Rechnernetze - Computer Networks

Lecture 10: End-to-end Protocols
UDP, TCP

Prof. Dr.-Ing. Markus Fidler

Institute of Communications Technology
Leibniz Universität Hannover

June 21, 2024

Outline

Overview

User datagram protocol (UDP)
- Multiplexing and Demultiplexing

Transmission control protocol (TCP)
- Connection Management
- Reliable data transfer
- Flow control

Overview

Task of transport layer protocols

- logical communication for the application layer
- end-to-end communication, not implemented on routers

Typical transport layer functionality (need not all be implemented)

- segmentation of messages
- multiplexing/demultiplexing
- reliable data transfer

Dominant transport protocols in the Internet

- user datagram protocol (UDP)
- transmission control protocol (TCP)

Overview continued

At the network layer IP offers an unreliable best-effort service

- does not guarantee delivery of data
- does not guarantee integrity of data

Transport layer services of UDP

- multiplexing and demultiplexing using port fields
- error checking using checksums
- unreliable data transfer, connectionless
- unregulated transmission

Transport layer services of TCP

- multiplexing and demultiplexing using port fields
- error checking using checksums
- reliable data transfer, connection-oriented
- flow and congestion control
- segmentation and reassambly

User datagram protocol (UDP)

Task

- multiplexing and demultiplexing applications
- error checking
- unreliable simplex data transfer

Specification

- defined in RFC 768, Internet standard 6

Characteristics

- connectionless
- no segmentation of (too large) messages
- unregulated transmission (no flow/congestion control)

Why UDP?

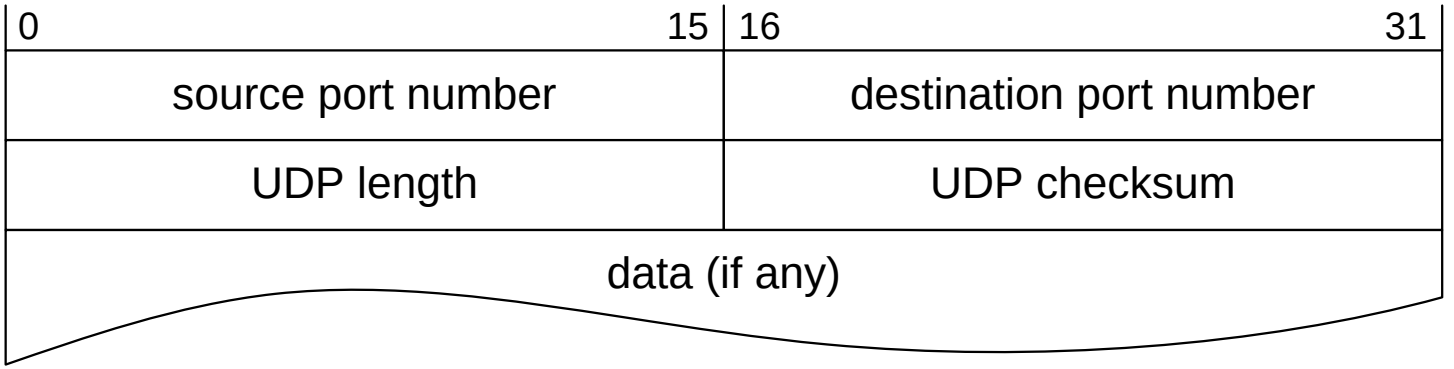
Although TCP provides much better service, i.e. reliable data transfer, UDP's minimalist functionality has advantages

- no delays due to connection establishment
- no memory requirements for
  - connection state
  - flow and congestion control
  - send and receive buffers for unacknowledged data
- unregulated send rate
- little overhead for UDP header (8 byte)

Examples of applications that use UDP are

- routing information protocol (RIP), sends periodic messages
- simple network management protocol (SNMP), may not afford congestion control e.g. in case of overload
- real-time apps that do not work well with congestion control

UDP Header



- source and destination port numbers for multiplexing and demultiplexing
- UDP length is the length of UDP header and data part (in fact the information is redundant, it is also included in IP)
- checksum for error detection, but not correction

Multiplexing and demultiplexing

Application multiplexing and demultiplexing

- IP delivers data between end systems that are identified by their IP address
- UDP delivers data between processes running on the end systems that are identified by their UDP port numbers

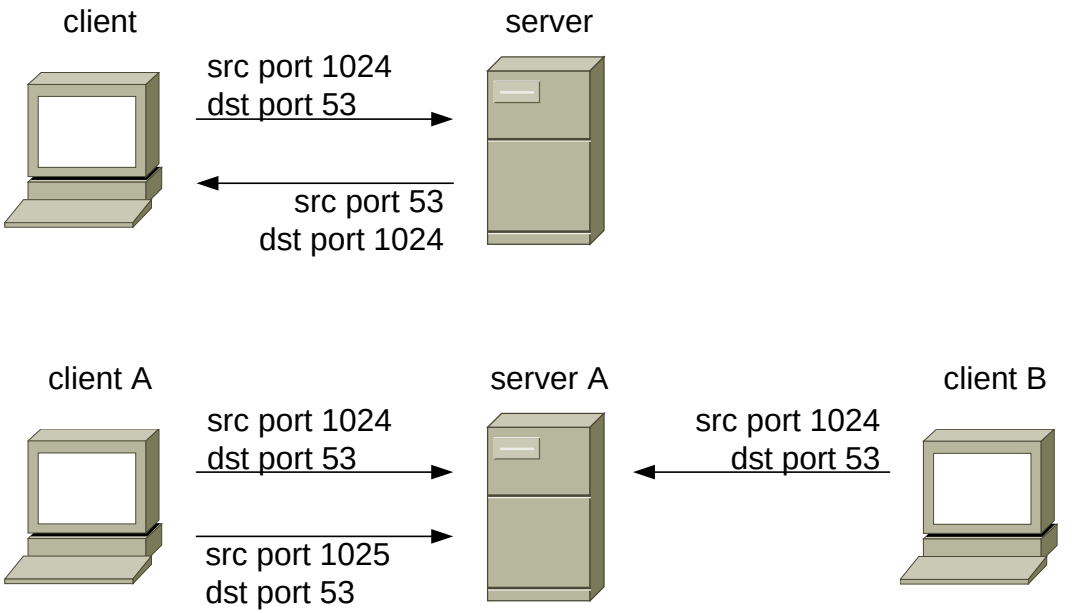
Port numbers

- the client uses the specific **destination port** number of the type of application
- the client chooses a **source port** number to identify the specific application process

Numbering

- 16 bit numbers ranging from 0 to 65535
- 0 to 1023 are well-known port numbers that are reserved for specific applications, see RFC 1700

Multiplexing and demultiplexing example



Two clients may access a server with the same combination of source and destination port numbers

- the IP addresses allow distinguishing the hosts
- on each client the combination of port numbers is unique
- the triple of client IP address, source and destination port identifies the corresponding application process

Outline

Transmission control protocol (TCP)

Task

- multiplexing and demultiplexing applications
- error checking
- reliable full-duplex data transfer

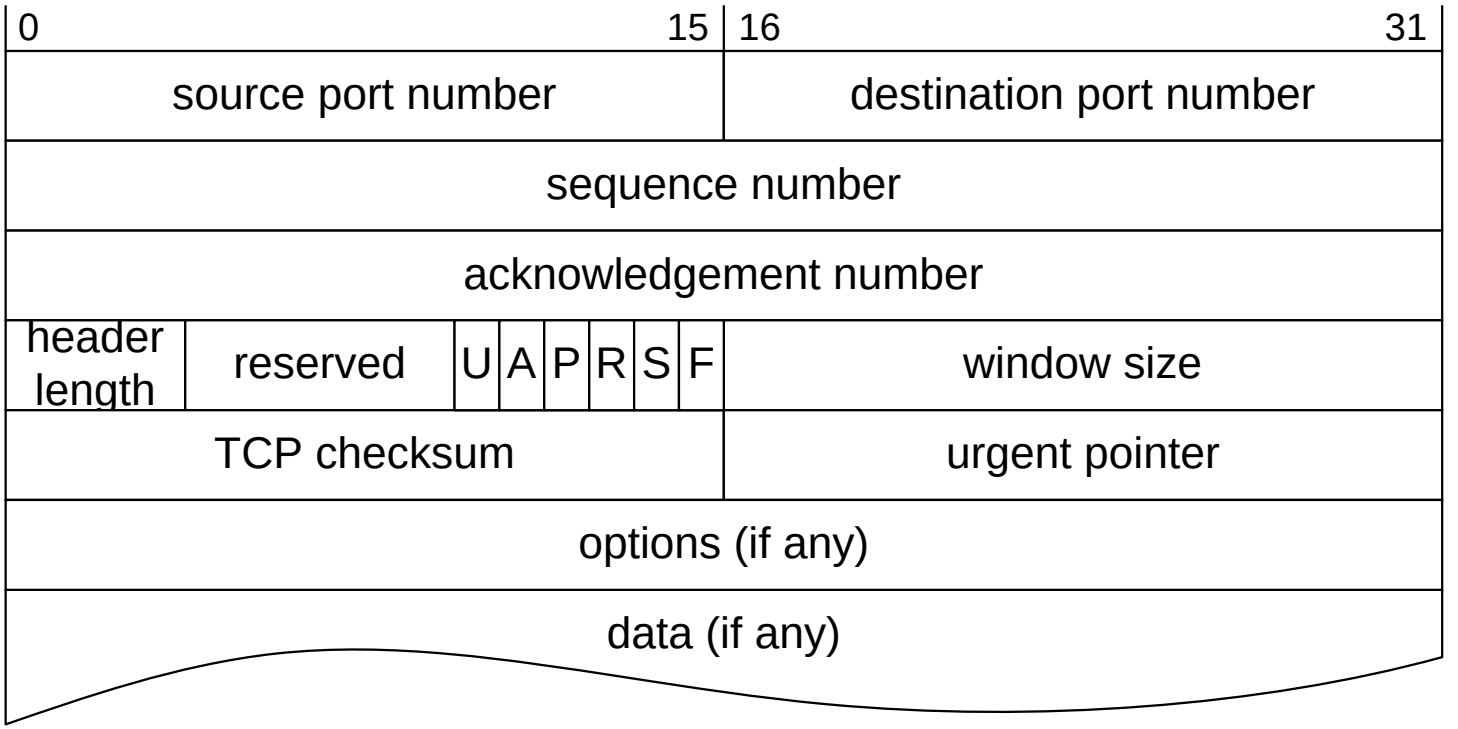
Specification

- defined in RFC 793, Internet standard 7, 1981

Characteristics

- connection-oriented
- segmentation and reassembly of messages
  TCP packets are called segments
- flow control
- congestion control for fair resource sharing

TCP Header



- port numbers for multiplexing and demultiplexing
- checksum for error detection, retransmissions for correction

Connection Management

Goal of connection management

- verify that communication partners are reachable
- initialization of communication partners
- release resources once communication is terminated

Phases

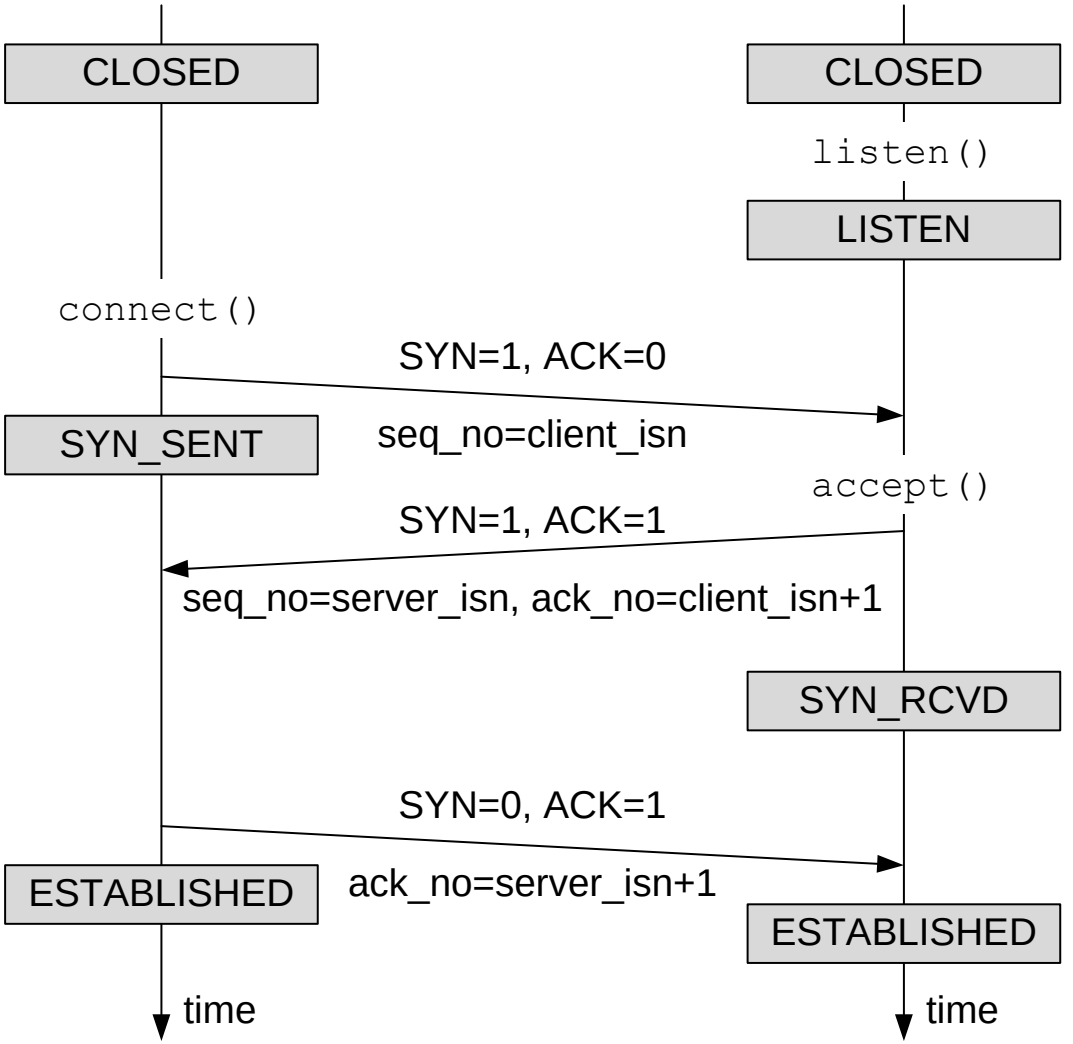
- connection establishment
  - three way handshake
  - synchronization of initial sequence numbers
- data transfer
  - data
  - acknowledgements
- connection release
  - four way close
  - deallocation of resources

Connection establishment

Three way handshake

- Connection request from the client
  - SYN = 1, ACK = 0
  - client's initial sequence number `client_isn` = random number
- Connection grant from the server
  - SYN = 1, ACK = 1
  - server's initial sequence number `server_isn` = random number
  - acknowledgement number = `client_isn` + 1
- Acknowledge connection grant from the client
  - SYN = 0, ACK = 1
  - acknowledgement number = `server_isn` + 1
  - may already contain user data

Connection establishment continued



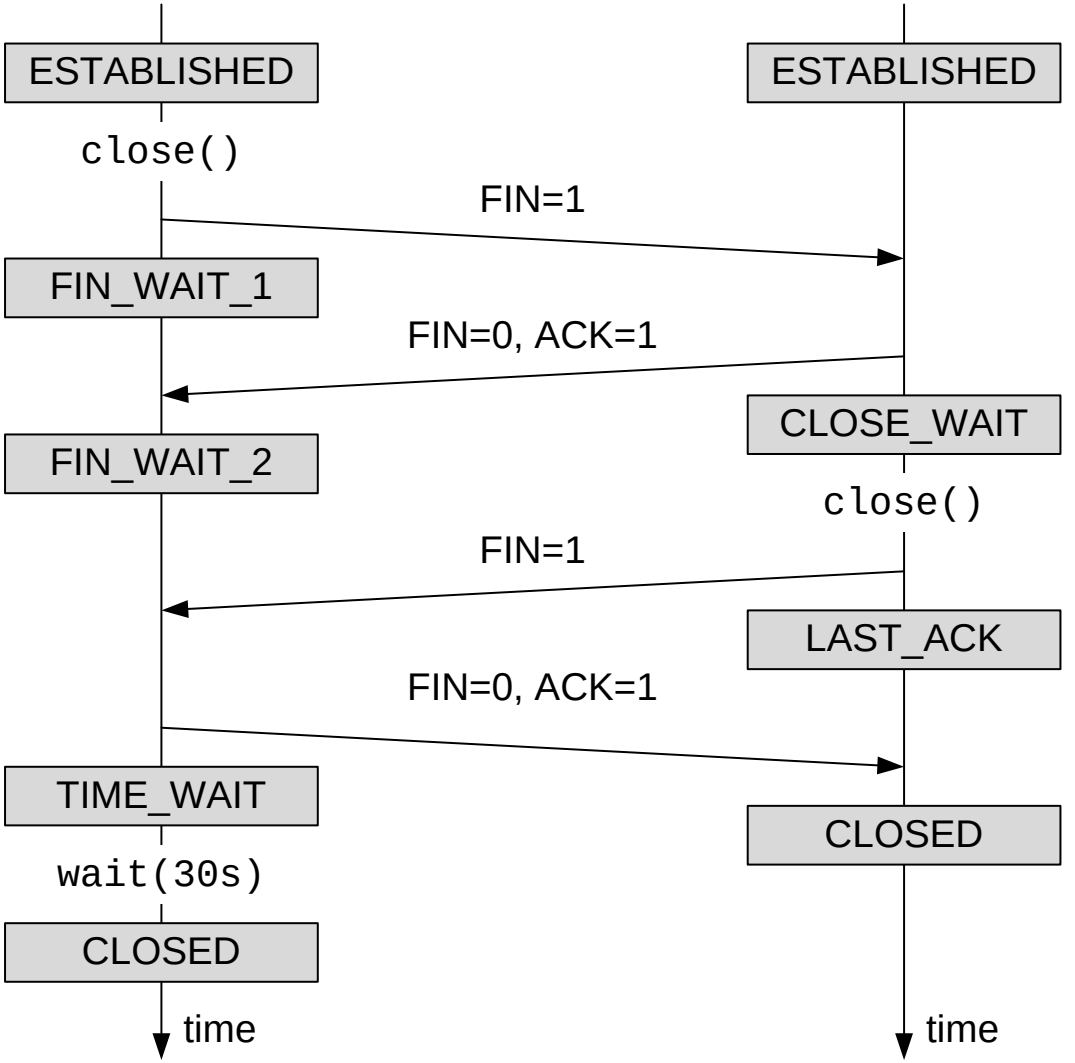
Connection release

Four way close (full duplex channel)

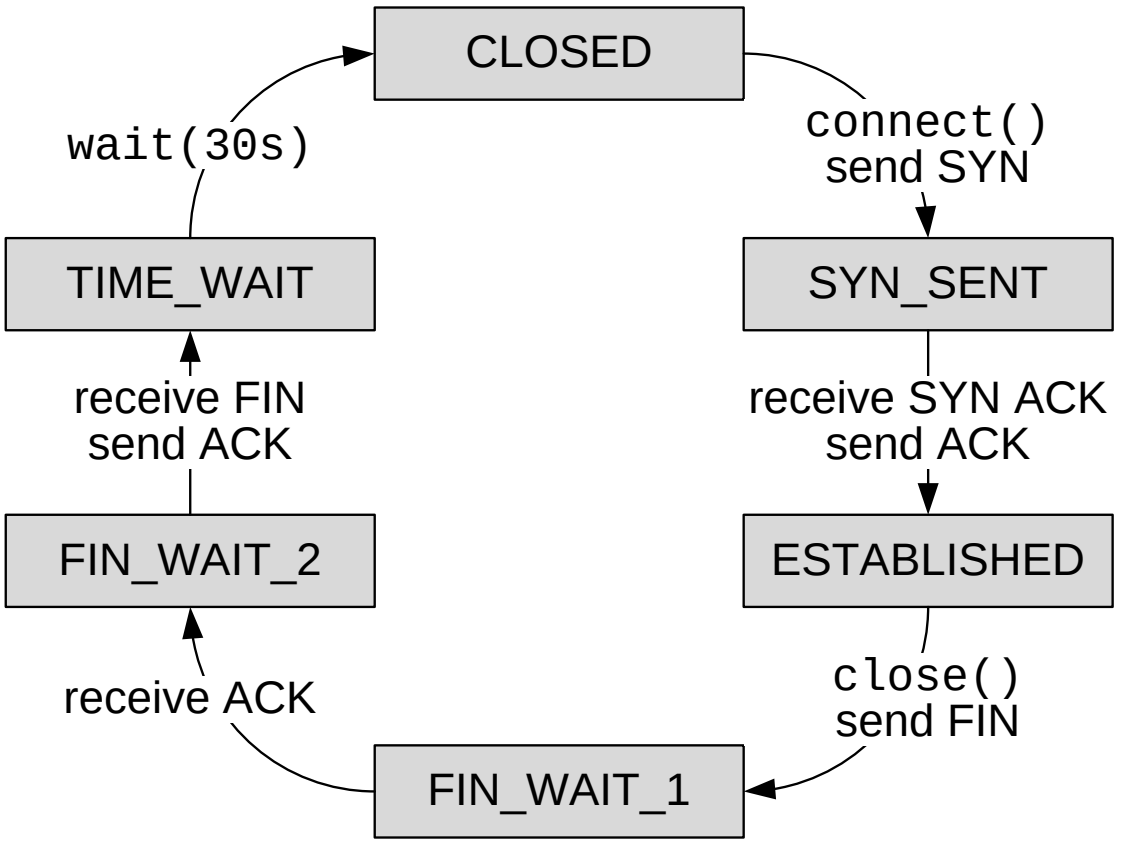
- Release request from the client
  - FIN $= 1$
- Release grant from the server
  - FIN $= 0$, ACK $= 1$
- Release request from the server
  - FIN $= 1$
- Release grant from the client
  - FIN $= 0$, ACK $= 1$

Segments without but also with FIN $= 1$ may contain user data.

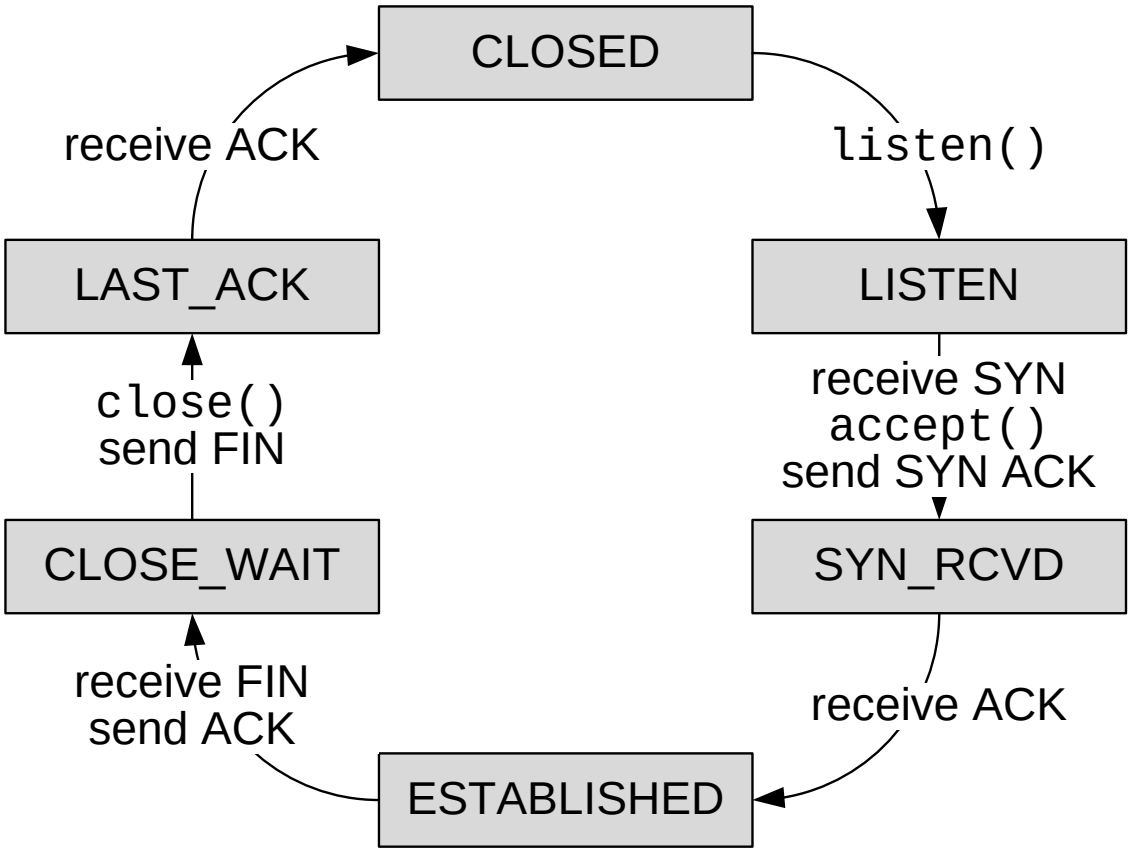
Connection release continued



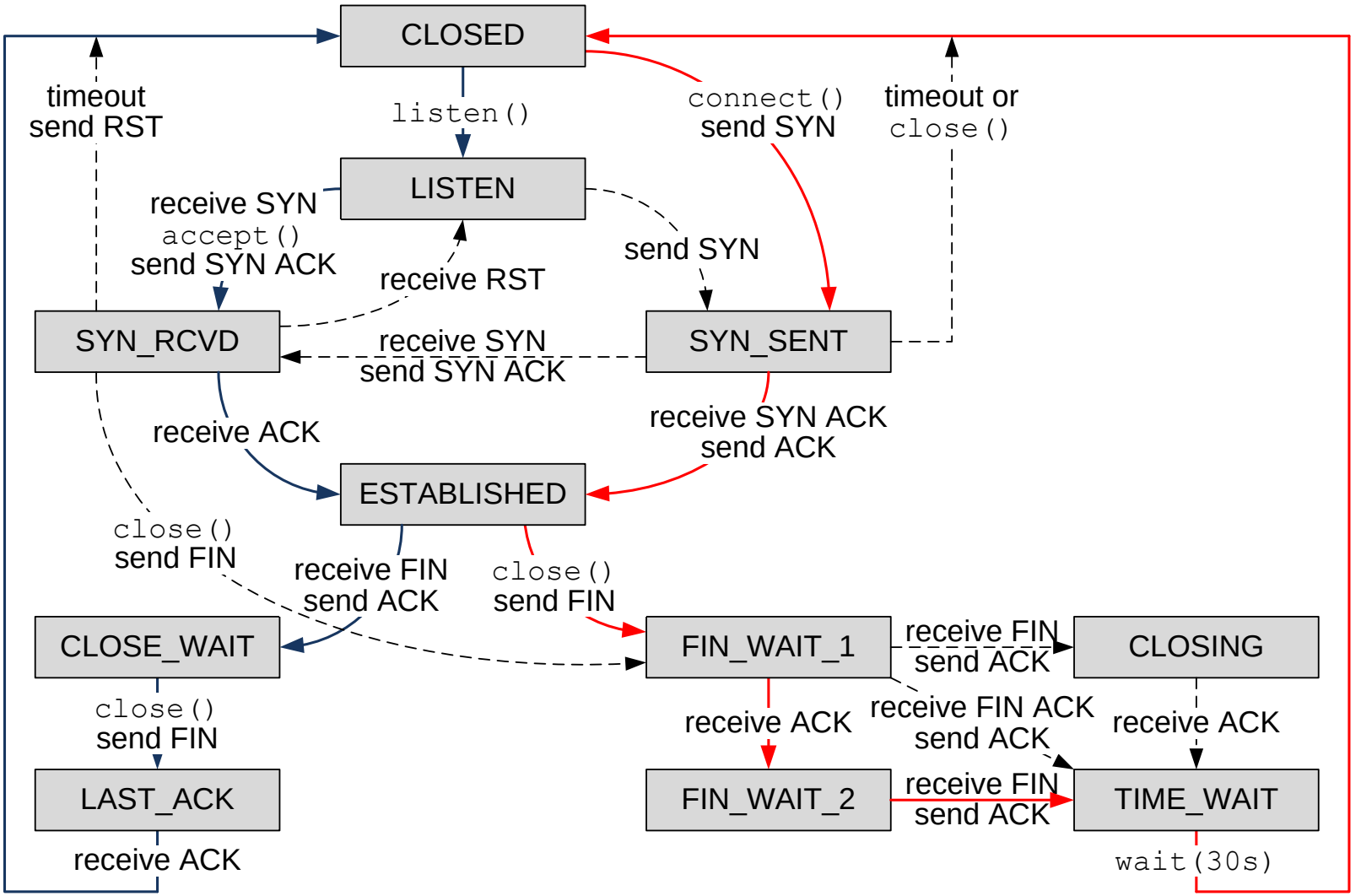
Client-side state-transition-diagram



Server-side state-transition-diagram



Full state-transition-diagram



Non straight-forward and exceptional cases

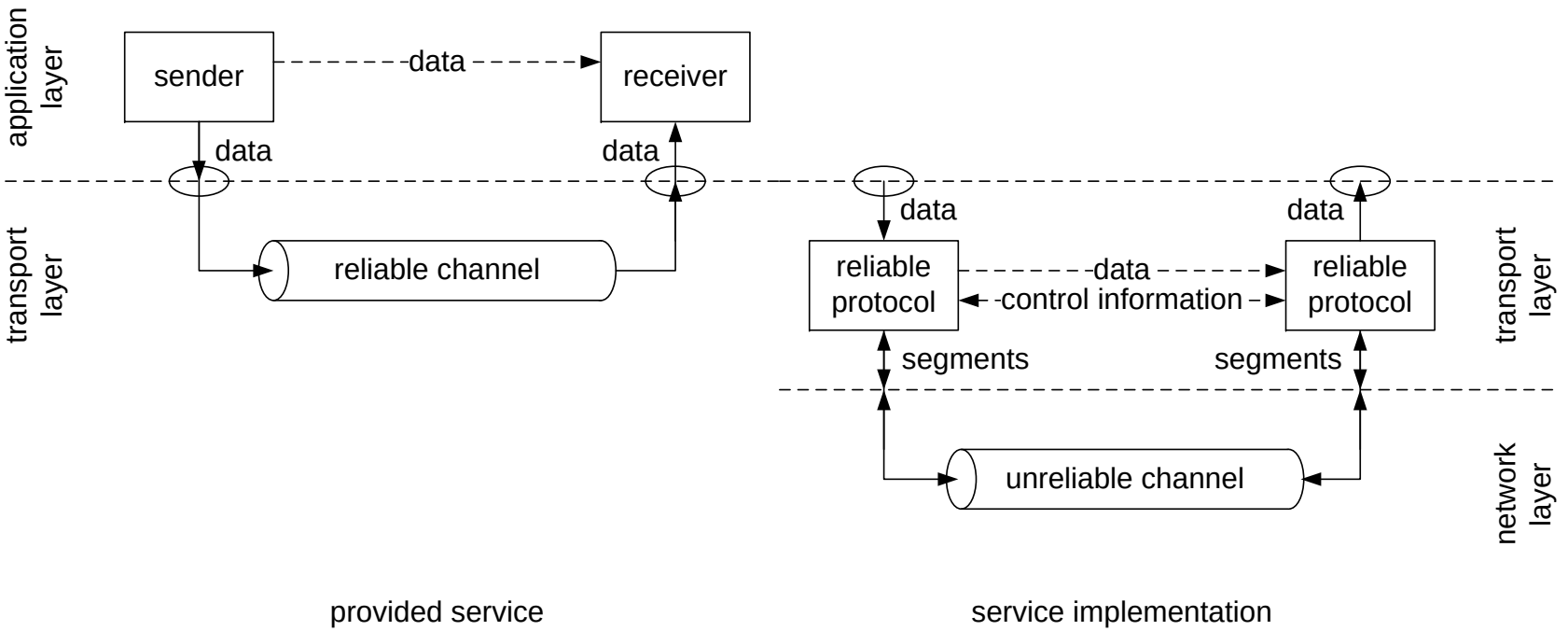
Taking care of all possible cases is usually much more involved than just implementing normal operation.

- Simultaneous close
- Simultaneous open
- Connection refused by server
- Application close during connection establishment
- Various abnormal cases

Timeouts are used (in all states) to close connections and release resources (allocated buffers etc.) if the peer does not respond.

Outline

Reliable data transfer



TCP implements a reliable channel using

- the unreliable channel that is provided by IP
- an automatic repeat request technique
  - control information, acknowledgements
  - retransmissions

TCP's ARQ implementation

TCP uses a hybrid solution that implements several ARQ options

- Go-back-N: the standard does not specify whether
  - out-of-sequence segments are discarded
  - out-of-sequence segments are cached
- Selective repeat
  - specified as a negotiable option

TCP's ARQ implementation has been modified and enhanced several times, e.g.

- RFC 1323: TCP extensions for high performance
- RFC 2001: TCP fast retransmit
- RFC 2018: TCP selective acknowledgement options
- RFC 2988: computing TCP's retransmission timer

Sequence numbers

TCP uses byte (not packet) counters as sequence numbers

- data are viewed as a continuous byte stream
- the byte stream is transmitted in segments
- each byte is numbered

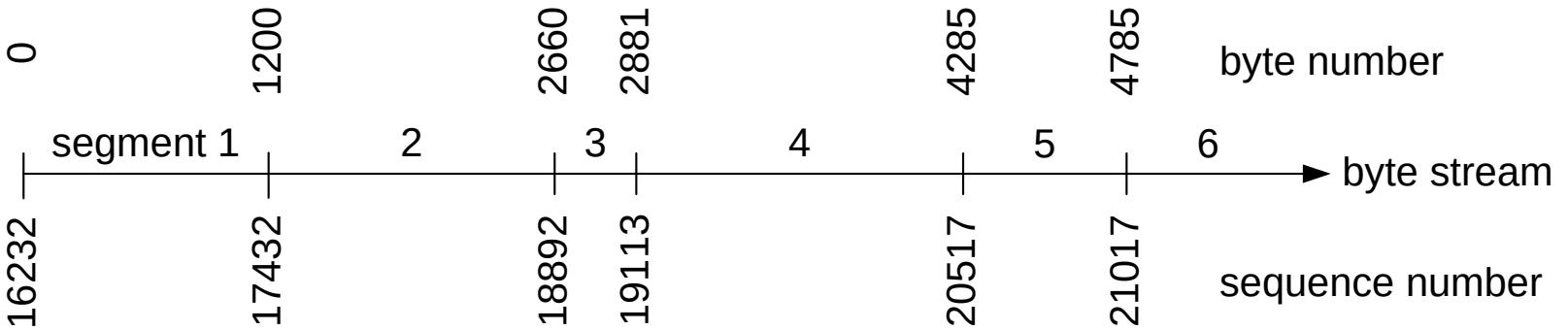
The TCP header specifies

- 32 bit sequence number
  - number of the first data byte of the segment
    subsequent bytes have subsequent numbers
- 32 bit acknowledgement number
  - number of the next in-sequence data byte that the host expects to receive
  - cumulatively acknowledges all lower sequence numbers

TCP is full-duplex. Data for one direction are sent with acknowledgements for the other direction, so-called piggyback ack.

Sequence numbers continued

- sequence numbers are 32 bit with wrap around, i.e. modulo 2^{32} arithmetic
- initial sequence numbers are chosen randomly and synchronized during connection establishment
  - to avoid that old segments that may still be in the network fit into the new sequence
  - for reasons of security: third party attackers cannot simply make a TCP receiver accept their segments



Sequence numbers continued

Protocol implementations at lower layers usually support only a certain maximum transmission unit (MTU).

Segmentation

- at the sender the byte stream that is received from the application layer is broken down into TCP segments
- the sequence numbers indicate where the bytes fit into the stream
- at the receiver all segments that are received in-sequence are passed to the application layer

Segmentation is a transport layer functionality

- only end-systems, i.e., hosts perform segmentation
- it does not apply at intermediate systems, i.e., routers

Segmentation vs. fragmentation

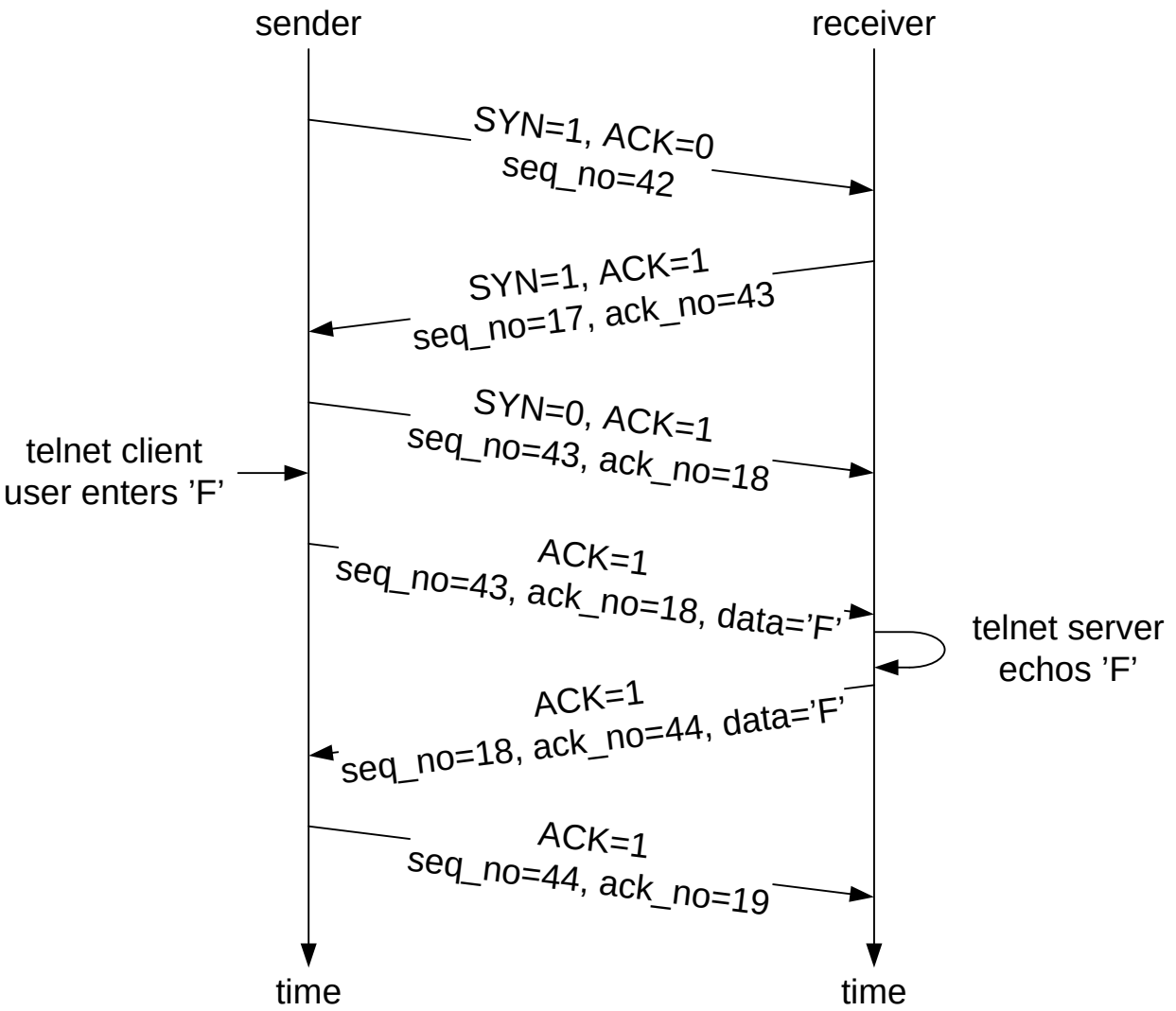
Recall: IP performs fragmentation if IP datagrams are too large.
Why is there a need for TCP segmentation?

The reason becomes obvious in case of packet loss:

- consider a large TCP segment that is fragmented by IP and transmitted as several fragments
  - reassembly fails if one ore more fragments are lost, since IP has no means to retransmit any missing fragments
  - hence, TCP has to retransmit the entire (large) segment
- now assume that TCP instead of one large segment uses several smaller segments that do not need to be fragmented
  - if a segment is lost, TCP can retransmit the lost segment

- TCP seeks to avoid that it's segments exceed the maximum transmission unit (MTU) of the path
- TCP can use the IP don't fragment bit to explore the MTU: segments with don't fragment that exceed the MTU cannot be delivered and cause an error message that contains the MTU

Example: use of sequence numbers



Retransmission timeout

How to configure the retransmission timeout

- should be small to avoid unnecessary waiting for lost segments
- must be larger than the RTT
  - RTT is not static; it can be highly variable
  - too short timeouts cause unnecessary retransmissions

Sender needs to estimate the RTT from measurements

- sample RTT: time between sending a segment and receiving the corresponding acknowledgement (not considering retransmissions)
- estimated RTT: exponentially weighted moving average of sample RTT (typically $\alpha = 1/8$)

$$\text{estimated RTT}_n = (1-\alpha) \cdot \text{estimated RTT}_{n-1} + \alpha \cdot \text{sample RTT}_n$$

Retransmission timeout continued

Need to add a safety margin to estimated RTT depending on the variability of sample RTT

- deviation RTT: exponentially weighted moving average of the deviation of sample RTT and estimated RTT (typically $\beta = 1/4$)

$$\text{deviation RTT}_n = (1-\beta)\cdot \text{deviation RTT}_{n-1} + \beta\cdot \mid \text{sample RTT}_n - \text{estimated RTT}_n \mid$$

- finally, the timeout is selected as

$$\text{timeout interval} = \text{estimated RTT} + 4\cdot\text{deviation RTT}$$

Events during TCP data transfer

During TCP data transfer any of three events may occur

- application data request
  - generate segment with seq_no indicating the next byte
  - if the timer is not already running, start timer
- cumulative acknowledgement:
  - if ack_no is in the current send window move the send window base to ack_no
  - restart timer for remaining unacknowledged segments
- timeout:
  - retransmit segment that caused the timeout
  - double timeout interval (doubled for each retransmission, i.e. exponential backoff: 2,4,8 etc.)
  - restart timer

Delayed acknowledgements

In order to reduce protocol overhead acknowledgements are delayed

- to utilize cumulative acknowledgements
- to utilize piggyback acknowledgements

The receiver delays acknowledgements for $\leq$ 500 ms unless

- a segment with push bit set to one is received
- an in-sequence segment is received and another in-sequence segment has already been received but not yet acknowledged
- an in-sequence segment is received and subsequent segments have already been received previously (out-of-sequence)
- an out-of-sequence segment is received

in which case a (cumulative) acknowledgement is sent immediately.

Fast retransmit

Under certain conditions TCP generates a retransmission without waiting for a timeout

- a lost segment in a row of transmitted segments causes a gap at the receiver
- the receiver sends an acknowledgement for each segment received out-of-sequence
  - the ack_no is stalled at the beginning of the gap
  - acknowledgements with identical ack_no are called duplicate acknowledgements
- the sender uses duplicate acks as an indication of a gap at the receiver at ack_no
- after three duplicate acks the sender retransmits the segment starting at ack_no without waiting for the timeout, so-called fast retransmit

Outline

Flow control

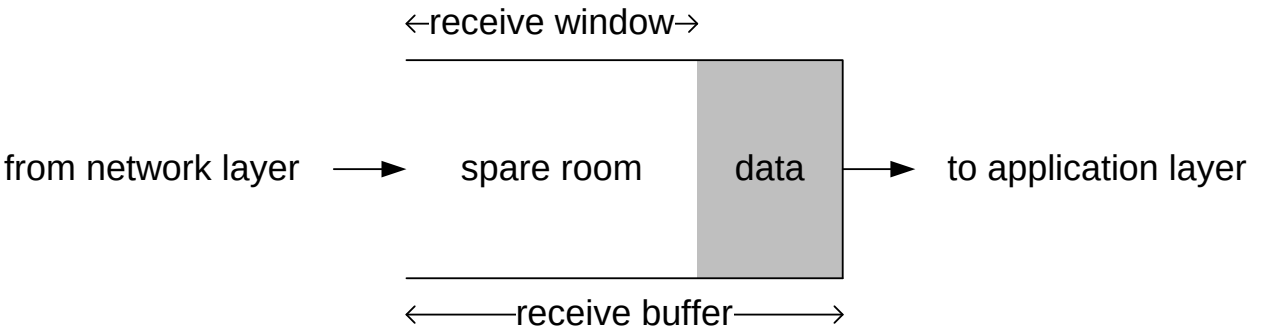
A fast sender may overwhelm a slow receiver with data

- using UDP the receiver discards data it cannot handle
- instead TCP uses flow control to throttle the sender

TCP's flow control implementation

- the receiver maintains a buffer (in the socket) for received but not yet processed data, the buffer size is denoted receive buffer
- the unused part of the receive buffer is called the receive window
- the receive window is advertised in the 16-bit receive window field in the TCP header
- small receive windows ($<$ MSS) are not advertised (Nagle's algorithm to avoid repeated transmissions of small segments, called "silly window syndrome")

Receive window



- the receiver computes the receive window as

$$\text{receive window} = \text{receive buffer} - (\text{last byte received} - \text{last byte read})$$

- the sender may send data only as long as

$$\text{receive window} \geq \text{last byte send} - \text{last byte acked}$$

Deadlock avoidance

The basic flow control mechanism can cause deadlocks, e.g. if

- receiver indicates receive window $= 0$
- sender is not allowed to send new data
- all segments are already acknowledged by the receiver
- since the sender cannot send new segments the receiver will not be triggered to send acknowledgements
- the receiver will never notify the sender if receive window $\neq 0$
- sender and receiver are blocked forever $\Rightarrow$ deadlock!

Simple solution

- if receive window $= 0$ the sender sends (nevertheless) segments with 1 byte of data (window probes)
- these segments are generated based on the expiry of the so-called persist timer