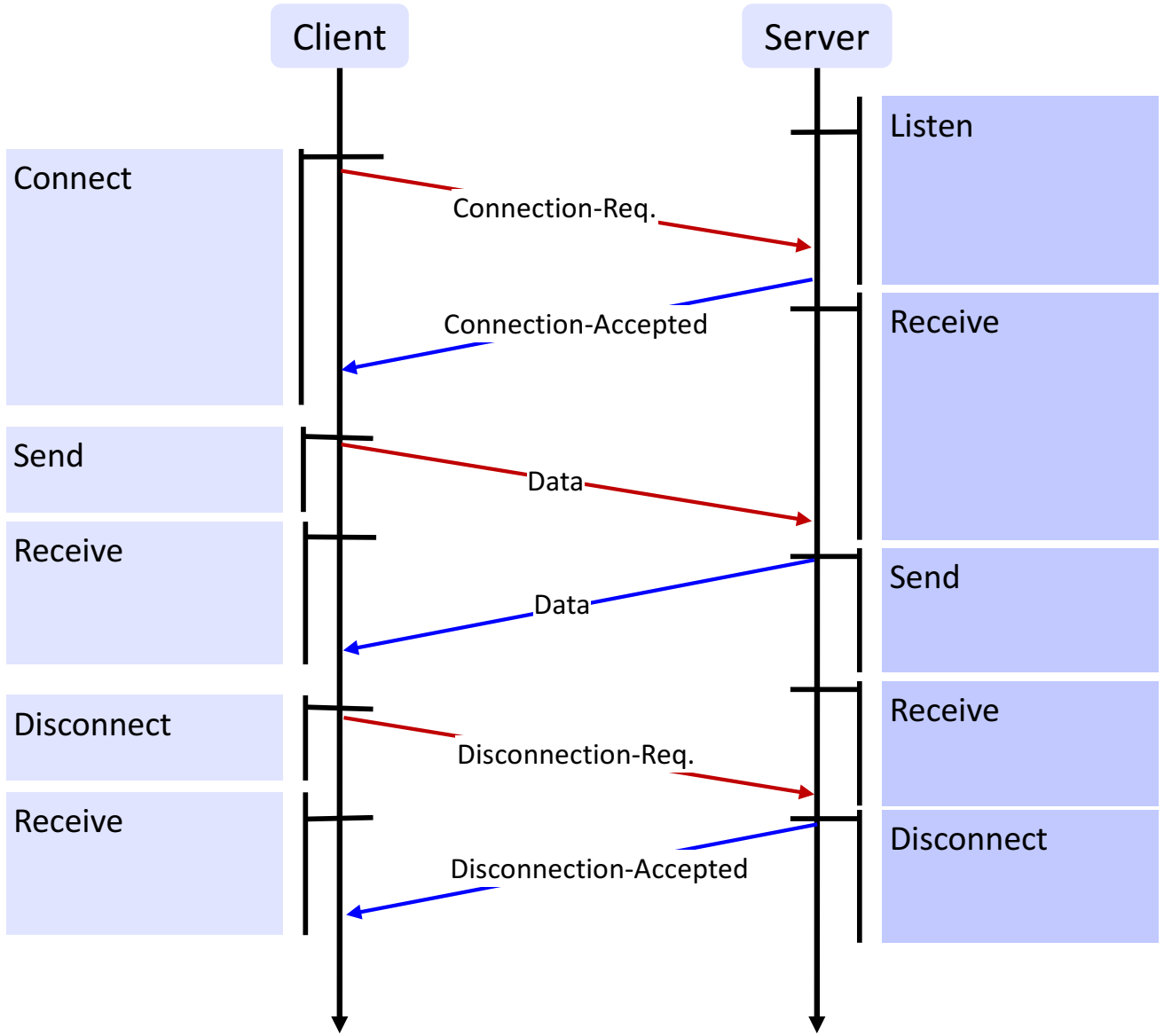


Transport Service Primitives

- **Processes on the application layer expect 100% reliable connections**
  - They are not interested in acknowledgements, lost packets, congestions, …
- **Transport layer provides**
  - Unreliable datagram service (Connectionless)
  - Reliable connection-oriented service
    - Three phases: establishment, communication, termination
- **The primitives for a simple connection-oriented transport service …**

Primitive	Packet sent	Meaning
LISTEN	(none)	Block until some process tries to connect
CONNECT	CONNECTION REQ.	Activeley attempt to establish a connection
SEND	DATA	Send information
RECEIVE	(none)	Block until a DATA packet arrives
DISCONNECT	DISCONNECTION REQ.	This side wants to release the connection

Transport Service Primitives



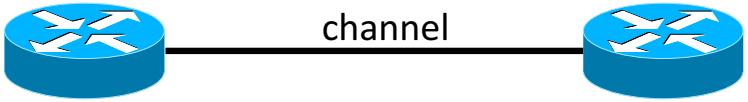
Transport Protocol

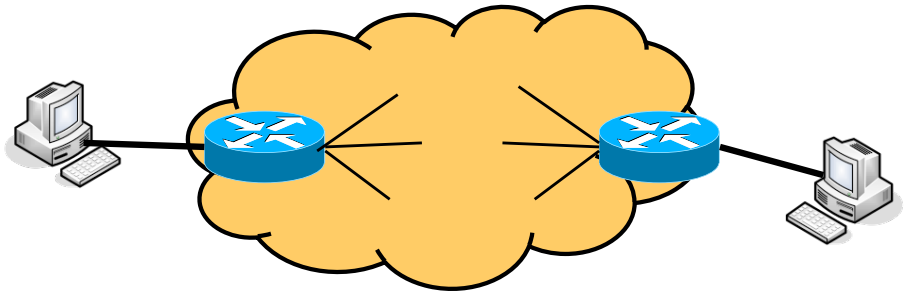
Transport protocol

- Transport service is implemented between transport entities by a transport protocol
- Similar to the protocols studied in chapter »Data Link Layer«
  - Have to deal with: error control, sequencing, and flow control

- **Data link layer environment**
  - On DLL two router communicate directly via a physical channel
    - No explicit addressing is required
    - Channel always there

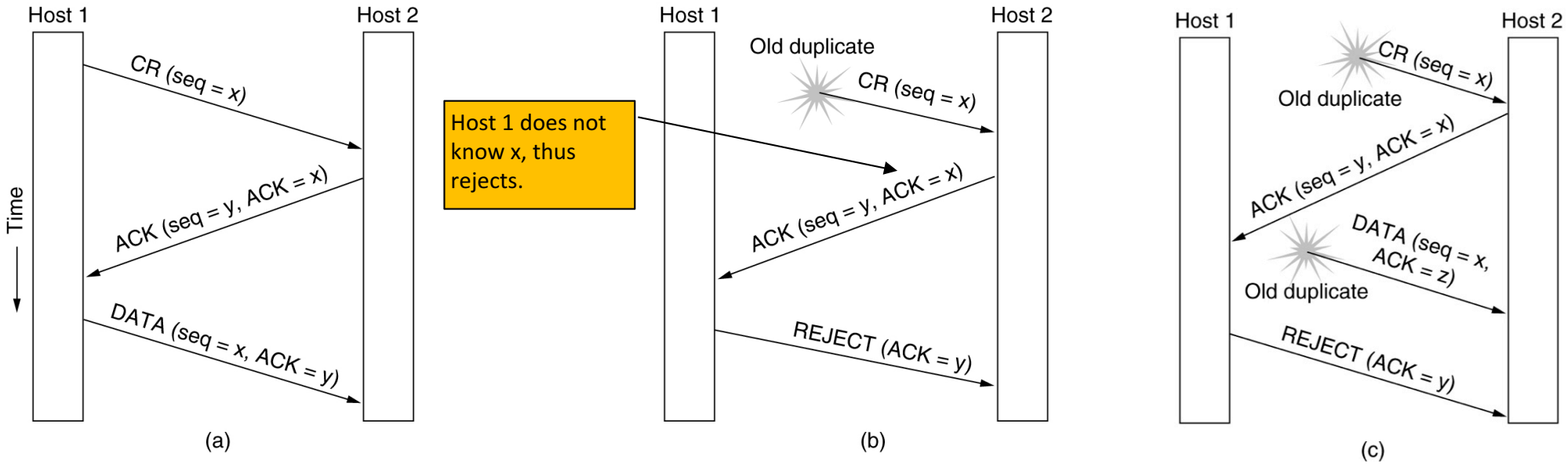
- **Transport layer environment**
  - On TL channel is given by the entire network
    - Explicit addressing of the destination is required
    - Channel is not always there
    - Connection establishment is complicated





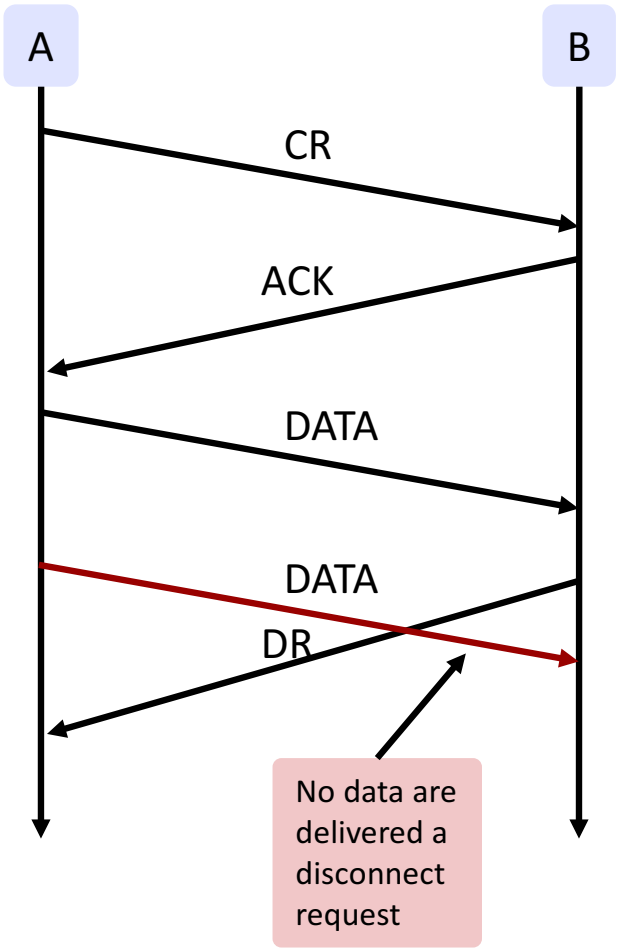
Transport Protocol: Connection Establishment

- **Connection Establishment with the Three-way Handshake**
  - Three protocol scenarios for establishing a connection using a three-way handshake. CR denotes CONNECTION REQUEST
    a) Normal operation
    b) Old CONNECTION REQUEST appearing out of nowhere
    c) Duplicate CONNECTION REQUEST and duplicate ACK



Transport Protocol: Connection Release

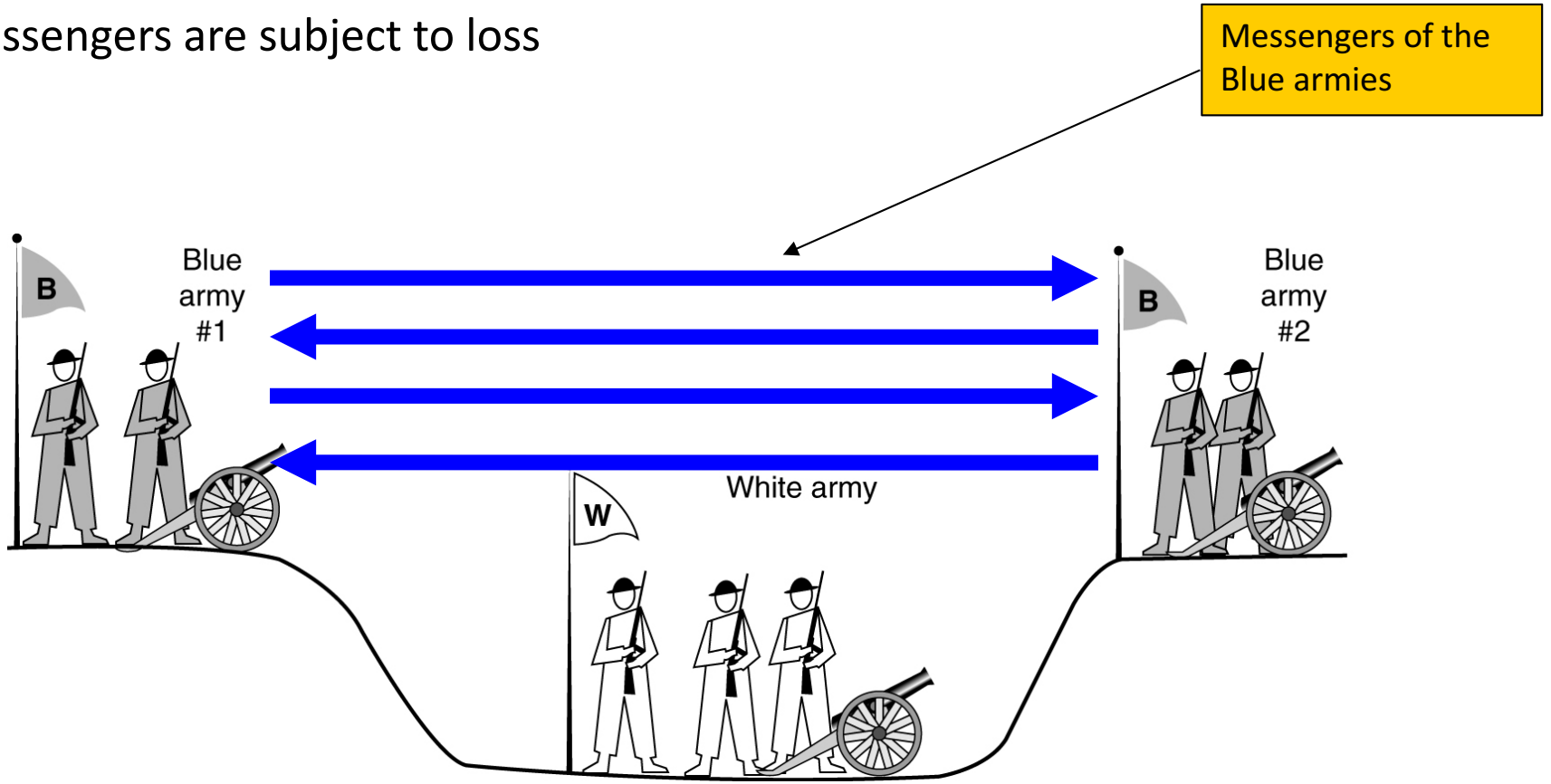
- **Asymmetric release**
  - Telephone system model
  - Either one peer can terminate the connection
  - Danger of data loss
- **Symmetric release**
  - Model of two independent unicast connections
  - Each peer has to terminate the connection explicitly
  - Data can be received by in the non-terminated direction
- **Problem: Data loss can happen on both cases**
  - Question: Is there an optimal solution?



Abrupt disconnection with loss of data.
DR denotes Disconnect Request

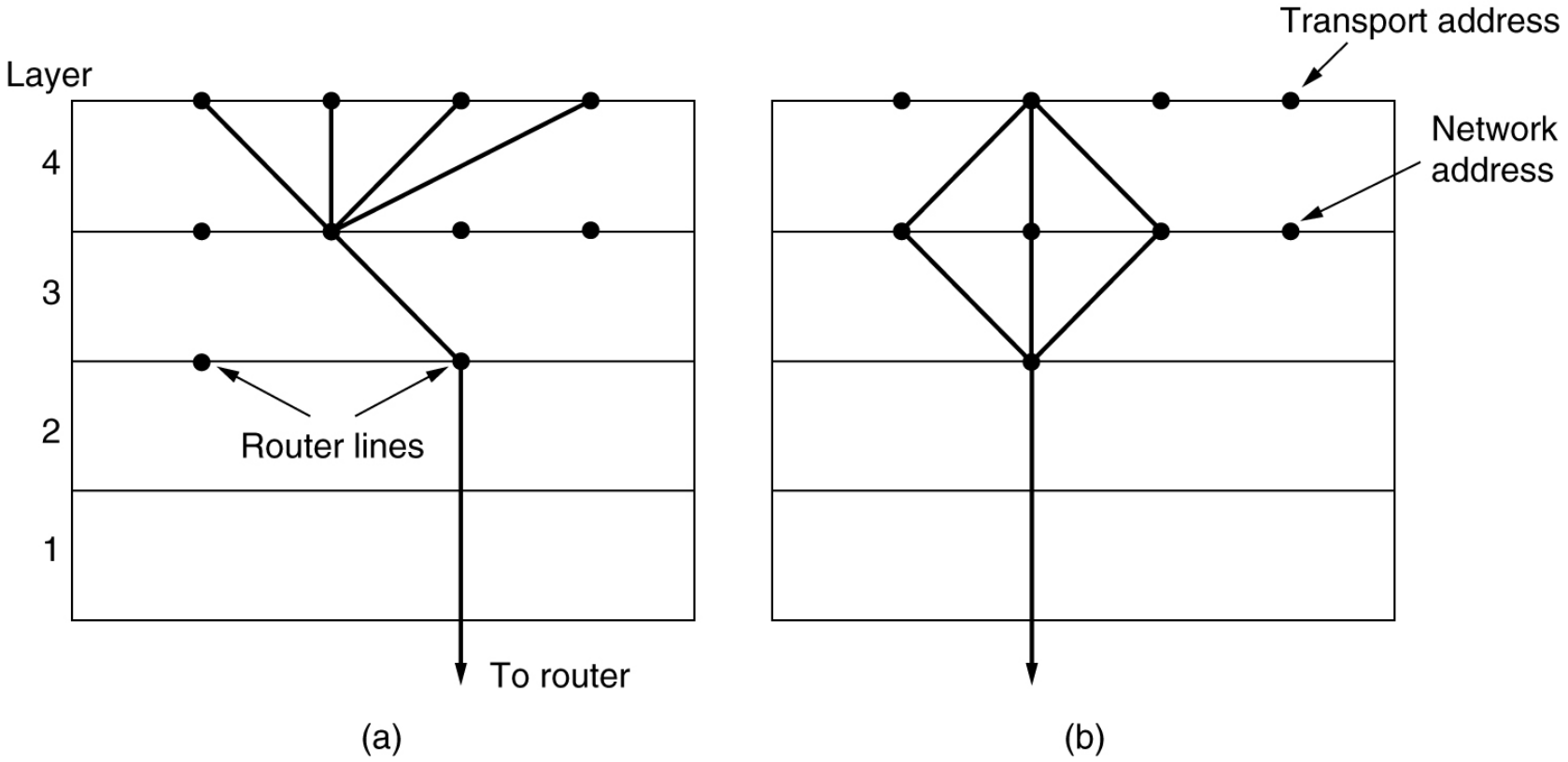
Transport Protocol: Connection Release

- **Famous example to illustrate the problem of controlled (reliable) connection termination: The two-army problem**
  - The Blue armies can only communicate with messengers, i.e., soldiers running through the valley
  - Messengers are subject to loss



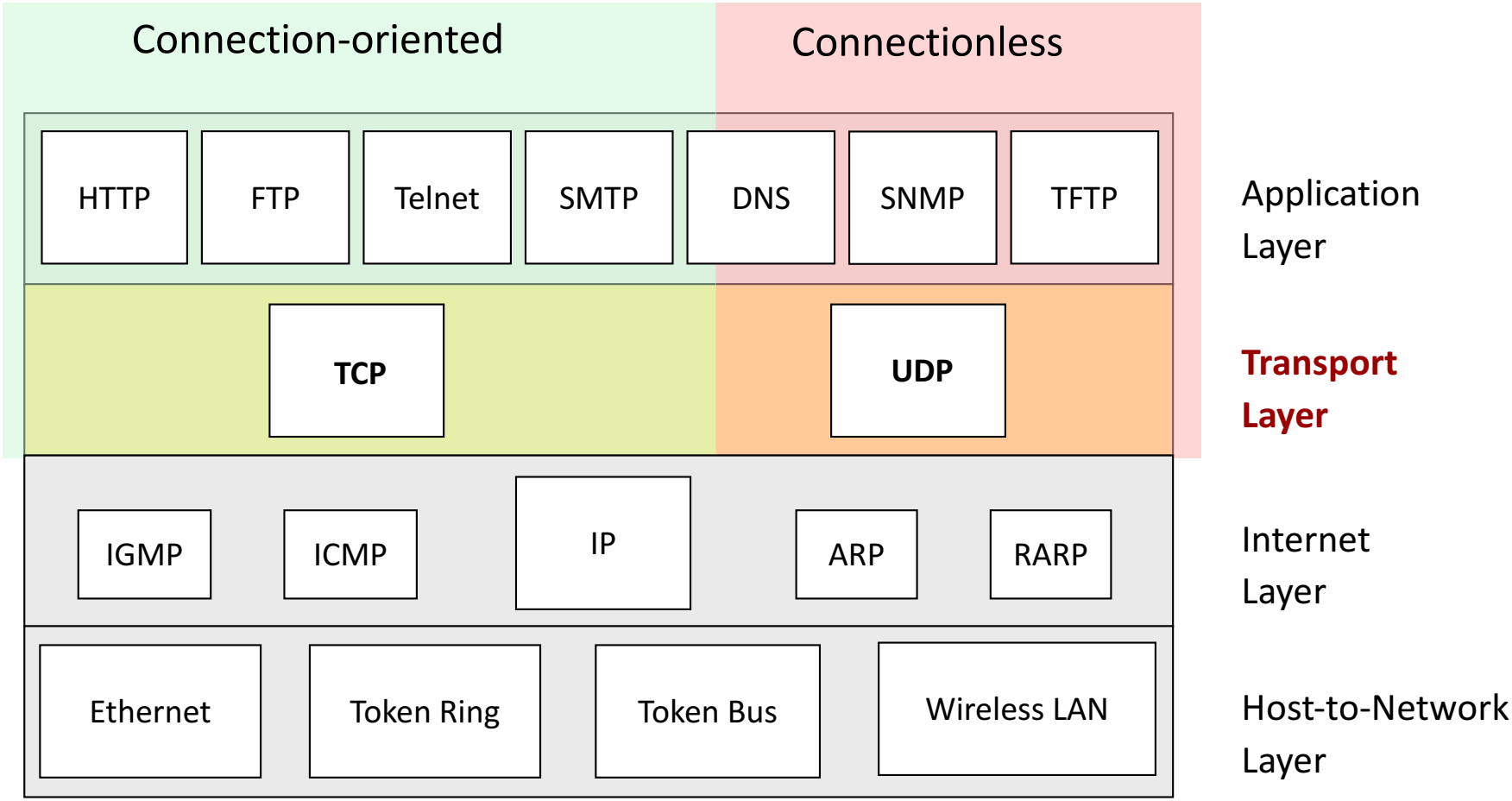
Transport Protocol: Multiplexing

- **Multiplexing of several conversations onto connections**
  - (a) Upward multiplexing: Many transport connections use the same network address
  - (b) Downward multiplexing: Distribute the traffic of one connection over many network connections



Transport Protocols in the TCP/IP Reference Model

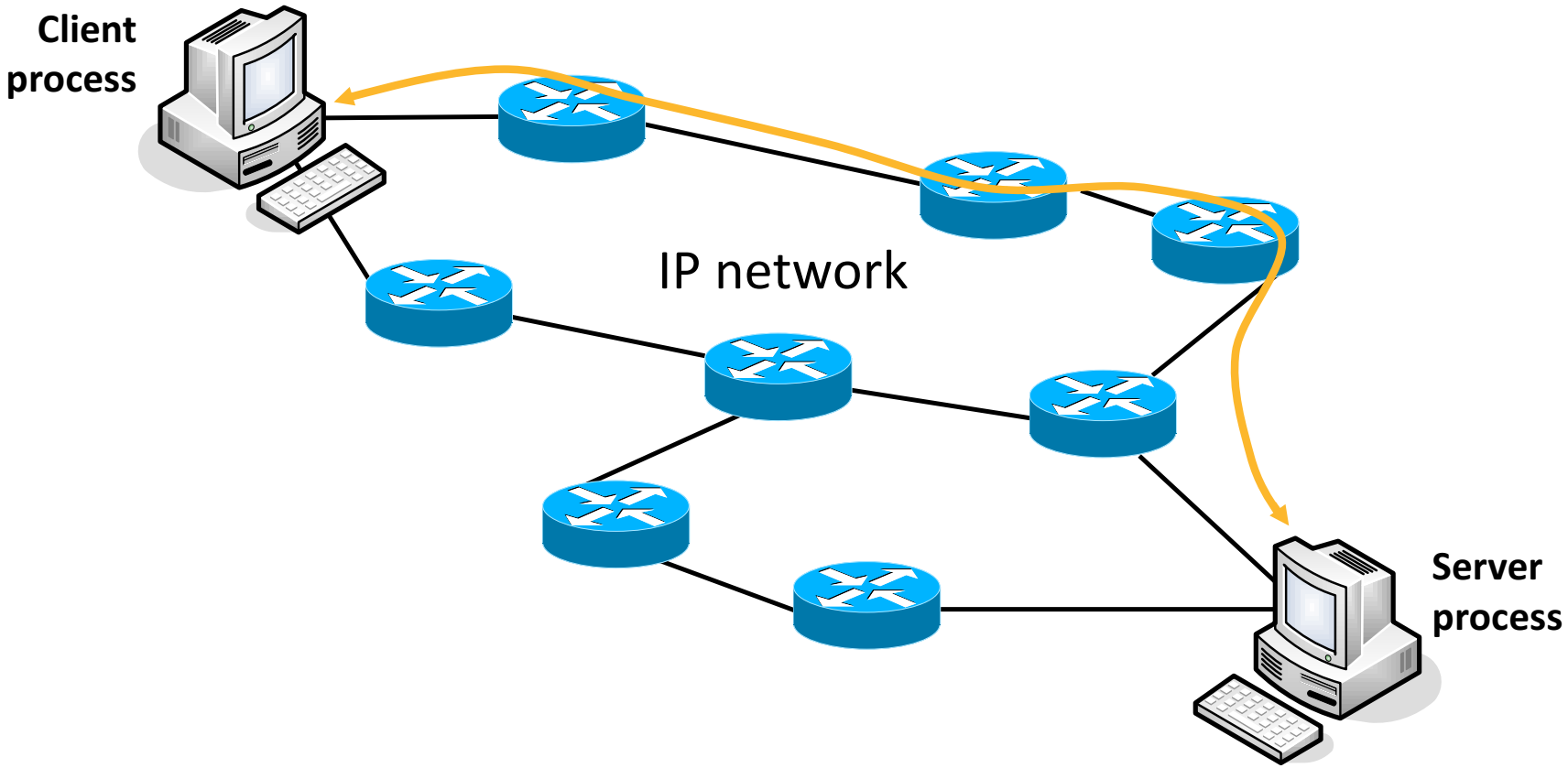
Transport Protocols in the TCP/IP Reference Model



TCP (Transmission Control Protocol): Reliable, connection-oriented.

UDP (User Datagram Protocol): Datagram principle, connectionless, unreliable, without flow control, permutation of the packet order can happen.

The Transport Layer: TCP and UDP



- Transport protocols are used by the application layer as communication services. They enable the communication between **application processes**.
- TCP is a connection-oriented protocol
- UDP is a connectionless and (fast) protocol

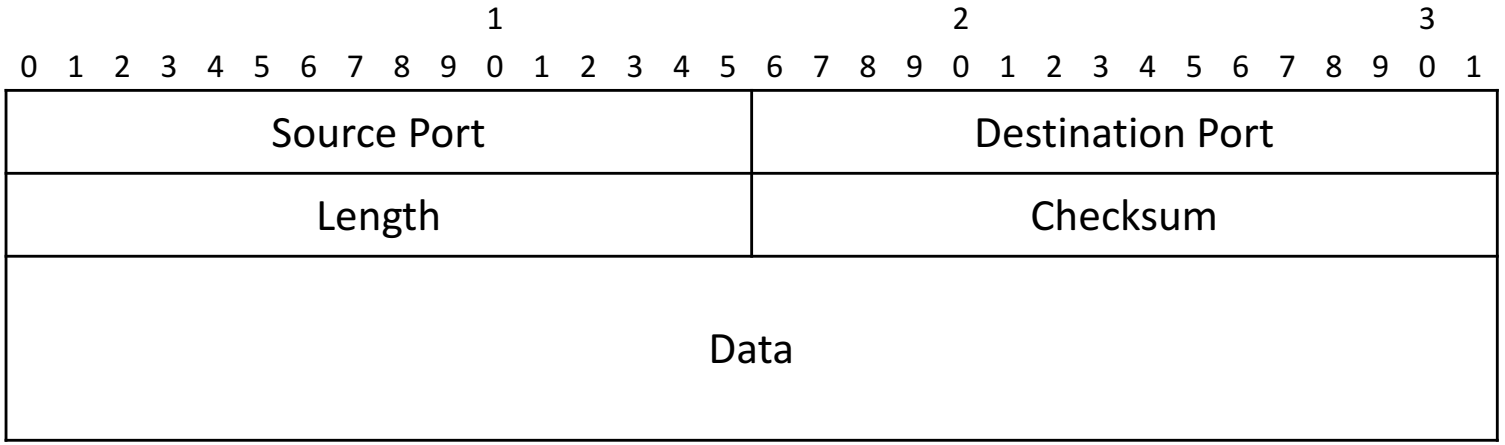
User Datagram Protocol (UDP)

The User Datagram Protocol (UDP)

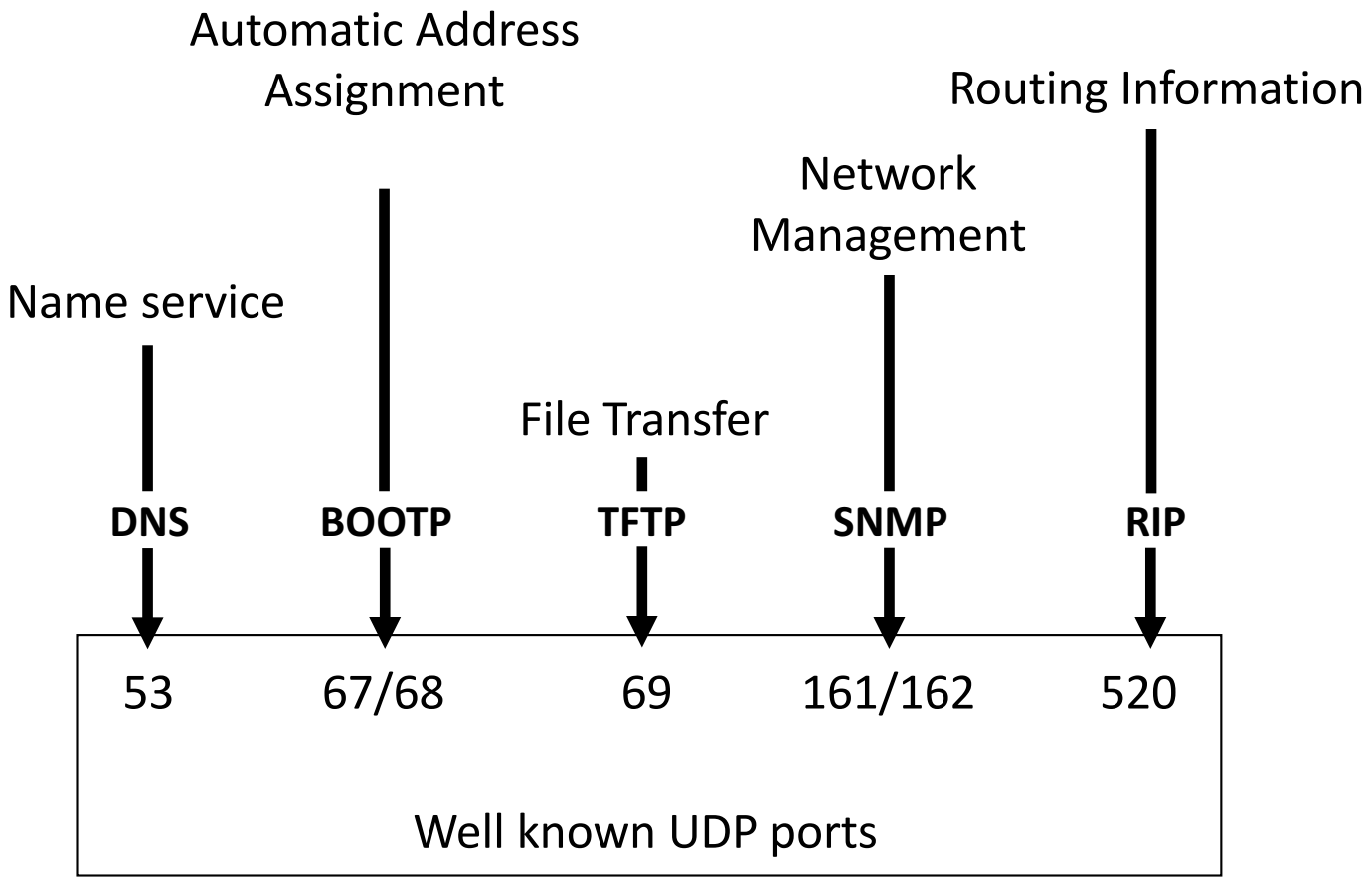
- **Principle: »Keep it simple!«**
  - 8 byte header
  - Like IP: Connectionless and unreliable
  - Small reliability, but fast exchange of information
  - No acknowledgement between communication peers with UDP
    - Incorrect packets are simply discarded
    - Duplication, sequence order permutation, and packet loss are possible
  - The checksum offers the only possibility of testing the packets on transfer errors
  - Possible: ACKs and retransmissions are controlled by the application.
  - Use in multicast (not possible with TCP)
- **Why at all UDP?**
  - Only the addition of a **port** to a network address marks communication clearly.

UDP Header

- **Addressing of the applications by port numbers**
- **Length: indicates the total length (header + data) in 32-bit words**
- **Checksum (optional!): IP does not have a checksum for the data part, therefore it can be a meaningful addition here.**
  - The same procedure as in TCP
- **Data are filled up if necessary to an even byte number, since Message Length counts in 32-bit words**



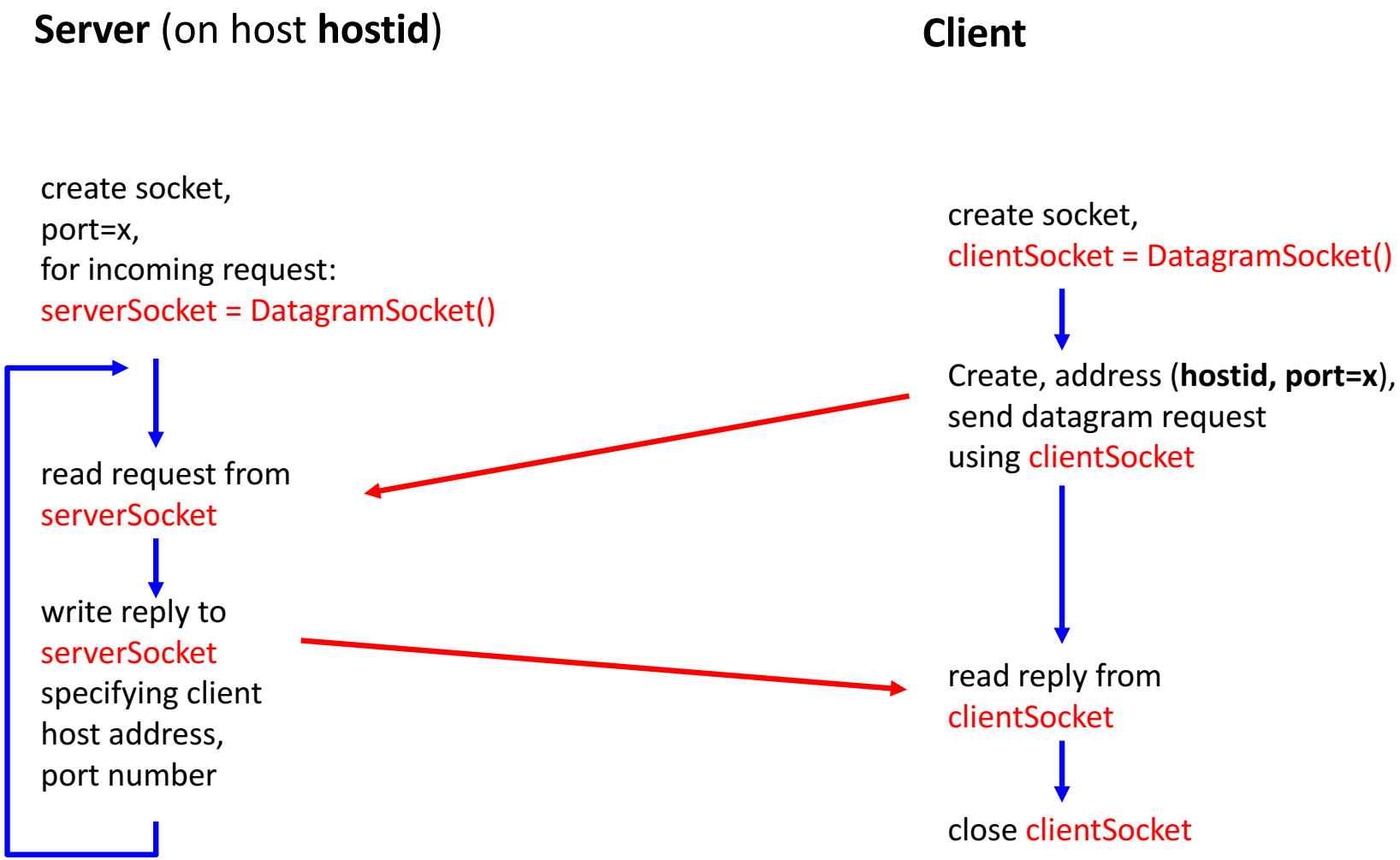
UDP-based Applications



User Datagram Protocol (UDP)

Socket Programming with UDP

Socket Programming with UDP



Example: Java Client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
    public static void main(String arg []) throws Exception
    {
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        DatagramSocket clientSocket = new DatagramSocket();
        InetAddress IPAddress = InetAddress.getByName("hostname");
        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes ();
        DatagramPacket send_pack = new DatagramPacket(sendData, sendData.length,
                                                      IPAddress, 9876);

        clientSocket.send(send_pack);
        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String modifiedSentence = new String(receivePacket.getData());
        System.out.println("FROM SERVER:" + modifiedSentence);
        clientSocket.close();
    }
}
```

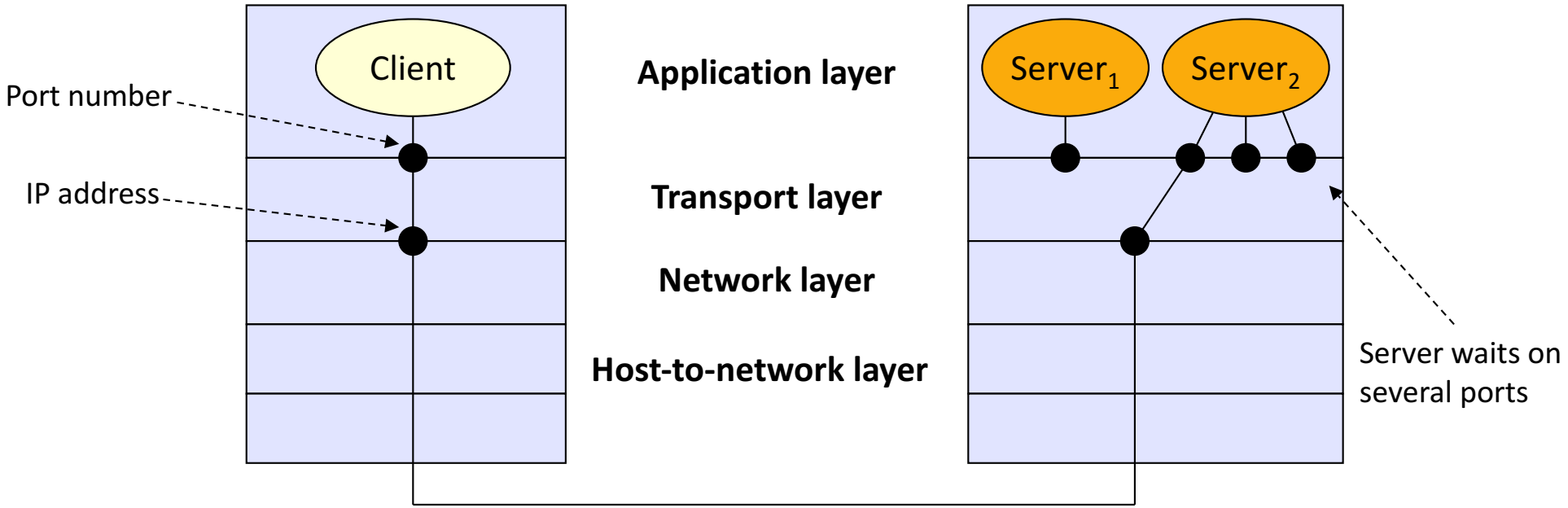
Example: Java Server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
  {

    DatagramSocket serverSocket = new DatagramSocket(9876);

    byte[] receiveData = new byte[1024];
    byte[] sendData = new byte[1024];

    while (true)
    {
      DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
      serverSocket.receive(receivePacket);
      String sentence = new String(receivePacket.getData());
      InetAddress IPAddress = receivePacket.getAddress();
      int port = receivePacket.getPort();
      String capitalizedSentence = sentence.toUpperCase();
      sendData = capitalizedSentence.getBytes();
      DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length,
                                                     IPAddress, port);
      serverSocket.send(sendPacket);
    }
  }
}
```

Transport Control Protocol (TCP)

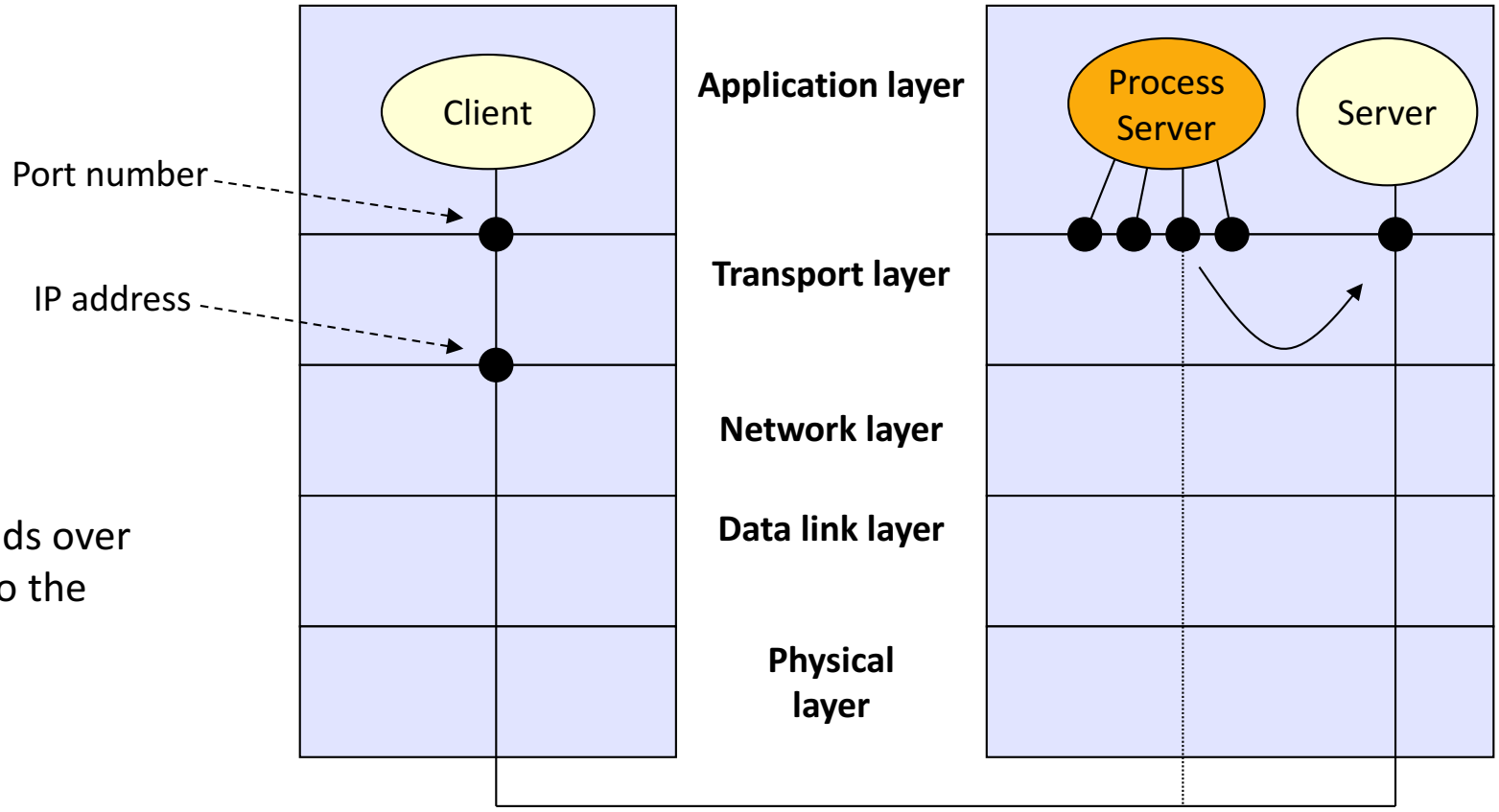
Characteristics of TCP

- **Connection-oriented and reliable (error-free, keeps packet order, without duplicates)**
- **Error handling, acknowledgements, flow control (Sliding Window procedure)**
- **Byte stream, not message stream, i.e., message boundaries are not preserved**
- **Segmentation (max. segment size of 64 KByte)**
- **»Urgent«-messages outside of flow control**
- **Limited QoS**
- **Addressing of the application by port numbers**
  - Port numbers below 1024 are called **well-known ports**, these are reserved for standard services



TCP as a Reliable Connection

If the server port is unknown, the use of a process server (Initial Connection Protocol) is possible:



Process server hands over new connections to the inquired server

Alternative:

Name server (comparable to a phone book) returns the destination port

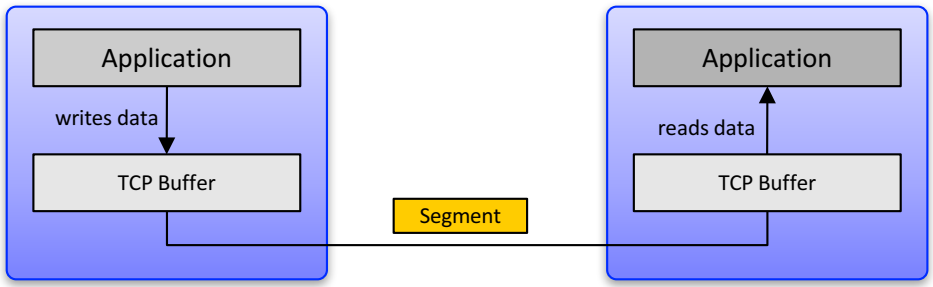
TCP as a Reliable Connection

- **Establishes logical connections between two Sockets**
  - IP address + 16 bit port number (48 bit address information)
- **For an application, sockets are the access points to the network**
- **A socket can be used for several connections at the same time**
- **TCP connections are always full-duplex and point-to-point connections**
- **TPDUs are called segments**
- **Segments are being exchanged for realizing**
  - Connection establishment
  - Agreement on a window size
  - Data transmission
  - Sending of confirmations
  - Connection termination

TCP: Overview

RFCs: 793, 1122, 1323, 2018, 2581

- **Point-to-point**
  - One sender, one receiver
- **Reliable, in-order byte stream**
  - No »message boundaries«
- **Pipelined**
  - TCP congestion and flow control set window size
- **Send & receive buffers**
- **Full duplex data**
  - Bi-directional data flow in same connection
  - Maximum segment size (MSS)
- **Connection-oriented**
  - Handshaking (exchange of control msgs) init's sender, receiver state before data exchange
- **Flow controlled**
  - sender will not overwhelm receiver



Transport Control Protocol (TCP)

Socket Programming in TCP

Socket Programming in TCP

Server side

- The receiving application process (server) has to run at first
- The server provides a socket over which connection requests are received (i.e. a port is made available)
- In order to be able to receive requests of several clients, the server provides a new socket for a connection request of each client

Client side

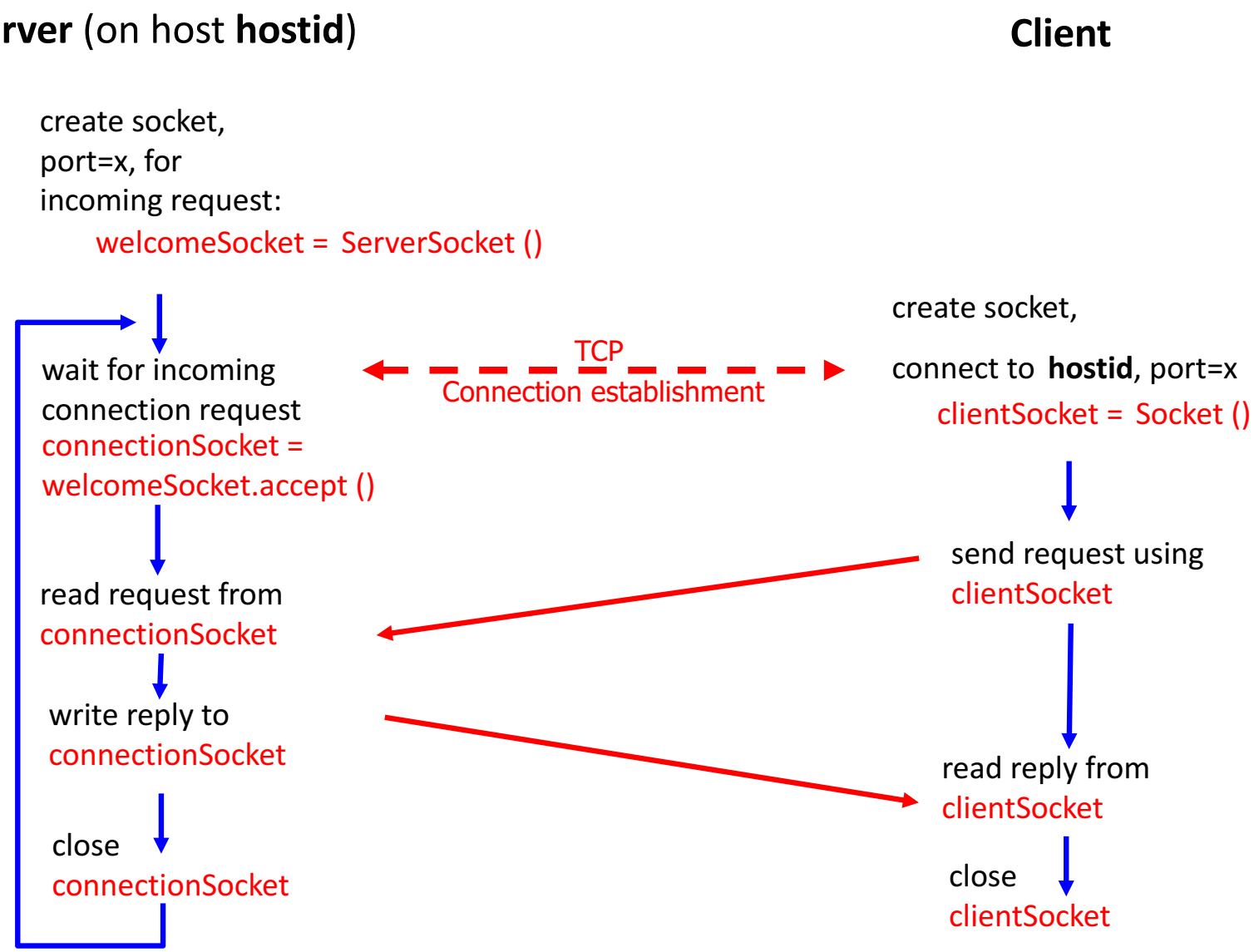
- The client generates a socket
- The client creates a request with IP address and port of the server
- When the client creates its socket, a connection establishment to the server is made

Socket Primitives in TCP

- **For communication via TCP, a set of primitives exists that an application programmer uses for initializing and carrying out a communication.**
- **The essential primitives are:**

Primitive	Meaning
SOCKET	Creation of a new network access point
BIND	Assign a local address with the socket
LISTEN	Wait for arriving connecting requests
ACCEPT	Accept a connecting request
CONNECT	Attempt of a connection establishment
SEND	Send data over the connection
RECEIVE	Receive data on the connection
CLOSE	Release of the connection

Socket programming in TCP



Example: Java Client (TCP)

```
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(string argv[]) throws Exception
    {
        String sentence;
        String modifiedSentence;
        BufferedReader inFromUser = new BufferedReader(new InputStreamReader(System.in));
        Socket clientSocket = new Socket("hostname", 6789);
        DataOutputStream outToServer = new
                DataOutputStream(clientSocket.getOutputStream ());

        BufferedReader inFromServer = new BufferedReader(new
                InputStreamReader(clientSocket.getInputStream()));

        sentence = inFromUser.readLine();
        outToServer.writeBytes(sentence + "\ n");
        modifiedSentence = inFromServer.readLine();
        System.out.println("FROM SERVER: " + modifiedSentence);
        clientSocket.close();
    }
}
```

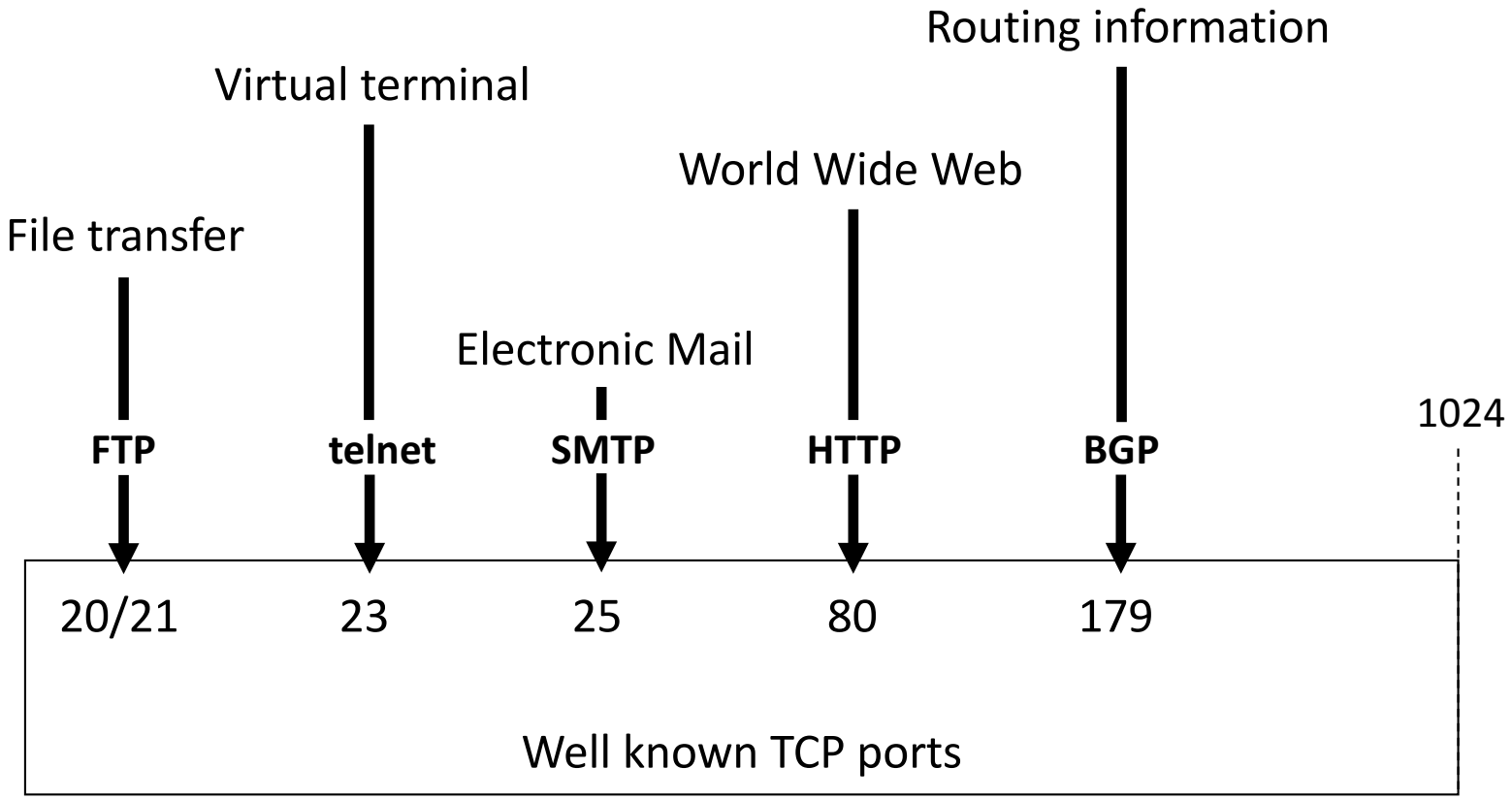
Example: Java Server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(string arg []) throws Exception
    {
        String clientSentence;
        String capitalizedSentence;
        ServerSocket welcomeSocket = new ServerSocket(6789);

        while (true) {
            Socket connectionSocket = welcomeSocket.accept();
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));

            DataOutputStream outToClient = new
                DataOutputStream(connectionSocket.getOutputStream());

            clientSentence = inFromClient.readLine();
            capitalizedSentence = clientSentence.toUpperCase() + "\ n";
            outToClient.writeBytes(capitalizedSentence);
        }
        welcomeSocket.close();
    }
}
```

Transport Control Protocol (TCP)

Details

TCP-based Applications



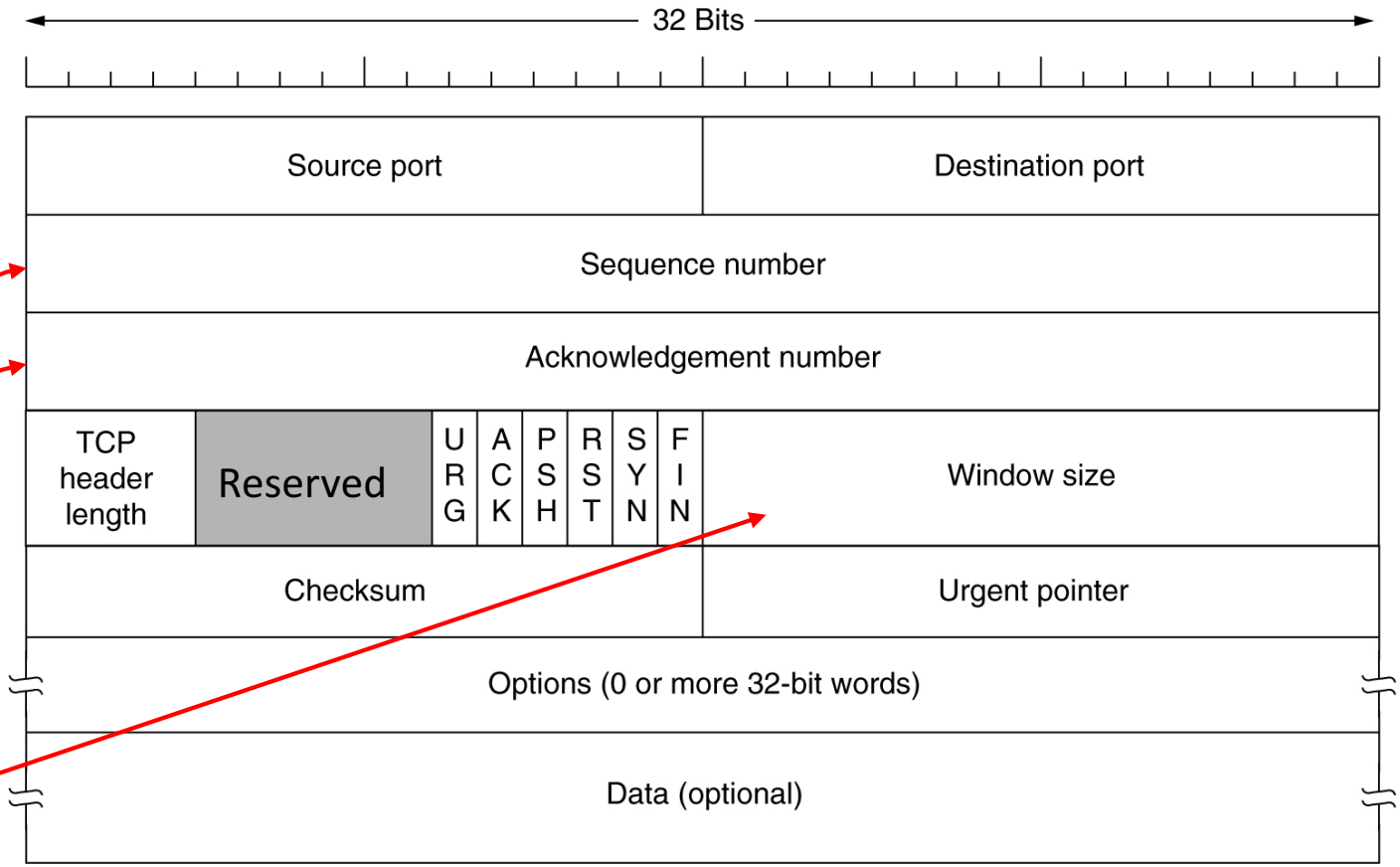
- Port number is 16-bit address, currently three different ranges
- Ports in the range 0-1023 are **Well-Known** (a.k.a. “system”)
- Ports in the range 1024-49151 are **Registered** (a.k.a. “user”)
- Ports in the range 49152-65535 are **Dynamic/Private**
- See for more information: http://www.iana.org/assignments/port-numbers

The TCP Header

- 20 byte header
- Plus options
- Up to 65495 data bytes

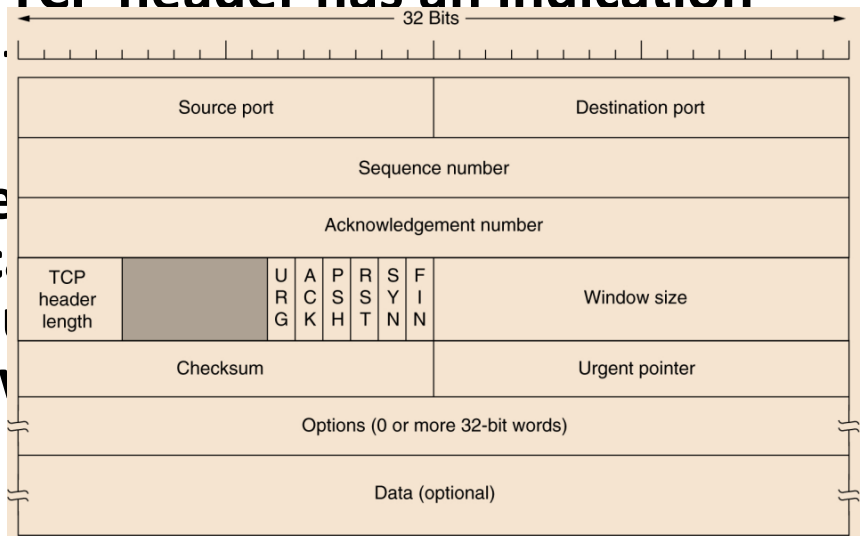
Counting bytes of data (not segments!)

Number of bytes receiver is willing to accept



The TCP Header

- **Source and Destination Port: port number of sender resp. receiver**
- **Sequence Number/Acknowledgment Number: Segments have a 32 bit sequence and acknowledgement number for the window mechanism in flow control (Sliding Window).**
  - Sequence and acknowledgement number count single bytes!
  - The acknowledgement number indicates the next expected byte!
  - Sequence numbers begin not necessarily with 0! A random value is chosen to avoid a possible mix-up of segments for a new connection with old segments from the last connection which could arrive late.
  - Piggybacking, i.e., an acknowledgement can be sent in a data segment.
- **Header Length: As in case of IP, also the TCP header has an indication of its length. The length is counted in 32**
- **Res = Reserved for later use.**
- **Window Size: Size of the receiver's buffe flow control: the window of a flow indic same time can be sent. The size of the b bytes can be stored at the receiver. The adapted to this value.**

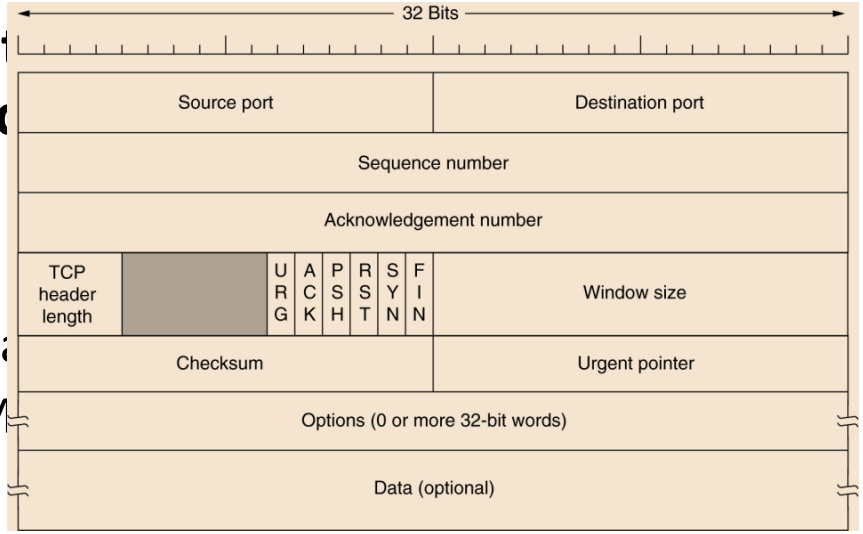


The TCP Header

- **Source and Destination Port: port number of sender resp. receiver**
- **Sequence Number/Acknowledgment Number: Segments have a 32 bit sequence and acknowledgement number for the window mechanism in flow control (Sliding Window).**
  - Sequence and acknowledgement number count single bytes!
  - The acknowledgement number indicates the next expected byte!
  - Sequence numbers begin not necessarily with 0! A random value is chosen to avoid a possible mix-up of segments for a new connection with old segments from the last connection which could arrive late.
  - Piggybacking, i.e., an acknowledgement can be sent in a data segment.
- **Header Length: As in case of IP, also the TCP header has an indication of its length. The length is counted in 32-bit words.**
- **Res = Reserved for later use.**
- **Window Size: Size of the receiver's buffer for the connection. Used in flow control: the window of a flow indicates, how many bytes at the same time can be sent. The size of the buffer indicates, how many bytes can be stored at the receiver. The window of flow control is adapted to this value.**

The TCP Header

- **Flags:**
  - URG: URGENT. When using the Urgent pointer, e.g., for special keyboard input (Ctrl-C).
  - ACK: This bit is set, if an acknowledgement is sent.
  - PSH: PUSH. Direct forwarding of the data, no more waiting for further data.
  - RST: RESET. Reset a connection, e.g., during a host crash or a connecting rejection. ("Generally problems arise when a segment with set RST bit is received.")
  - SYN: set to 1 for the establishment of a connection.
  - FIN: set to 1 for the termination of a connection.
- **Urgent pointer: indicates, at which posi**
  data ends (byte offset of the current se
- **Option: additional functions:**
  - Negotiation of a window scale
  - Use of e.g. Selective Repeat instead of Go-B
  - Indication of the Maximum Segment Size (M
    field



The TCP Header

- **Flags:**
  - URG: URGENT. When using the Urgent pointer, e.g., for special keyboard input (Ctrl-C).
  - ACK: This bit is set, if an acknowledgement is sent.
  - PSH: PUSH. Direct forwarding of the data, no more waiting for further data.
  - RST: RESET. Reset a connection, e.g., during a host crash or a connecting rejection. (“Generally problems arise when a segment with set RST bit is received.”)
  - SYN: set to 1 for the establishment of a connection.
  - FIN: set to 1 for the termination of a connection.
- **Urgent pointer: indicates, at which position in the data field the urgent data ends (byte offset of the current sequence number).**
- **Option: additional functions:**
  - Negotiation of a window scale
  - Use of e.g. Selective Repeat instead of Go-Back-n in the event of an error
  - Indication of the Maximum Segment Size (MSS) to determine the size of the data field

TCP Pseudo Header

- **Checksum: serves among other things for the verification that the packet was delivered to the correct device.**
  - The checksum is computed using a pseudo header. The pseudo header is placed in front of the TCP header, the checksum is computed from both headers (the checksum field is set to 0).
    - The checksum is computed as the 1-complement of the sum of all 16-bit words of the segment including the pseudo header.
  - The receiver also places the pseudo header in front of the received TCP header and executes the same algorithm (the result must be 0).

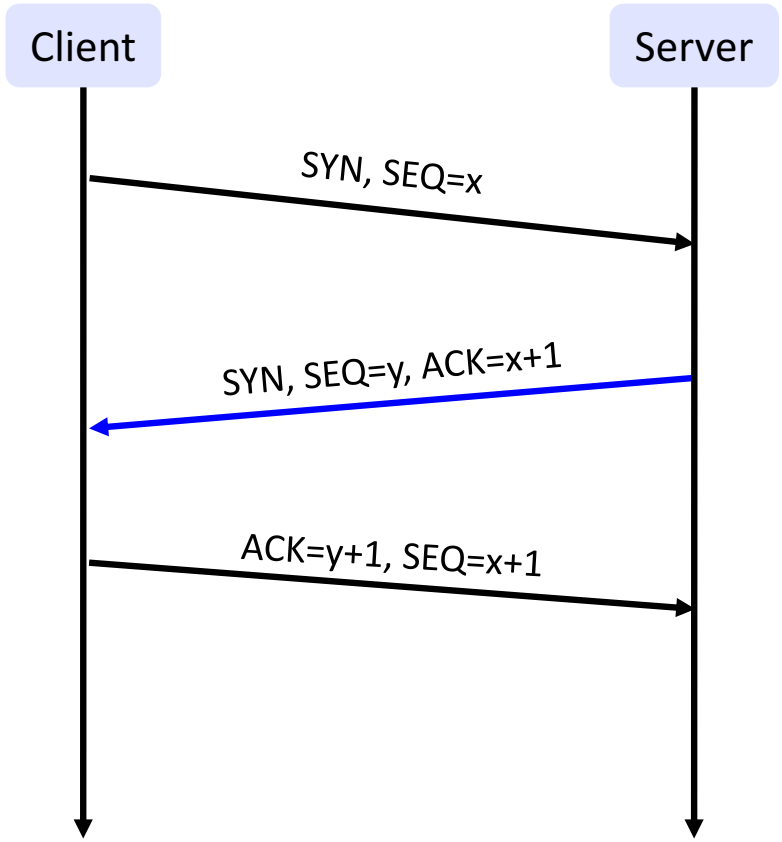
Source address (IP)		
Destination address (IP)		
00000000	Protocol = 6	Length of the TCP segment

Transport Control Protocol (TCP)

Connection Management

TCP Connection Management:

1. Connection Establishment

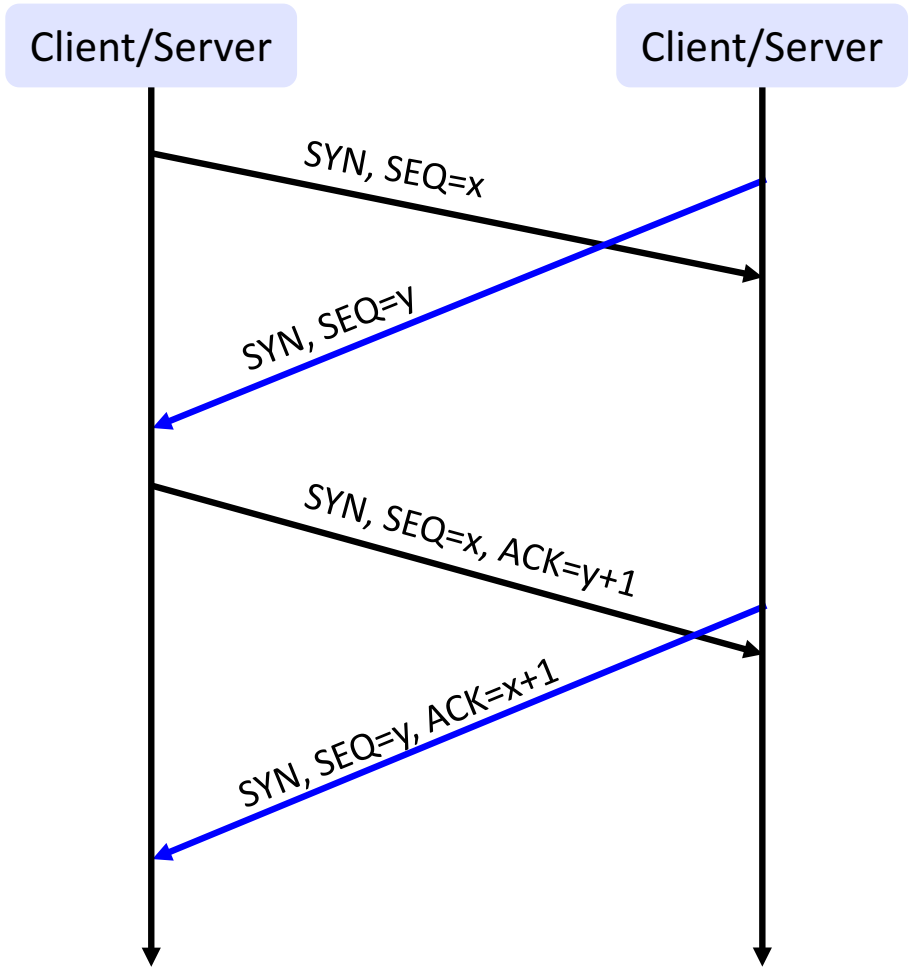


Three Way Handshake

- The server waits for connection requests using LISTEN and ACCEPT.
- The client uses the CONNECT operation by indicating IP address, port number, and maximally acceptable segment size.
- CONNECT sends a SYN.
- If the destination port of the CONNECT is identical to the port number on which the server waits, the connection is accepted, otherwise it is rejected with RST.
- The server also sends a SYN to the client and acknowledges at the same time the receipt of the client's SYN segment.
- The client sends an acknowledgement for the SYN segment of the server. The connection is established.

TCP Connection Management:

Irregular Connection Establishment

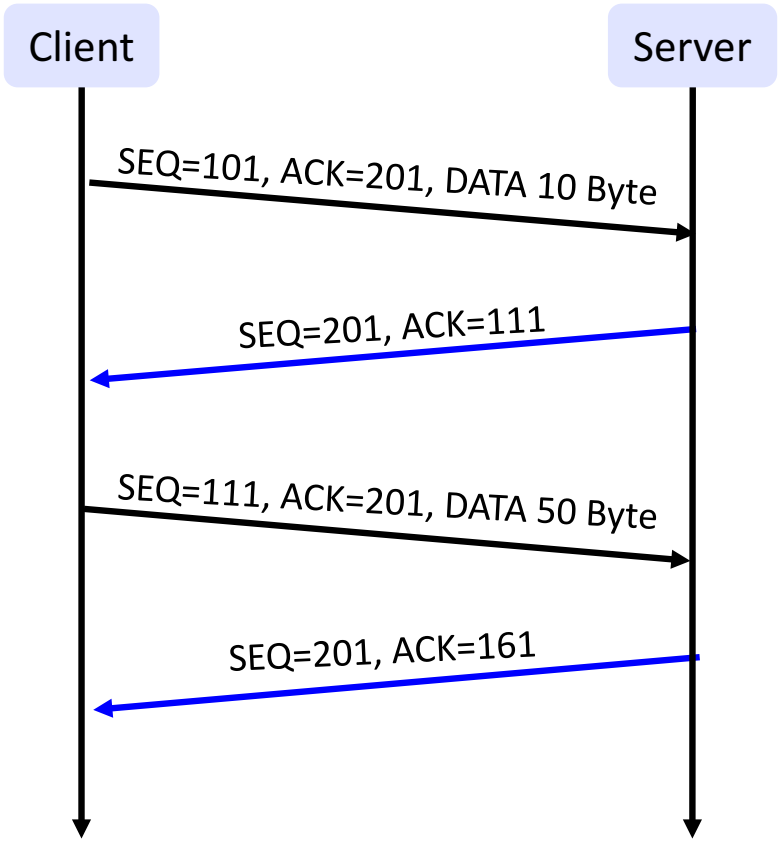


- Possibly, two computers at the same time try to establish a connection between the same sockets.
- Connections are characterized by their endpoints; only **one** connection is established between a pair of endpoints.
- The endpoints are uniquely characterized by:

  (IP Address$_1$, Port$_1$, IP Address$_2$, Port$_2$)

TCP Connection Management:

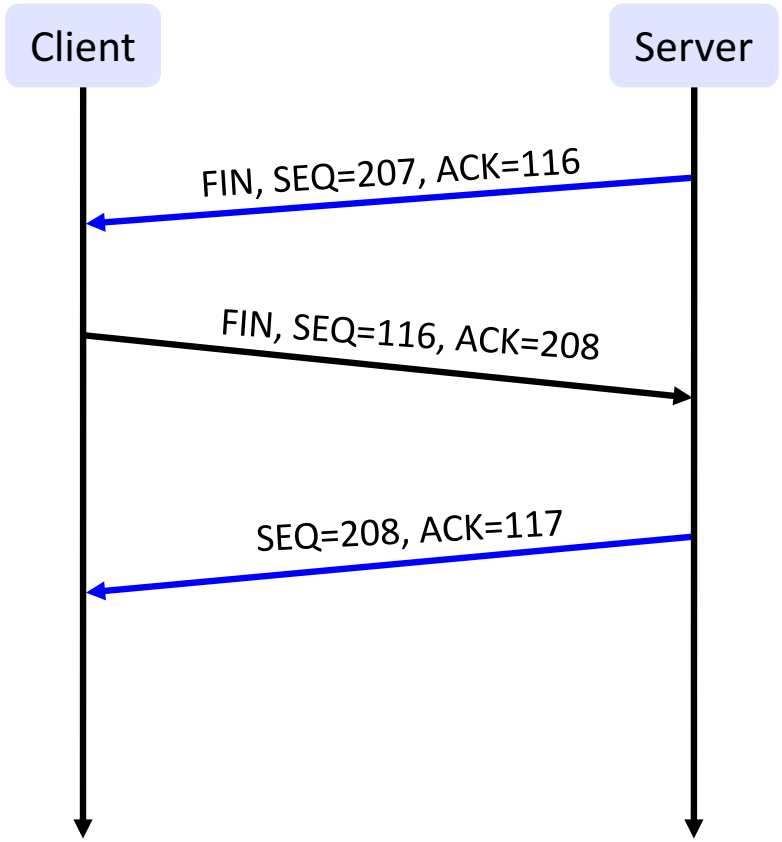
2. Data Transmission



- Full-duplex connection
- Split the byte stream into segments.
  - Usual sizes are 1500, 536, or 512 byte
    - ➢ IP fragmentation is avoided.
  - All hosts have to accept TCP segments of 536 byte +20 byte = 556 byte
- Usual acknowledgement mechanism:
  - All segments up to ACK-1 are confirmed. If the sender has a timeout before an ACK, he repeats the sending.
- Usual procedure for repeating:
  - Go-Back-N or Selective Repeat

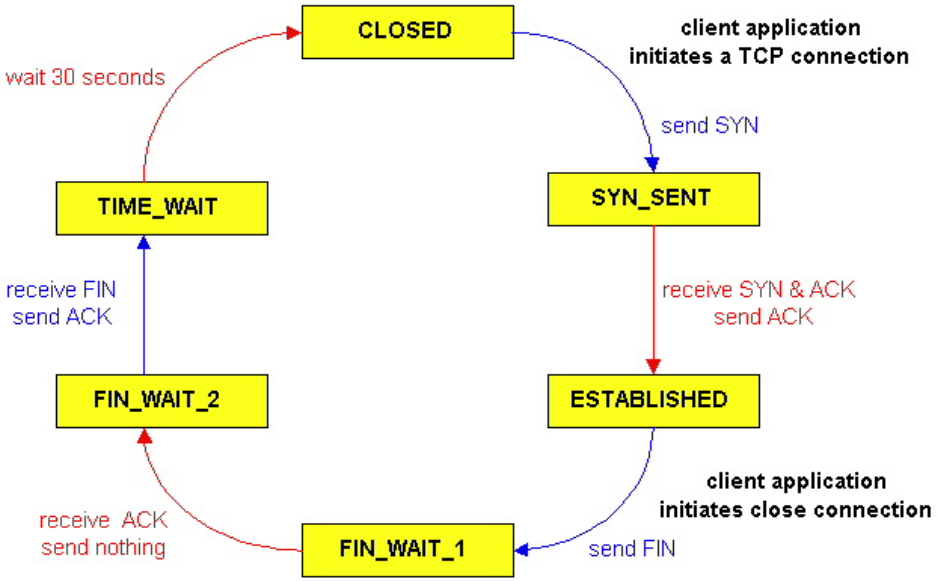
TCP Connection Management:

3. Connection Termination



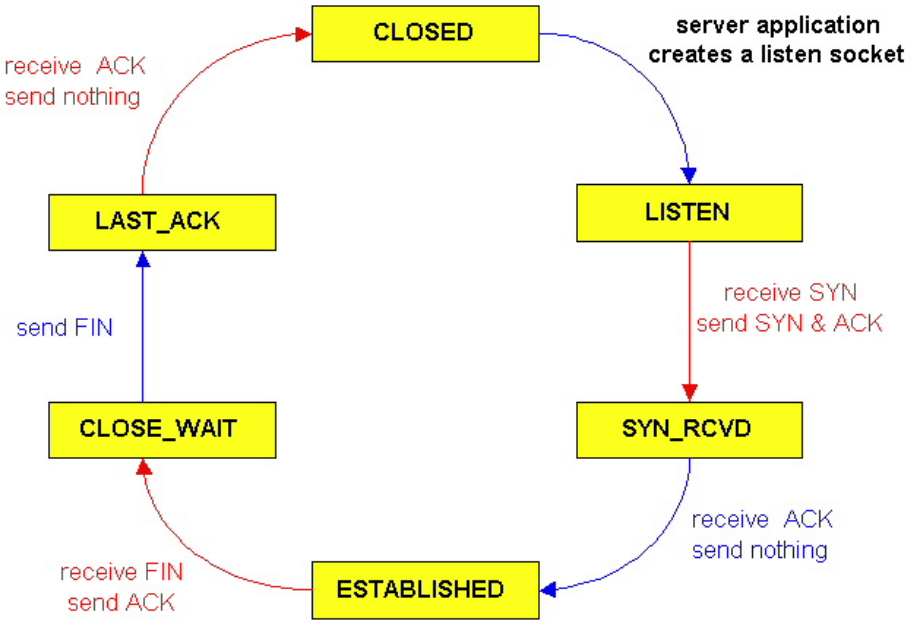
- Termination as two simplex connections
- Send a FIN segment
- If the FIN segment is confirmed, the direction is “switched off”. The opposite direction remains however still open, data can be still further sent.
  - Half-open connections!
- Use of timers to protect against packet loss.

TCP Connection Management

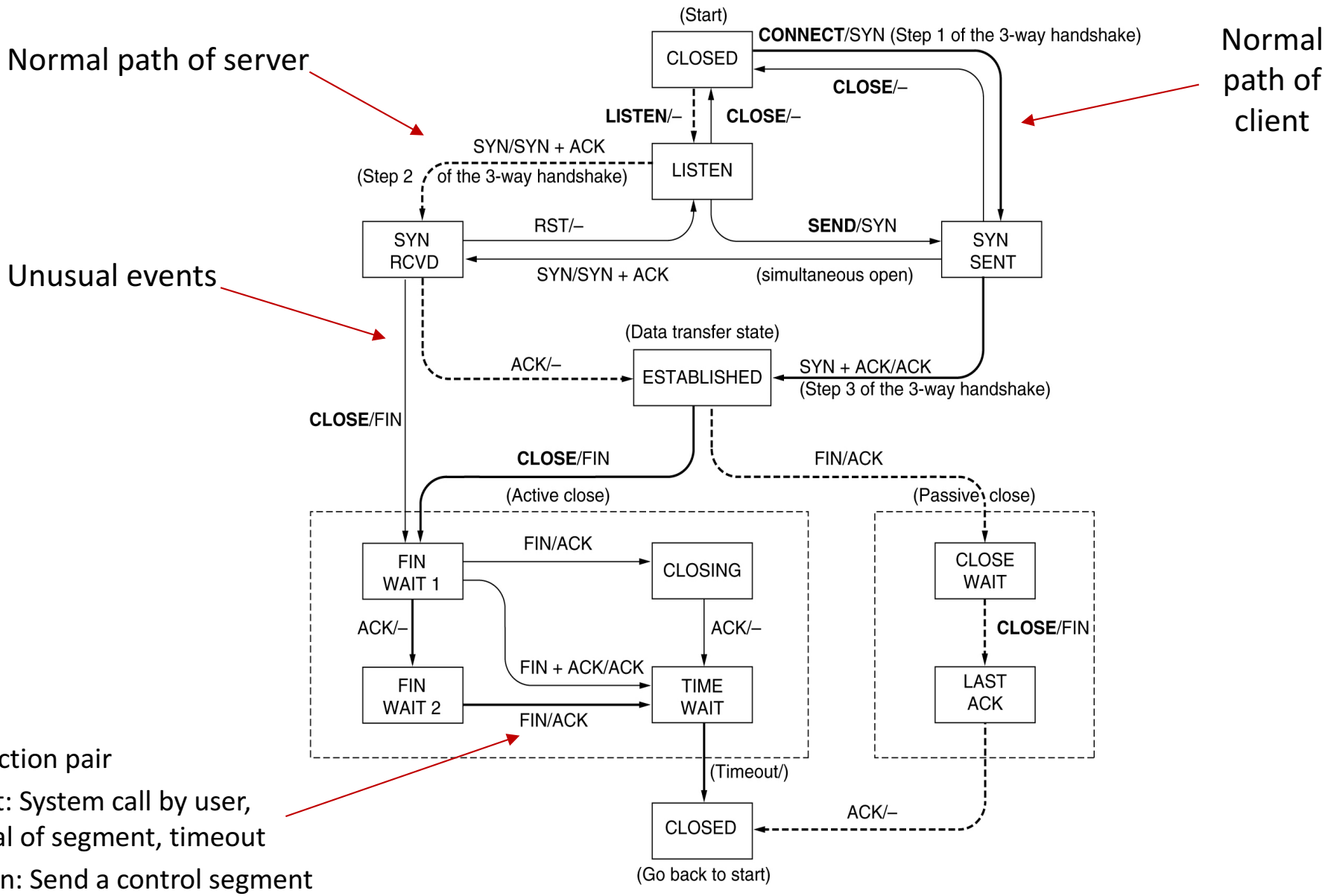


TCP client lifecycle

TCP server lifecycle



The Entire TCP Connection



Event/action pair

- Event: System call by user, arrival of segment, timeout
- Action: Send a control segment

States during a TCP Session

State	Description
CLOSED	No active communications
LISTEN	The server waits for a connection request
SYN RCVD	A connection request was received and processed, wait for the last ACK of the connection establishment
SYN SENT	Application began to open a connection
ESTABLISHED	Connection established, transmit data
FIN WAIT 1	Application starts a connection termination
FIN WAIT 2	The other side confirms the connection termination
TIME WAIT	Wait for late packets
CLOSING	Connection termination
CLOSE WAIT	The other side initiates a connection termination
LAST ACK	Wait for late packets

Transport Control Protocol (TCP)

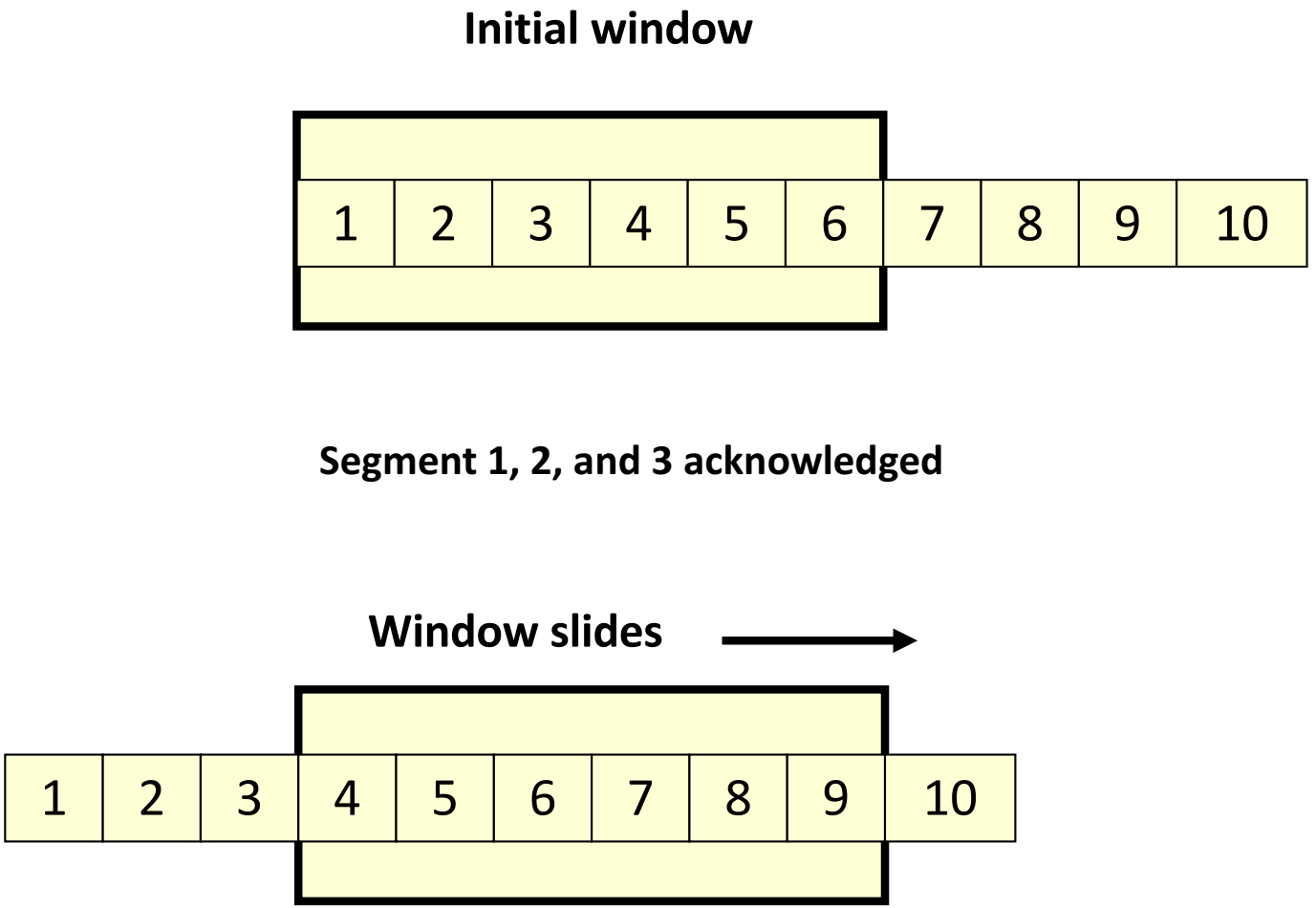
Flow Control

Flow Control: Sliding Window

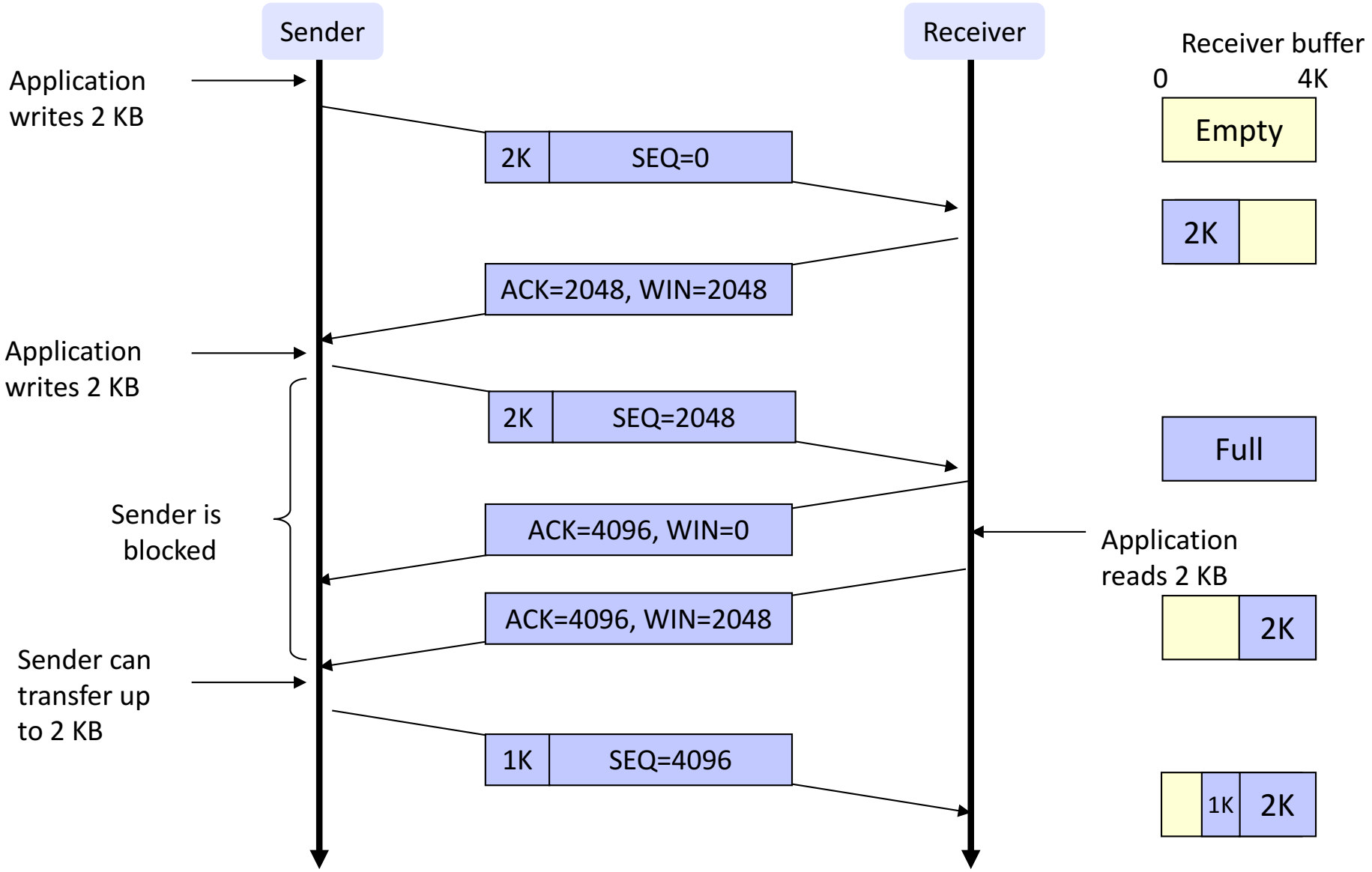
To provide reliable data transfer, as on Layer 2, a sliding window mechanism is used.

Differences:

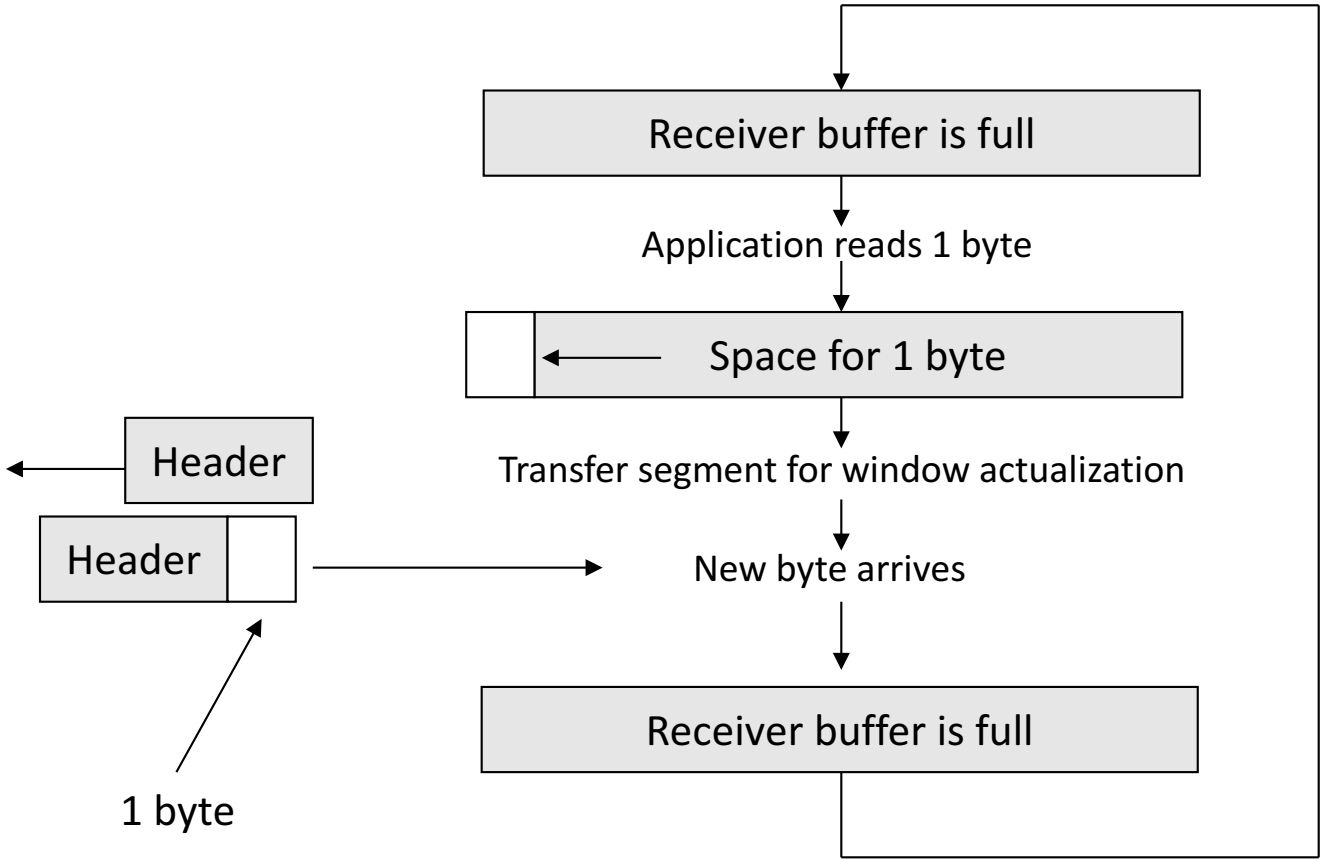
- Sender sends **bytes** according to the window size
- Window is shifted by **n** byte as soon as an ACK for **n** bytes arrives
- Exception: Urgent data (URGENT flag is set) are sent immediately
- Characteristic: the window size can be changed during the transmission phase



Sliding Window: Example

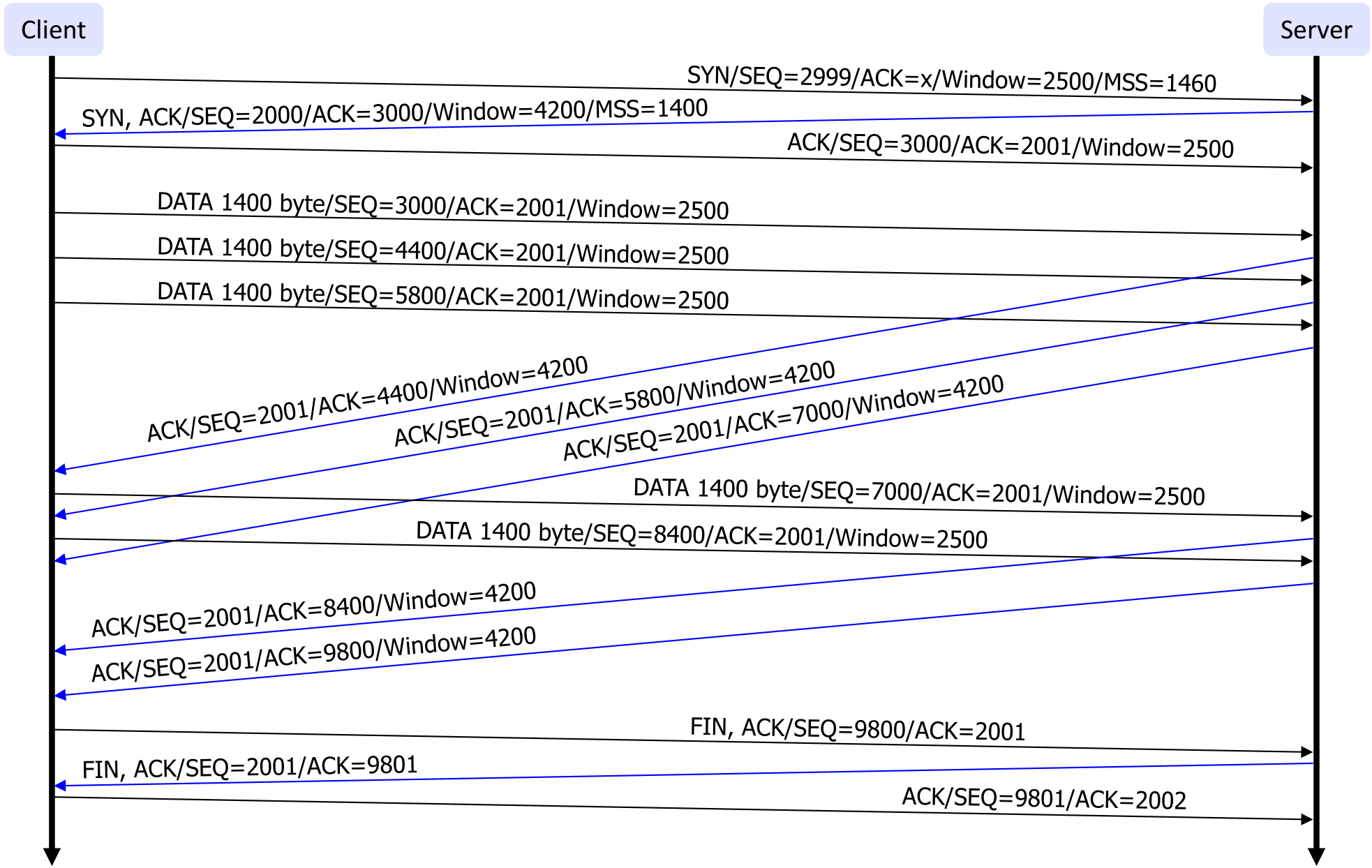


»Silly Window« Syndrome

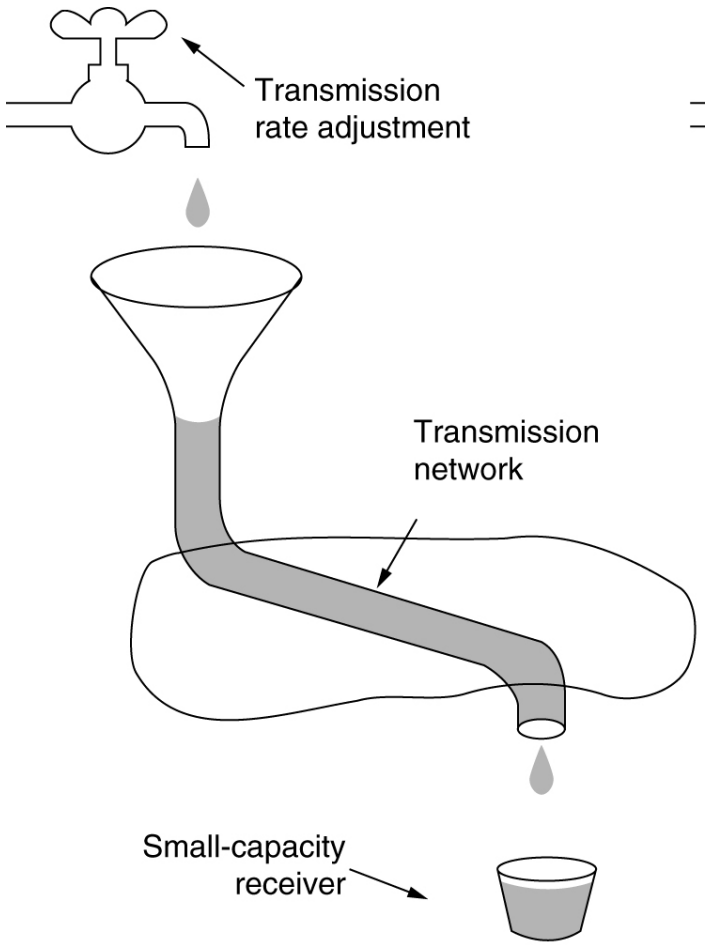


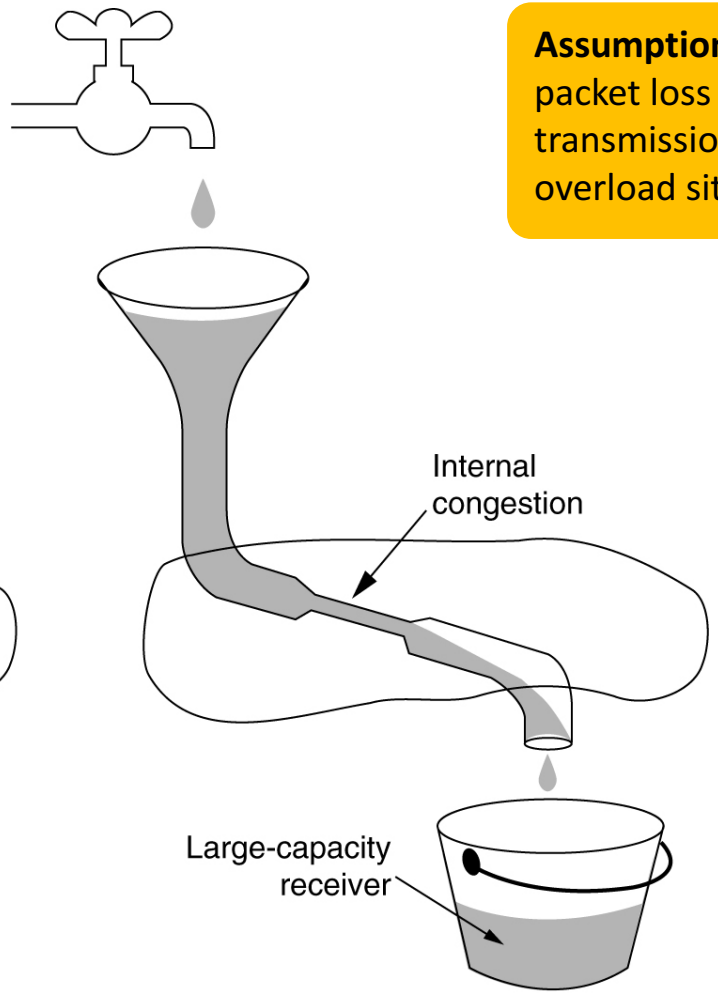
Solution of Clark:
The receiver must wait with the next window actualization until the receiver buffer again is reasonably empty.

The whole TCP Session



Flow Control: Network Bottlenecks





Assumption:
packet loss is rarely because of transmission errors, rather because of overload situations.

Necessary:

Congestion Control

Capacity of the receiver:

Flow Control Window

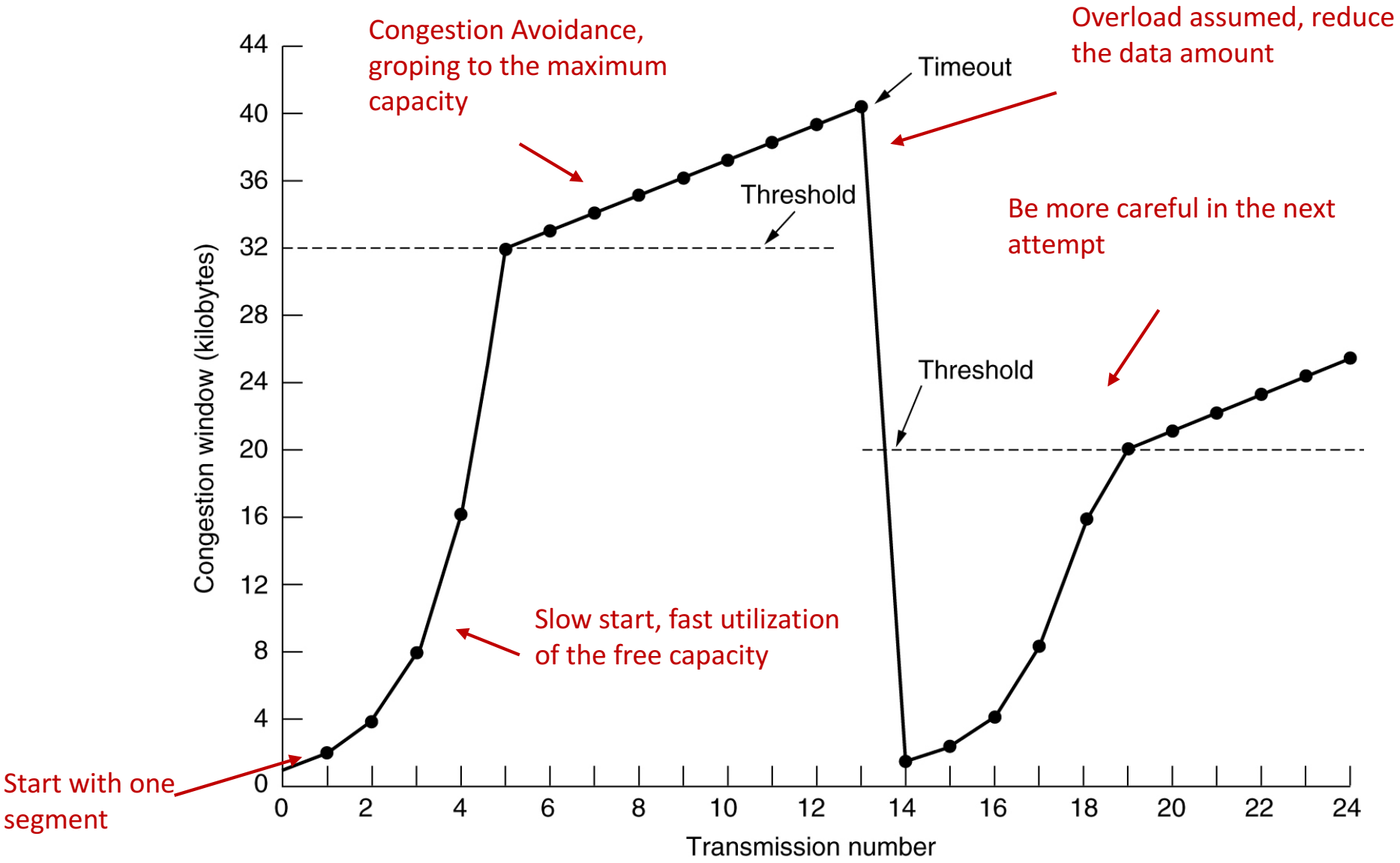
Capacity of the network:

Congestion Window

Control Algorithm in the Internet

- **Each sender maintains two windows for the number of bytes which can be sent:**
  - Flow Control Window: granted receiver buffer
  - Congestion Window: »network granted« capacity (**cwnd**)
- **Minimum of both windows is the number of bytes which can be sent maximally**
- **With connection establishment, the sender initializes the congestion window to the size of the maximum segment (MSS, the MTU the sender is able to send)**
- **Maximum segment is sent**
- **If an acknowledgement arrives before timeout, double the congestion window (Slow Start Algorithm), otherwise reset it to the initial value. Thus a "grope" takes place up to the transmission capacity.**
- **Enlargement stops with reaching the flow control window**
- **Refinement by introduction of a threshold value ssthresh (at the beginning 64 Kbyte):**
  - Only linear enlargement by one maximum segment size (**Congestion Avoidance**)
  - With a timeout the threshold value is put back **to half of the maximum window size** reached before

Control Algorithm in the Internet



Fast Retransmit and Fast Recovery

Slow Start is not well suited when only a single packet is lost…

Fast Retransmit

- The receiver should send a duplicate ACK immediately when an out-of-order segment arrives
- When the sender has received 3 duplicate ACKs, it performs a retransmission of the segment which seems to be missing. Hopefully, the acknowledgement for the repeated segment arrives before a timeout for this segment occurs.

Fast Retransmit has to be enhanced to be useful in overload situations…

Fast Recovery

- When the third ACK is received, reduce **ssthresh** to max(current size/2, 2×MSS)
- Retransmit the missing segment, set **cwnd** to ssthresh + 3×MSS
- For each more duplicated ACK, increment **cwnd** by MSS
- This reduces **cwnd** by the amount of lost segments to adapt to the network situation
- If the new **cwnd** (and the receiver's buffer) allows, send a segment
- When the next (normal) ACK arrives, set **cwnd** to **ssthresh** to go on normally

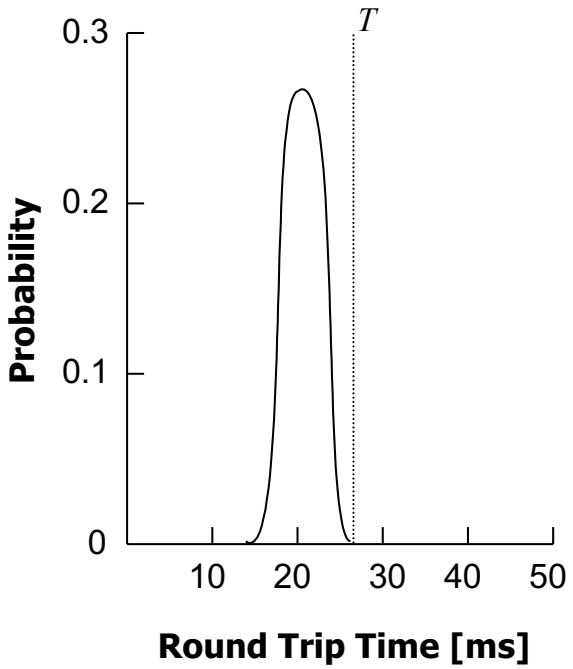
Transport Control Protocol (TCP)

Timer Management

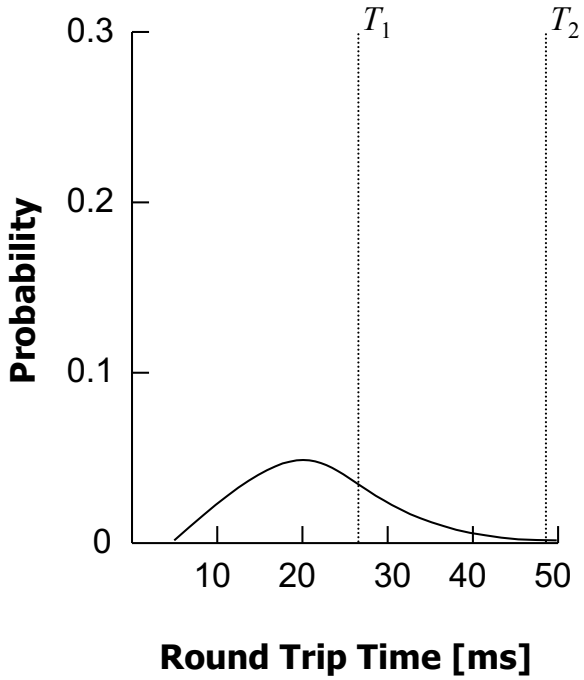
Timer Management with TCP

TCP uses several timers:

- **Retransmission timer** (for repeating a transmission)
  - But: how to select the timer value?
  - Probability density of the time till an acknowledgement arrives:



Data Link Layer

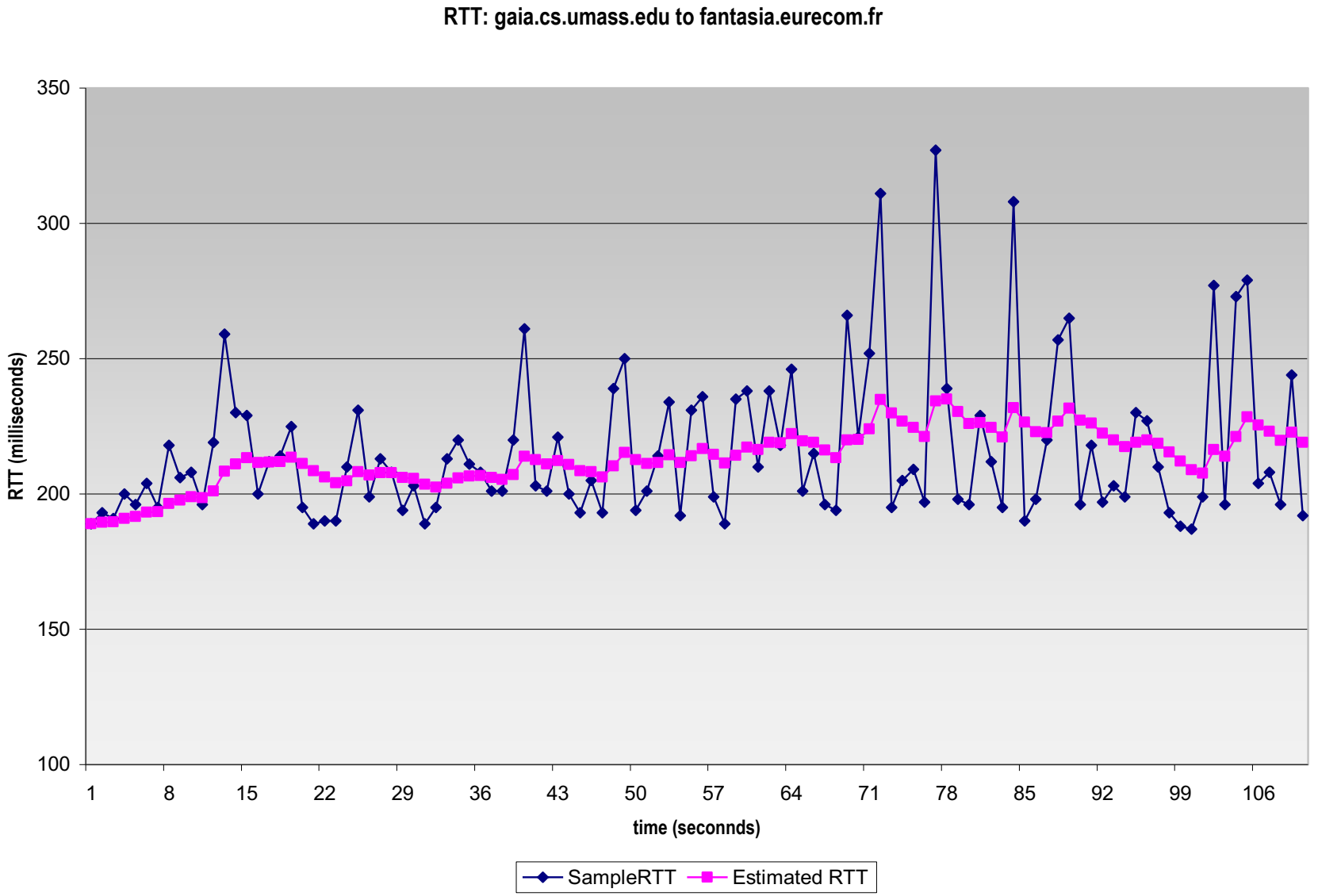


Transport Layer

Problem on the transport layer:

- T_1 is too small: too many retransmissions
- T_2 is too large: inefficient for actual packet loss

Example RTT estimation



Timer Management with TCP

- **How to set TCP timeout value?**
  - Longer than RTT
    - but RTT varies
  - Too short: premature timeout
    - unnecessary retransmissions
  - Too long: slow reaction to segment loss
- **How to estimate RTT?**
  - SampleRTT: measured time from segment transmission until ACK receipt
    - ignore retransmissions
  - SampleRTT will vary, want estimated RTT “smoother”
    - average several recent measurements, not just current SampleRTT

Retransmission Timer

- **Solution: dynamic algorithm, which adapts the timer by current measurements of the network performance.**
- **Algorithm of Jacobson (1988)**
  - TCP manages a variable RTT (Round Trip Time) for each connection
  - RTT is momentarily the best **estimation** of the round trip time
  - When sending a segment, a timer is started which measures the time the acknowledgement needs and initiates a retransmission if necessary.
  - If the acknowledgement arrives before expiration of the timer (after a time unit $sampleRTT$), RTT is updated:

$$RTT = \alpha \cdot RTT + (1 - \alpha) sampleRTT$$

  - α is a smoothing factor, typically 0.875
  - The choice of the timeout is based on RTT

$$Timeout = \beta \bullet RTT$$

  - At the beginning, β was chosen as 2, but this was too inflexible

Retransmission Timer

- **Improvement (Jacobson): set timer proportionally to the standard deviation of the arrival time of acknowledgements**
  - Computation of standard deviation:

$$devRTT = \gamma \bullet devRTT + (1 - \gamma) \bullet |RTT - sampleRTT|$$

  - Typically $\gamma = 0.75$

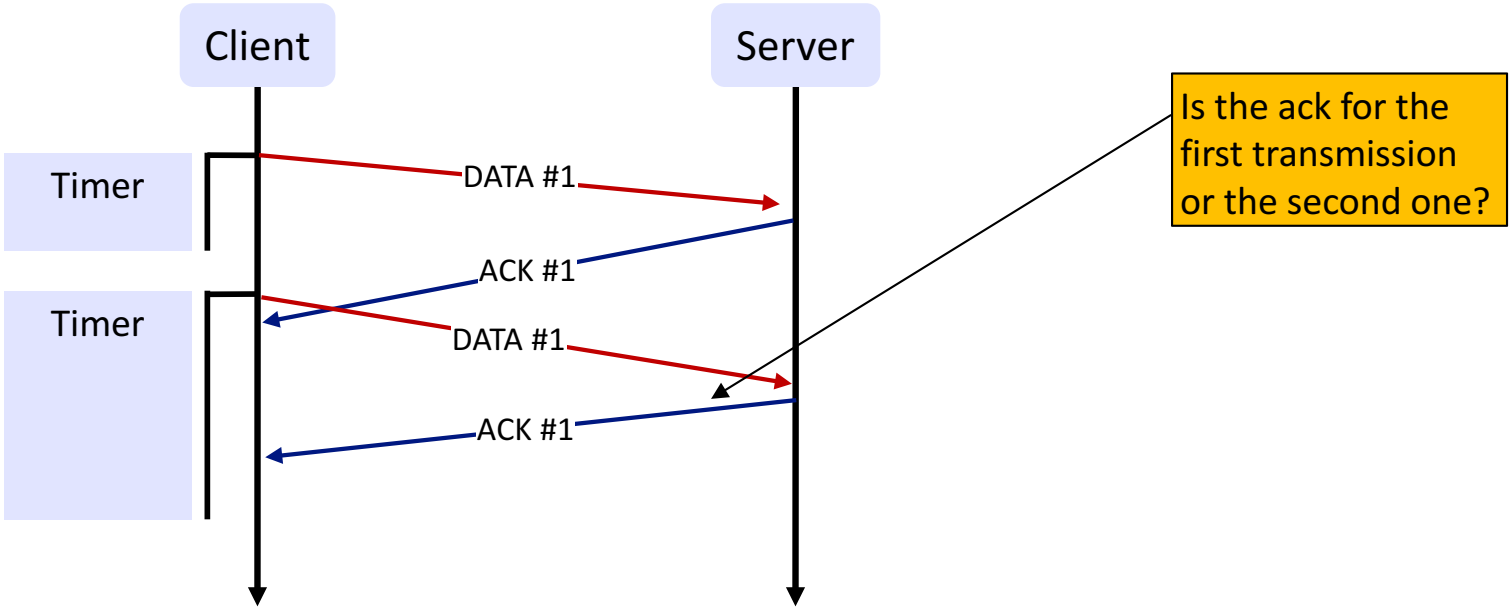
  - Standard timeout interval:

$$Timeout = RTT + 4 \bullet devRTT$$

  - The factor 4 was determined on the one hand by trying out, on the other hand because it is fast and simple to use in computations.

Retransmission Timer

- **Karn's Algorithm**
  - Alternative, very simple proposal, which is used in most TCP implementations
  - Do not update RTT on any segments that have been retransmitted. The timeout is doubled on each failure until the segments get through.



Other Timers

Persistence timer

- Prevents a deadlock with a loss of the buffer release message of a receiver
- With expiration of the timer, the sender transfers a test segment. The response to this transmission contains the current buffer size of the receiver. If it is still zero, the timer is started again.

Keep-alive timer

- If a connection is inactive for a longer time, at expiration of the timer it is examined whether the other side is still living.
- If no response is given, the connection is terminated.
- Disputed function, not often implemented.

Time Wait timer

- During the termination of a connection, the timer runs for the double packet life time to be sure that no more late packets arrive.

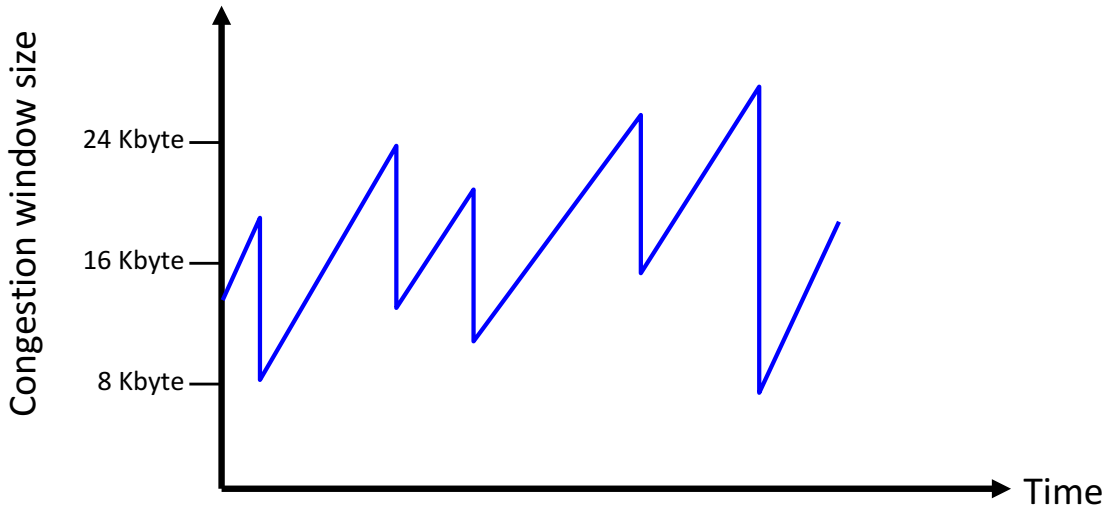
Transport Control Protocol (TCP)

Congestion control

TCP congestion control

- **Approach: increase transmission rate (window size), probing for usable bandwidth, until loss occurs**
  - Additive increase: increase **CongWin** by 1 MSS every RTT until loss detected
  - Multiplicative decrease: cut **CongWin** in half after loss

Saw tooth behavior: probing for bandwidth



TCP Congestion Control: details

- **Sender limits transmission:**

  LastByteSent - LastByteAcked ≤ CongWin

- **Roughly:**

$$\text{rate} = \frac{\text{CongWin}}{\text{RTT}} \text{ Bytes/sec}$$

- **CongWin is dynamic, function of perceived network congestion**

- **How does sender perceive congestion?**
  - Loss event = timeout or 3 duplicate acks
- **TCP sender reduces rate (CongWin) after loss event**
- **Three mechanisms:**
  - AIMD
  - Slow start
  - Conservative after timeout events

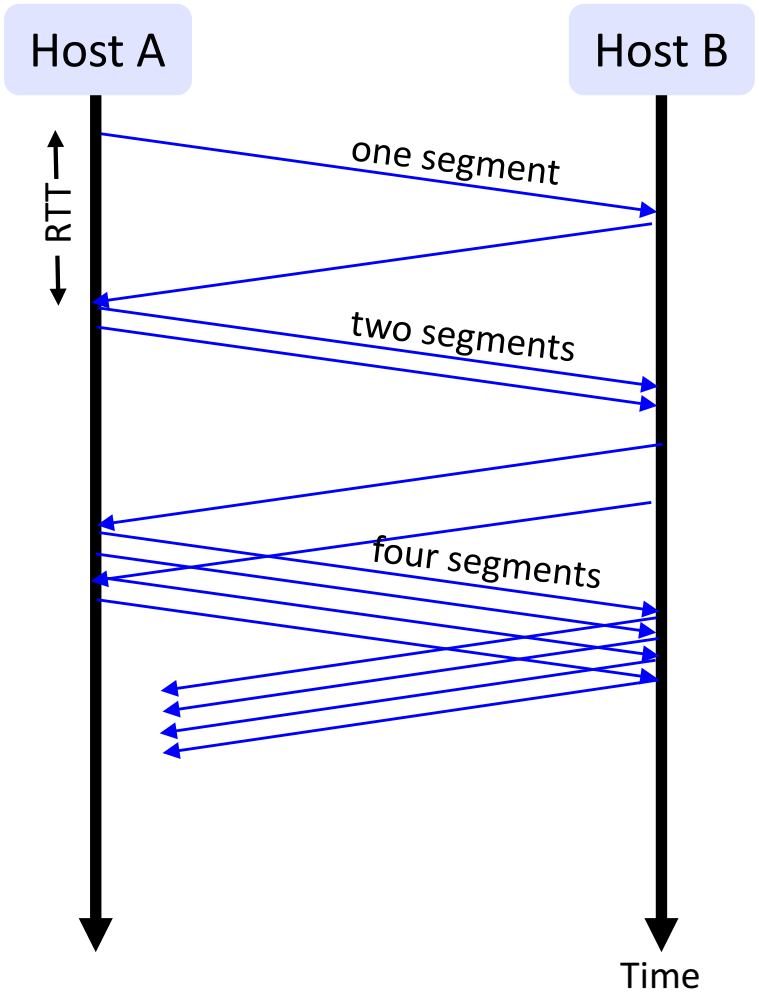
TCP Slow Start

- **When connection begins, CongWin = 1 MSS**
  - Example: MSS = 500 bytes & RTT = 200 msec
  - Initial rate = 20 kbps
- **Available bandwidth may be >> MSS/RTT**
  - Desirable to quickly ramp up to respectable rate
- **When connection begins, increase rate exponentially fast until first loss event**

TCP Slow Start (more)

- **When connection begins, increase rate exponentially until first loss event:**
  - Double CongWin every RTT
  - Done by incrementing CongWin for every ACK received

- **Summary:**
  - Initial rate is slow but ramps up exponentially fast



Refinement: Inferring loss

- **Philosophy**
  - 3 dup ACKs indicates network capable of delivering some segments
  - timeout indicates a “more alarming” congestion scenario

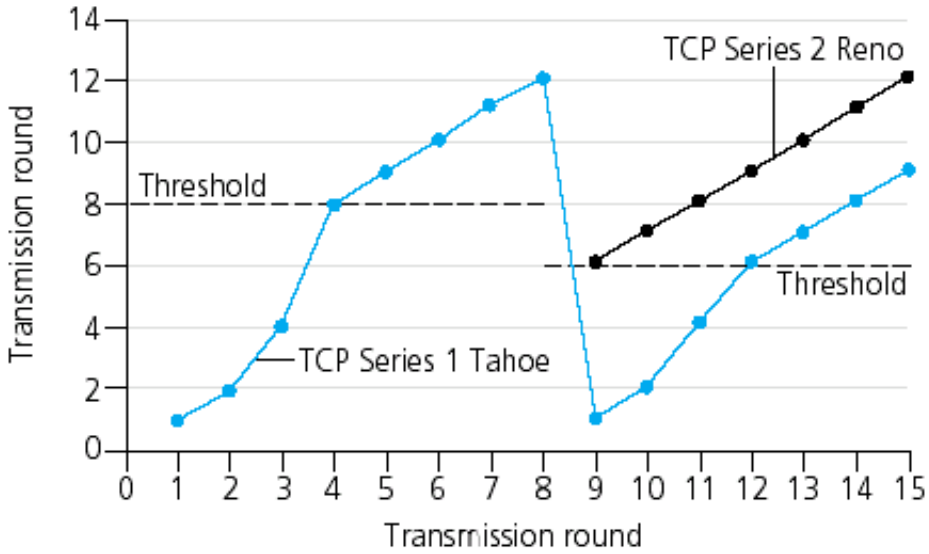
- **After 3 dup ACKs**
  - CongWin is cut in half
  - window then grows linearly

- **But after timeout event**
  - CongWin instead set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly

Refinement

- **Questions**
  - Q: When should the exponential increase switch to linear?
  - A: When CongWin gets to 1/2 of its value before timeout.

- **Implementation:**
  - Variable Threshold
  - At loss event, Threshold is set to 1/2 of CongWin just before loss event



Summary: TCP Congestion Control

- When CongWin is below Threshold, sender in slow-start phase, window grows exponentially.
- When CongWin is above Threshold, sender is in congestion-avoidance phase, window grows linearly.
- When a triple duplicate ACK occurs, Threshold set to CongWin/2 and CongWin set to Threshold.
- When timeout occurs, Threshold set to CongWin/2 and CongWin is set to 1 MSS.

TCP sender congestion control

State	Event	TCP Sender Action	Commentary
Slow Start (SS)	ACK receipt for previously unacked data	CongWin = CongWin + MSS, If (CongWin > Threshold) set state to "Congestion Avoidance"	Resulting in a doubling of CongWin every RTT
Congestion Avoidance (CA)	ACK receipt for previously unacked data	CongWin = CongWin+MSS * (MSS/CongWin)	Additive increase, resulting in increase of CongWin by 1 MSS every RTT
SS or CA	Loss event detected by triple duplicate ACK	Threshold = CongWin/2, CongWin = Threshold, Set state to "Congestion Avoidance"	Fast recovery, implementing multiplicative decrease. CongWin will not drop below 1 MSS.
SS or CA	Timeout	Threshold = CongWin/2, CongWin = 1 MSS, Set state to "Slow Start"	Enter slow start
SS or CA	Duplicate ACK	Increment duplicate ACK count for segment being acked	CongWin and Threshold not changed

Transport Control Protocol (TCP)

TCP Throughput

TCP throughput

- **What's the average throughout of TCP as a function of window size and RTT?**
  - Ignore slow start
- **Let W be the window size when loss occurs.**
- **When window is W, throughput is W/RTT**
- **Just after loss, window drops to W/2, throughput to W/2RTT.**
- **Average throughout: .75 W/RTT**

TCP Futures: TCP over “long, fat pipes”

- **Example: 1500 byte segments, 100ms RTT, want 10 Gbps throughput**
- **Requires window size W = 83,333 in-flight segments**
- **Throughput in terms of loss rate:**

$$\frac{1.22 \cdot MSS}{RTT\sqrt{L}}$$

  - ➢ L = $2 \cdot 10^{-10}$
  - New versions of TCP for high-speed

Summary

- **The TCP/IP reference model has only two protocols on the transport layer**
  - UDP for connectionless, but lightweight protocol
  - TCP for connection oriented and reliable communication
    - Connection establishment
    - Flow control / congestion control
    - Connection termination
    - Fairness