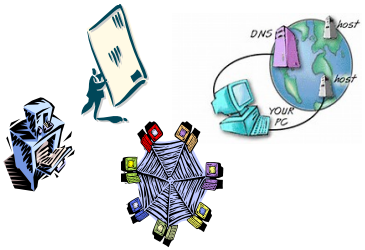
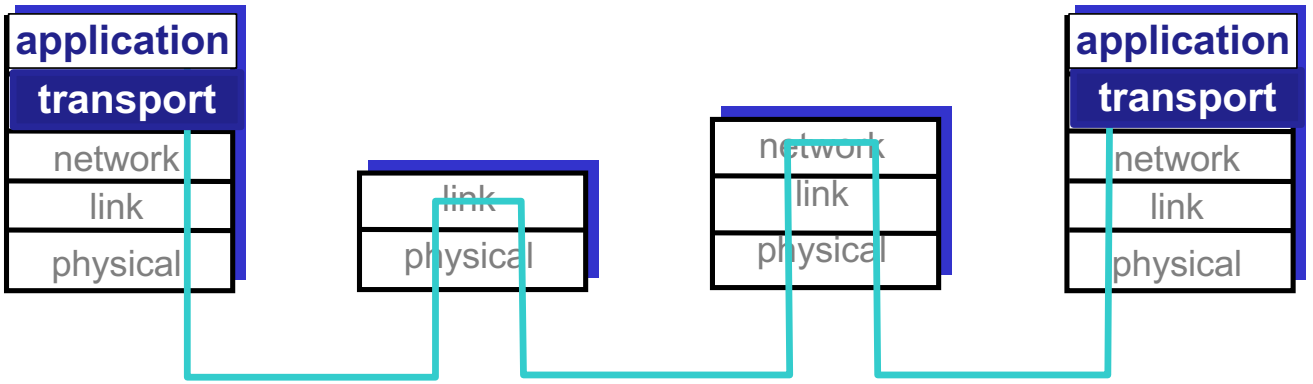
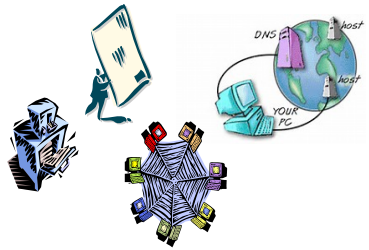
Transport layer
Chapter 3

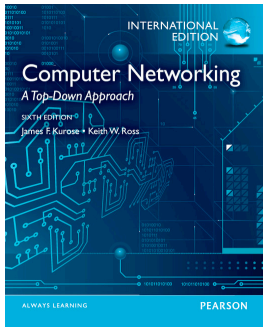
Kjersti Moldeklev, Prof II
Department of
Information Security and
Communication Technology

kjersti.moldeklev@ntnu.no



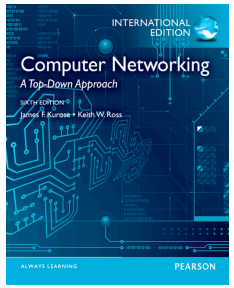






The *content* of some of these slides are based on slides available from the web-site of the book by J.F Kurose and K.W. Ross.

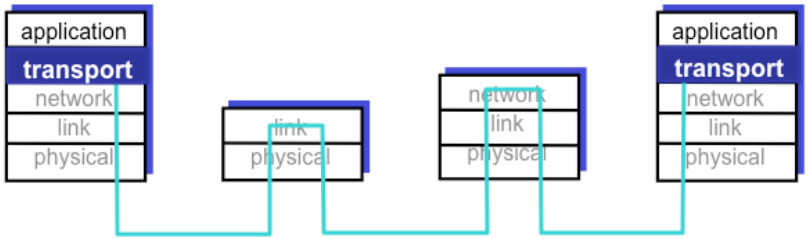
This week, January 26-27



3	Friday 09:15 – 11:00	Application Layer (cont)	R1	Kjersti	Chapter 2 Chapter 8.2-3 and 8.5.1
4	Thursday 12:15 – 14:00 Thursday 14:15 – 15:00	Transport Layer Theory Assignment 2: *Application Layer*	R1 R1	Kjersti Assistants/ Ida/Norvald	Chapter 3 One must deliver and pass at least 5 of the 8 theory assignments.
4	Friday 09:15 – 11:00	Transport Layer (cont)	R1	Kjersti	Chapter 3
5	Thursday 12:15 – 14:00	Transport Layer (cont)	R1	Kjersti	Chapter 3 Chapter 8.6
5	Friday 09:15 – 11:00	Network Layer	R1	Kjersti	Chapter 4

With socket programming from ch 2

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP (Transmission Control Protocol)

- segment structure
- reliable data transfer
- flow control
- connection management

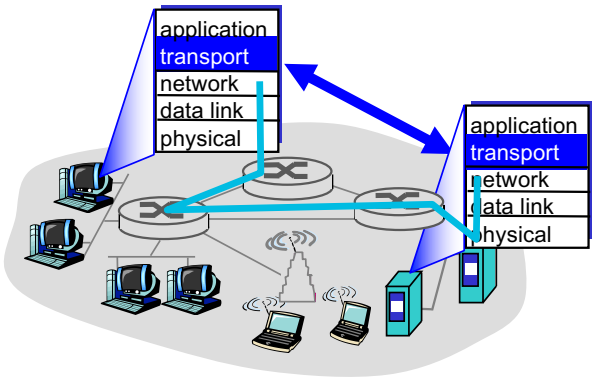
3.6 Principles of congestion control

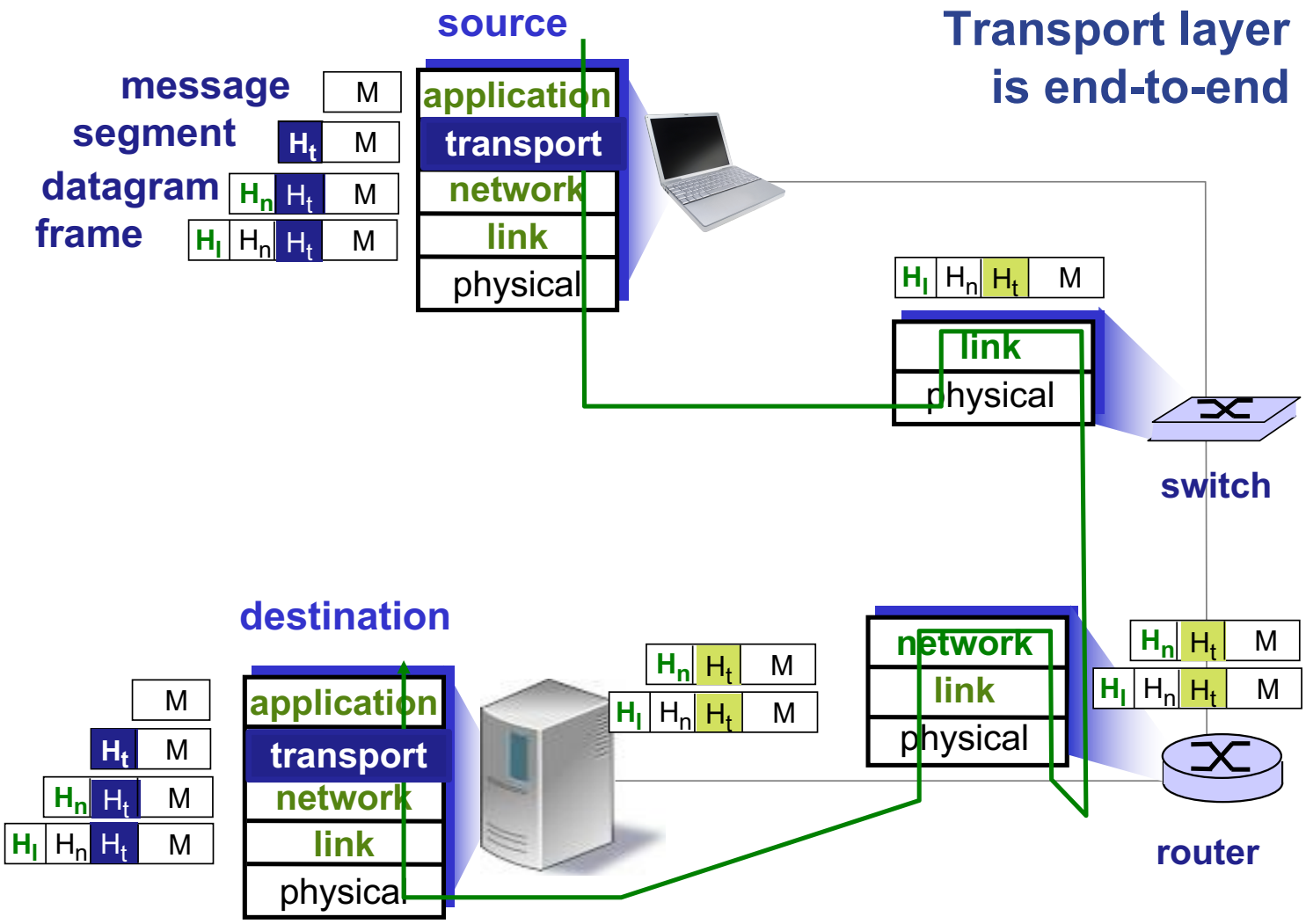
3.7 TCP congestion control

Transport services are provided by end-to-end protocols

- Transport layer **provides** communication **between end-system processes**
  - transfers application data/protocol
  - source and destination port number
- Sender: breaks application **messages into segments**, passes to network layer
- Receiver: reassembles **segments into messages**, passes to application layer

- Transport layer **relies on and enhances the network layer service**
  - **IP** datagrams including
    - transport-layer segment as service data
    - source and destination IP address




source
Transport layer
is end-to-end
message
segment
datagram
frame
M
Ht M
Hn Ht M
Hl Hn Ht M
application
transport
network
link
physical
Hl Hn Ht M
link
physical
switch
destination
application
transport
network
link
physical
Hn Ht M
Hl Hn Ht M
network
link
physical
router

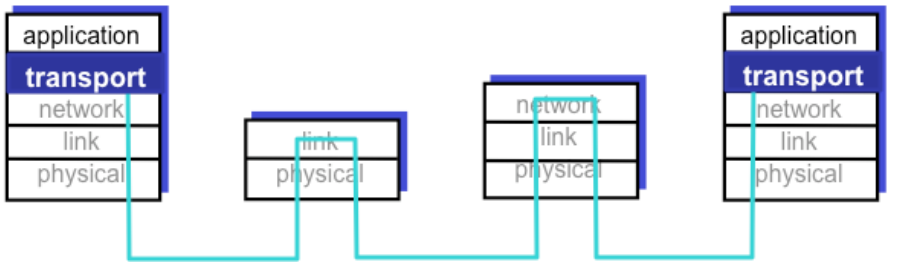
Transport layer is accessed from the applications through a socket API

Goals

- understand **principles** behind **transport layer services**
  - multiplexing/demultiplexing
  - reliable data transfer
  - flow control
  - congestion control

- **UDP – User Datagram Protocol**
  - unreliable, unordered delivery
  - no-frills extension of “best-effort” IP
- **TCP – Transmission Control Protocol**
  - connection setup
  - error control
  - flow control
  - congestion control
- Services not available
  - delay guarantees
  - bandwidth guarantees

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Port numbers in transport protocol header

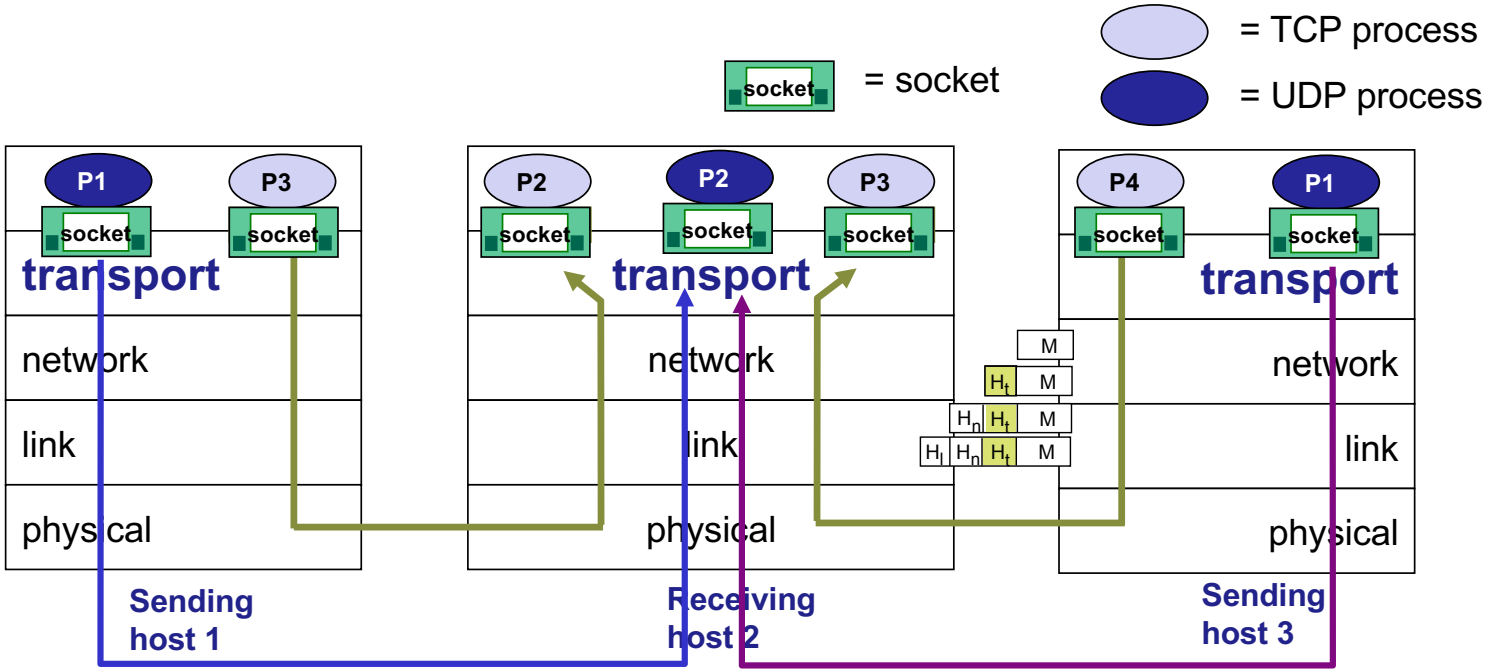
Multiplexing/demultiplexing in transport layer

Multiplexing at sender:

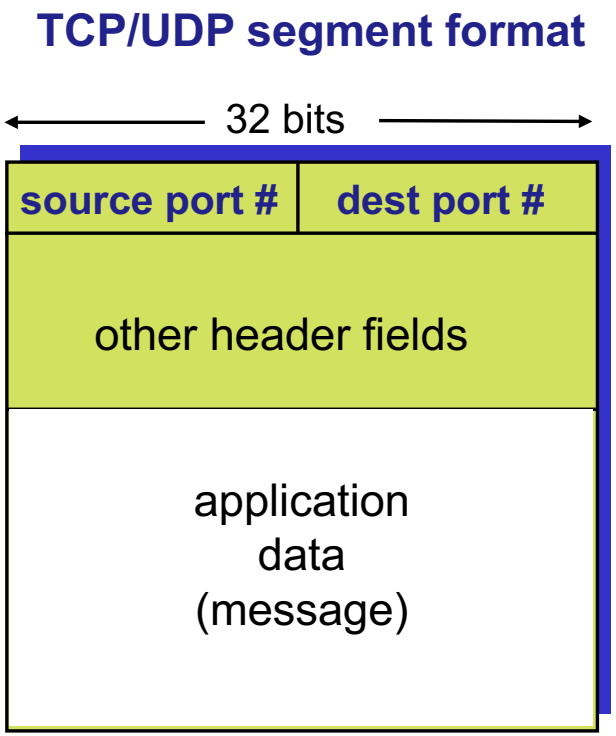
Gathering data from sockets, adding protocol header

Demultiplexing at receiver:

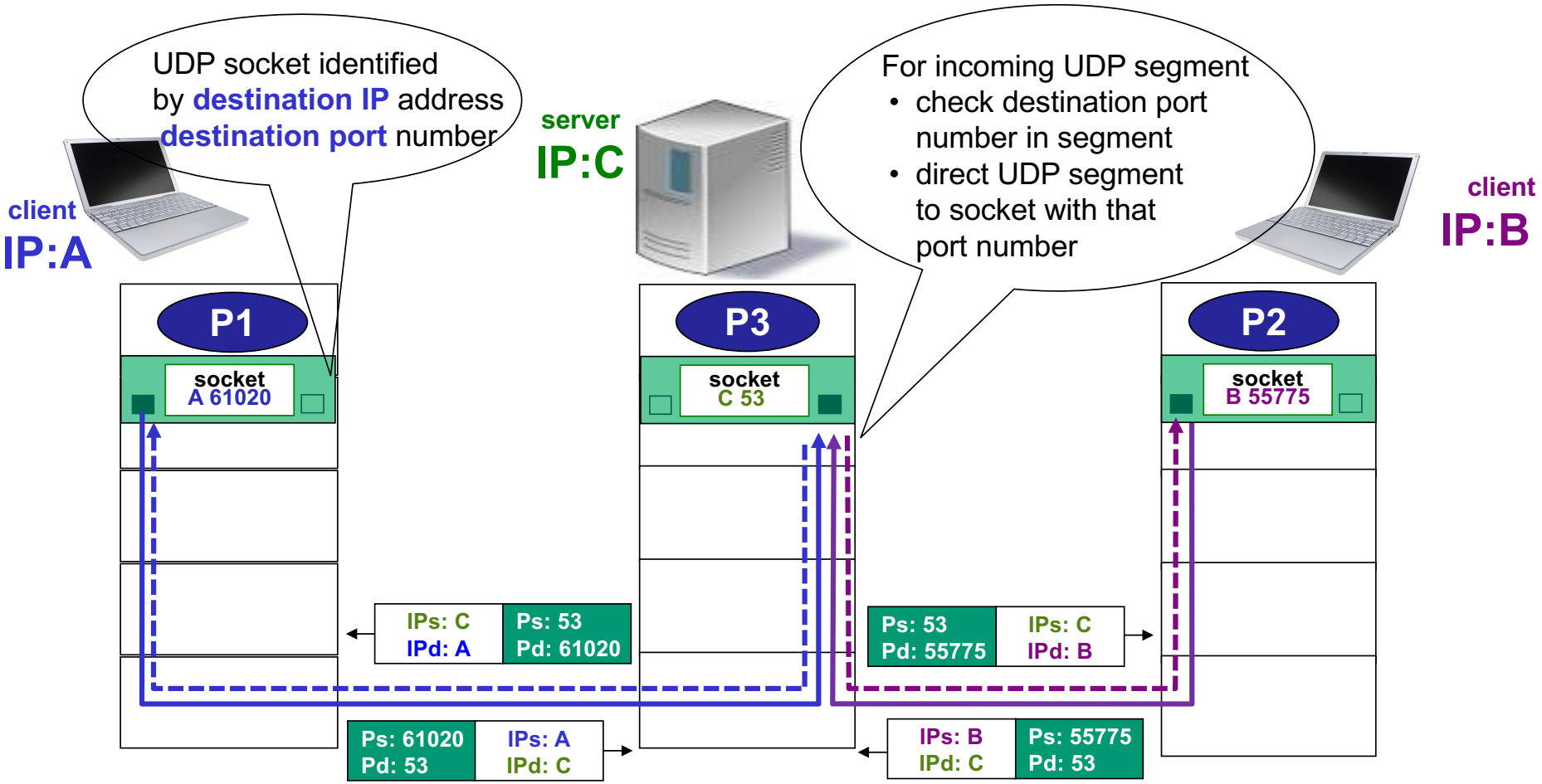
Delivering received segments to correct socket



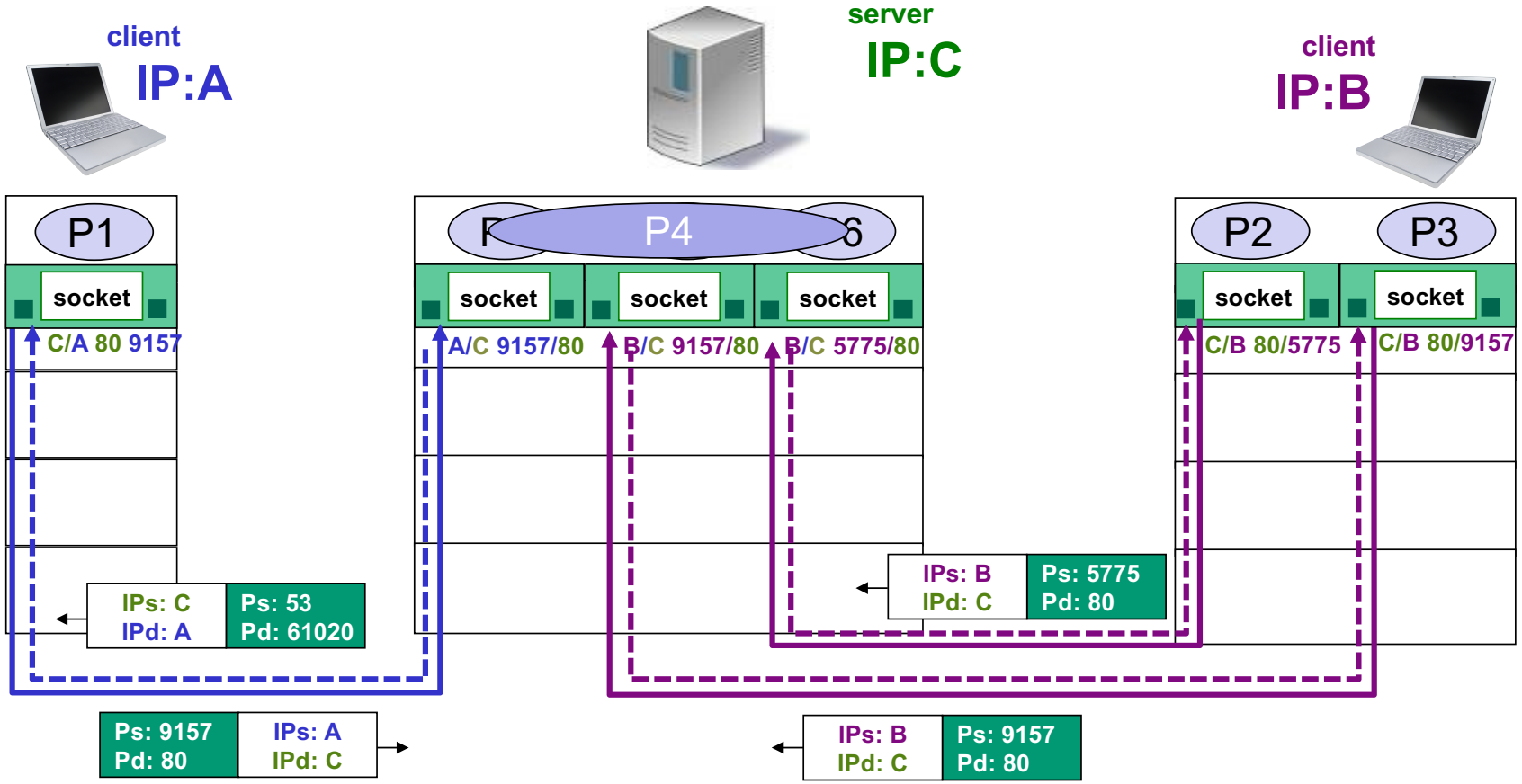
Port numbers of the transport protocols are used for demultiplexing



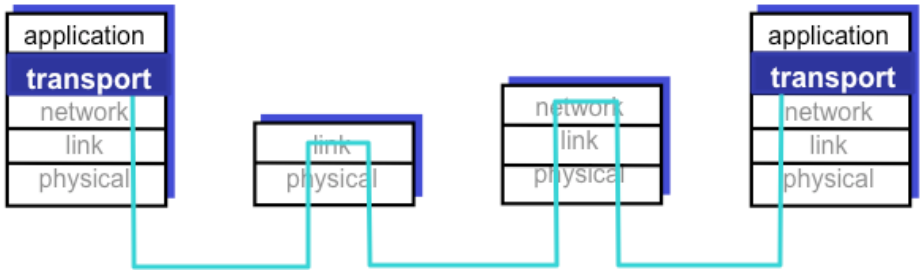
UDP socket identified by [IP destination address, UDP destination port]



TCP socket identified by connection 4-tuple
[IP source & destination address, TCP source & destination port]



Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Connectionless and unreliable

NTNU

User Datagram Protocol – UDP

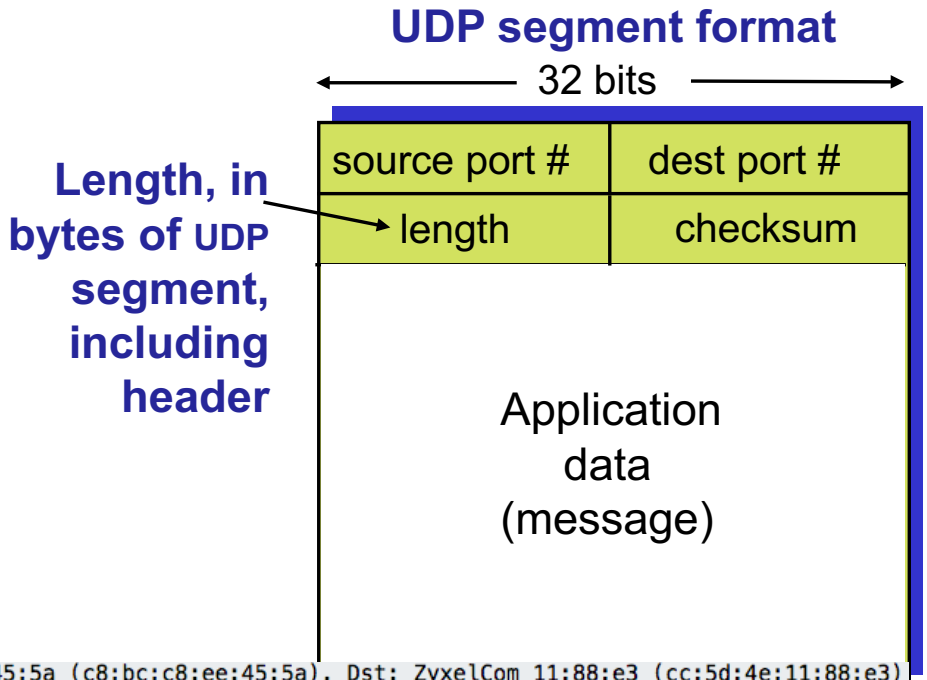
- "No frills," "**bare bones**" Internet transport protocol
- "**Best effort**" service, UDP segments may be
  - lost
  - delivered out of order
- **Connectionless**
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

- No connection establishment (which add delay)
- Simple: no connection state at sender, receiver
- Small segment header
- No congestion control: UDP can blast away as fast as desired

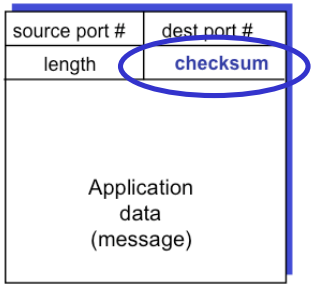
UDP is used for streaming and low delay applications

- Streaming multimedia applications
  - loss tolerant
  - rate sensitive
- Other UDP users
  - DNS
  - SNMP (Simple Network Management Protocol)
- Reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!




```
▸ Ethernet II, Src: Apple_ee:45:5a (c8:bc:c8:ee:45:5a), Dst: ZyxelCom_11:88:e3 (cc:5d:4e:11:88:e3)
▸ Internet Protocol Version 4, Src: 10.0.0.37, Dst: 130.67.15.198
▾ User Datagram Protocol, Src Port: 61037 (61037), Dst Port: domain (53)
    Source Port: 61037 (61037)
    Destination Port: domain (53)
    Length: 40
  ▸ Checksum: 0xa3c7 [validation disabled]
    [Stream index: 18]
▸ Domain Name System (query)
```

UDP Internet checksum detects “errors” (e.g. flipped bits) in transmitted segment



Senders

- Treat segment content as sequence of 16-bit integers
- Checksum: addition (1’s complement sum)
- Sender puts checksum value into UDP checksum field

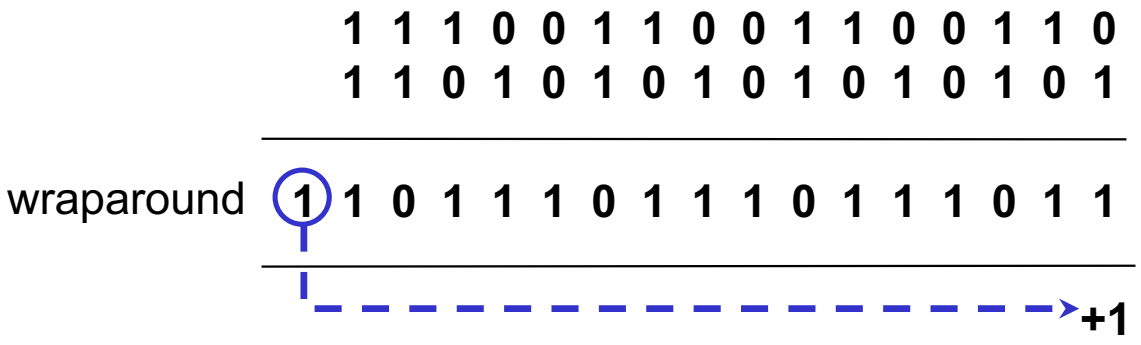
Receivers

- Compute checksum of received segment
- Check if computed checksum equals checksum field value
  - NO - error detected
  - YES - no error detected. But maybe errors nonetheless?

UDP checksum is used end-to-end

Internet checksum: 1s complement of the sum of all 16-bit words

- When adding numbers, a carryout from the most significant bit needs to be added to the result
  - Example: add two 16-bit integers

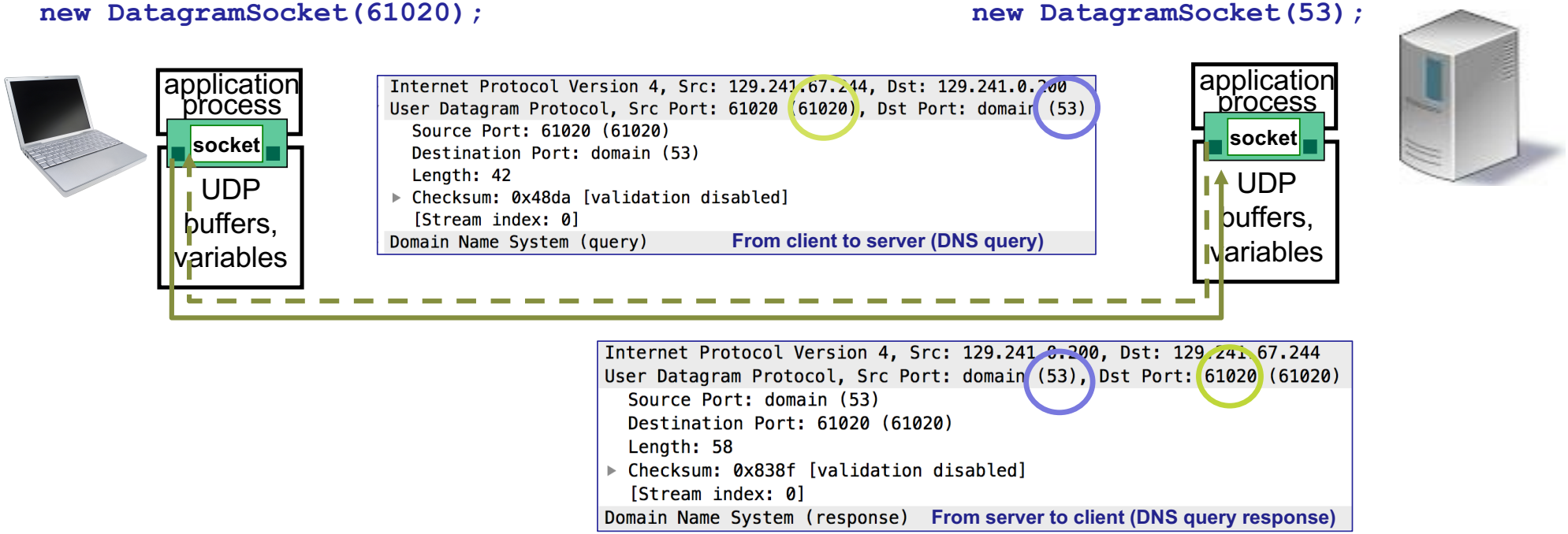


Checksum= 1s complement of sum

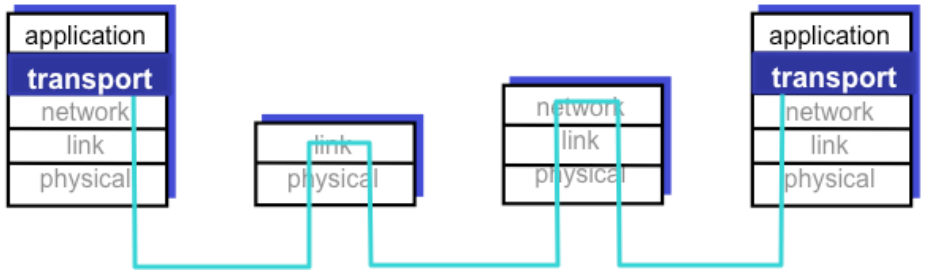
Source IP address and source UDP port provide return IP address and UDP port

```
DatagramSocket mySocket =
  new DatagramSocket(61020);
```

```
DatagramSocket serverSocket =
  new DatagramSocket(53);
```



Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP (Transmission Control Protocol)

- segment structure
- reliable data transfer
- flow control
- connection management

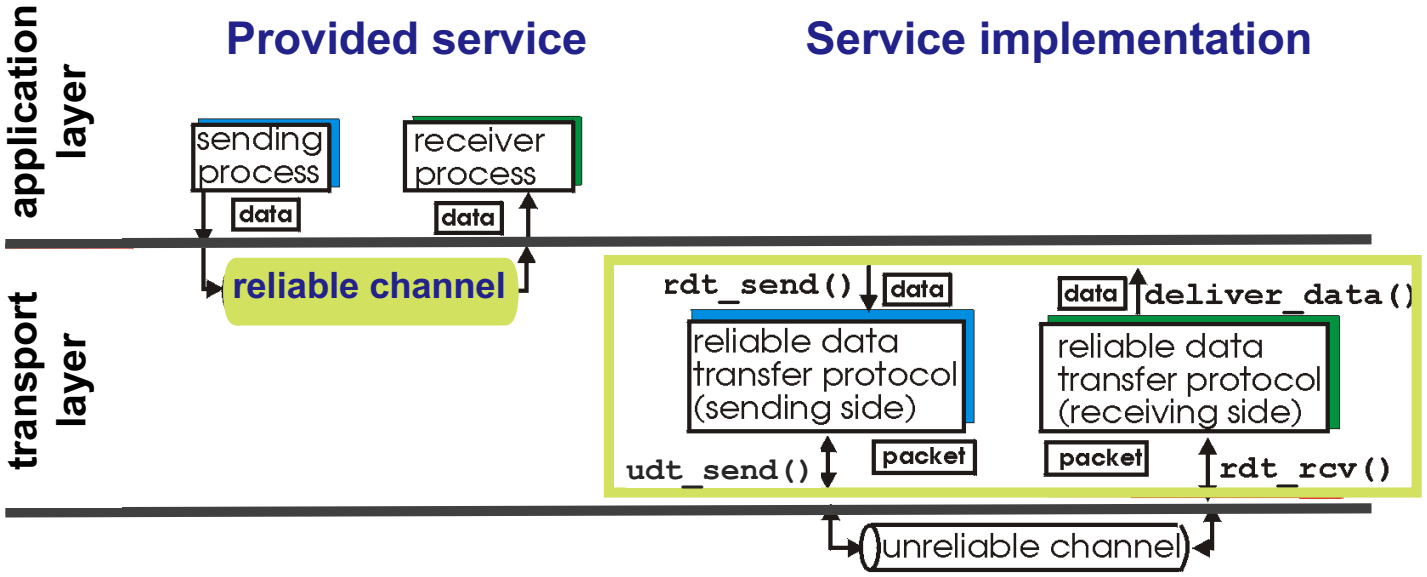
3.6 Principles of congestion control

3.7 TCP congestion control

checksum, ACKs, timeout, sequence numbers, window, retransmission: go-back-N, selective

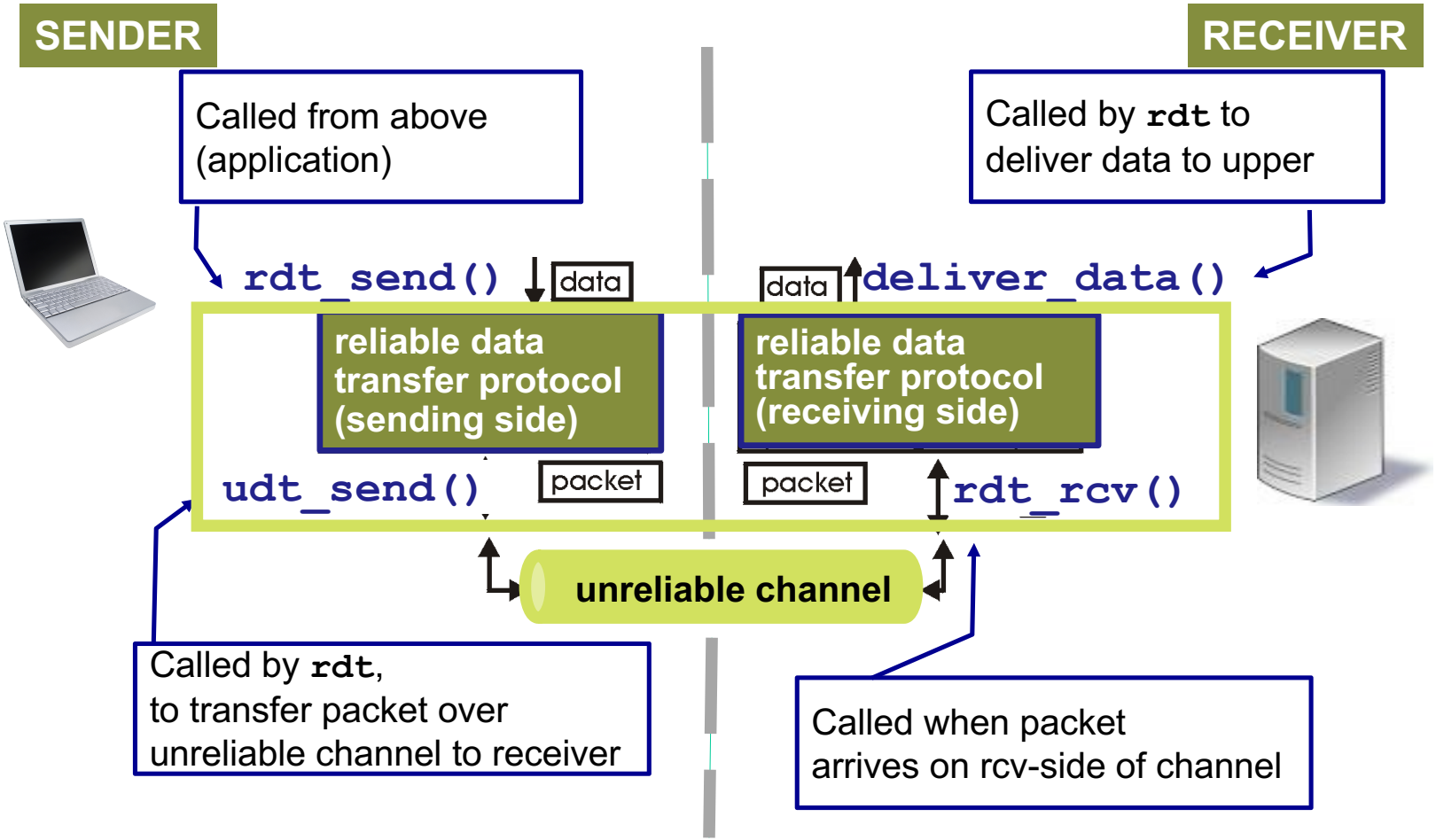
Reliable data transfer may be done in different layers

- Where depends on which network and protocols

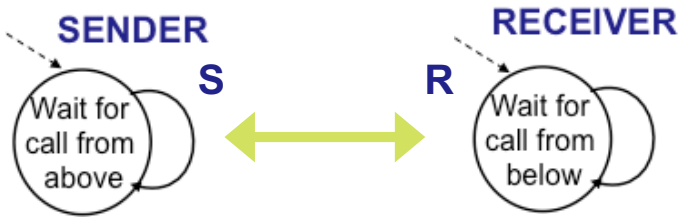


Characteristics of the unreliable channel will determine the complexity of the reliable data transfer protocol

Reliable data transfer (rdt) over an unreliable channel



Reliable data transfer (rdt) over various channel characteristics

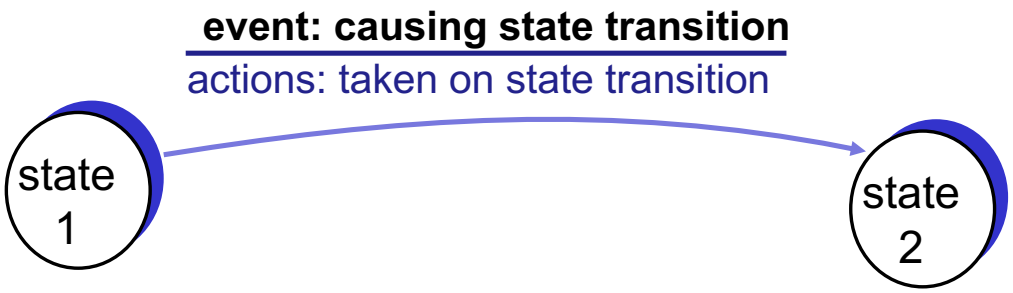


Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK

NTNU

Reliable data transfer is stateful – next state determined by event

- Use finite state machines (FSM) to specify sender and receiver
  - consider only unidirectional data transfer
    - but control info will flow in both directions!



Next state uniquely determined by next event

NTNU

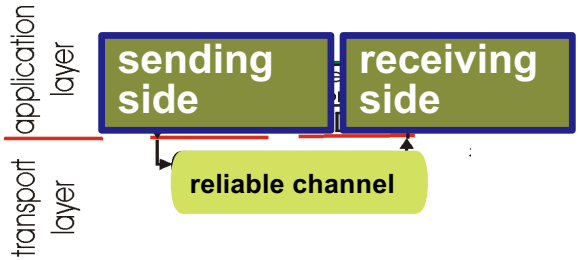
Reliable data transfer 1.0

Reliable transfer over a reliable channel

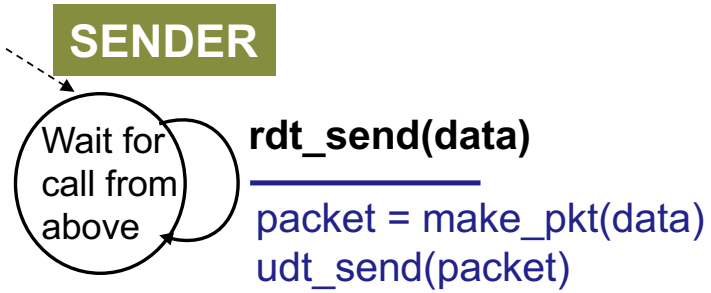
Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	

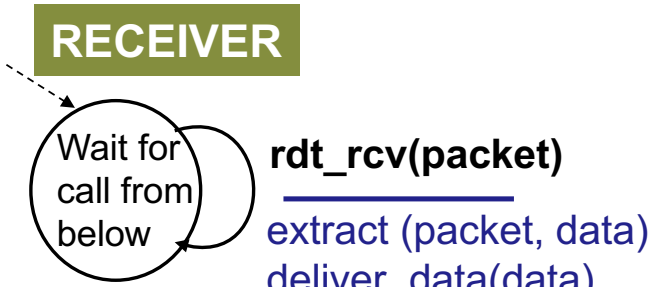
event

actions



- Underlying channel perfectly reliable
  - no bit errors
  - no loss of packets
- Sender sends data into underlying channel
- Receiver reads data from underlying channel





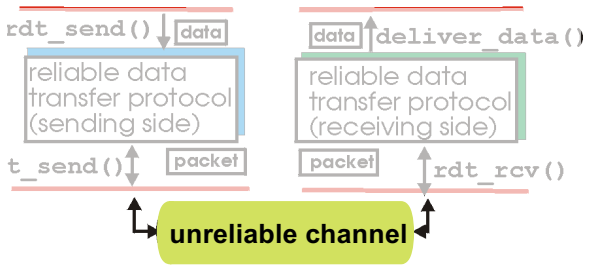
Reliable data transfer 2.0

Channel with bit errors needs more functionality in the transport layer

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK

- Unreliable channel: underlying channel may flip data bits in packet

- New mechanisms in Rdt2.0
  - **error detection:** checksum to detect bit errors
  - **receiver feedback**: control messages (ACK, NAK) from receiver -> sender



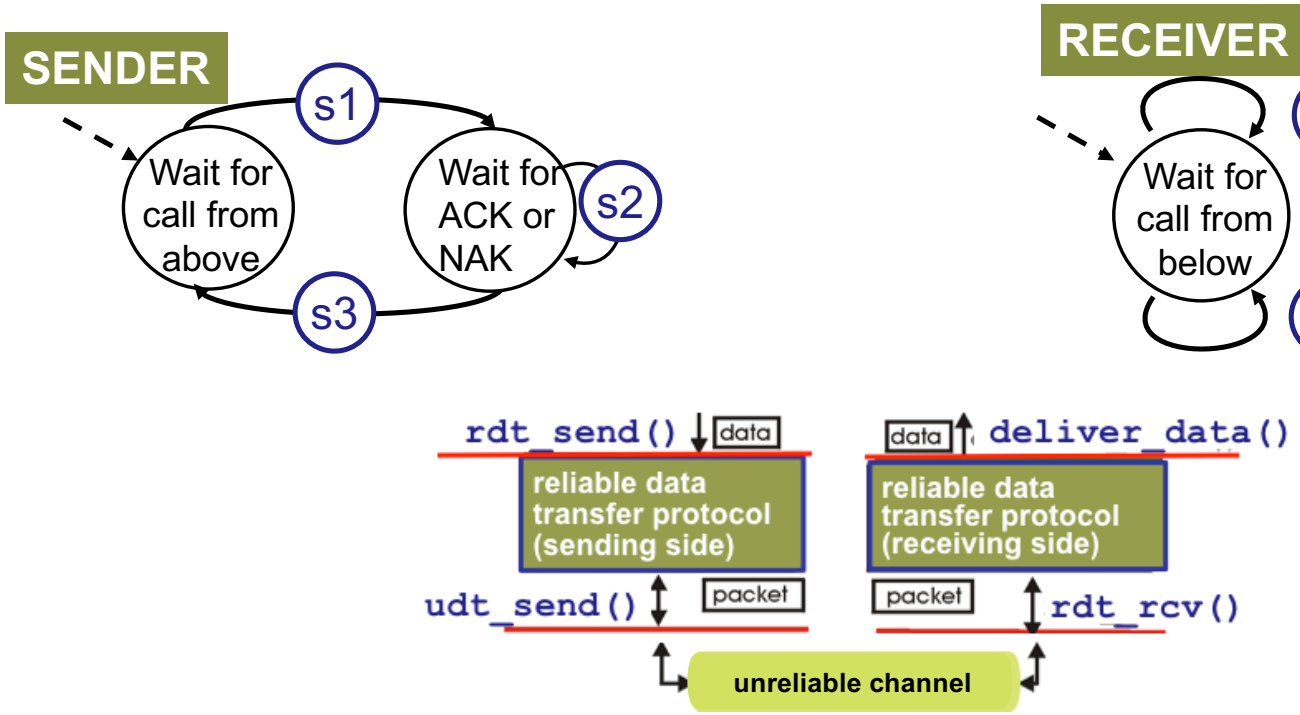
- **Acknowledgements (ACKs)**: receiver explicitly tells sender that packet received OK

- **Negative acknowledgements (NAKs)**: receiver explicitly tells sender that packet had errors
  - sender **retransmits** packet on receipt of NAK

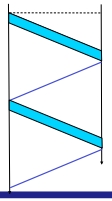
Reliable data transfer 2.0

Finite state machine – stop and wait

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK



stop and wait



Sender sends one packet, then waits for receiver response

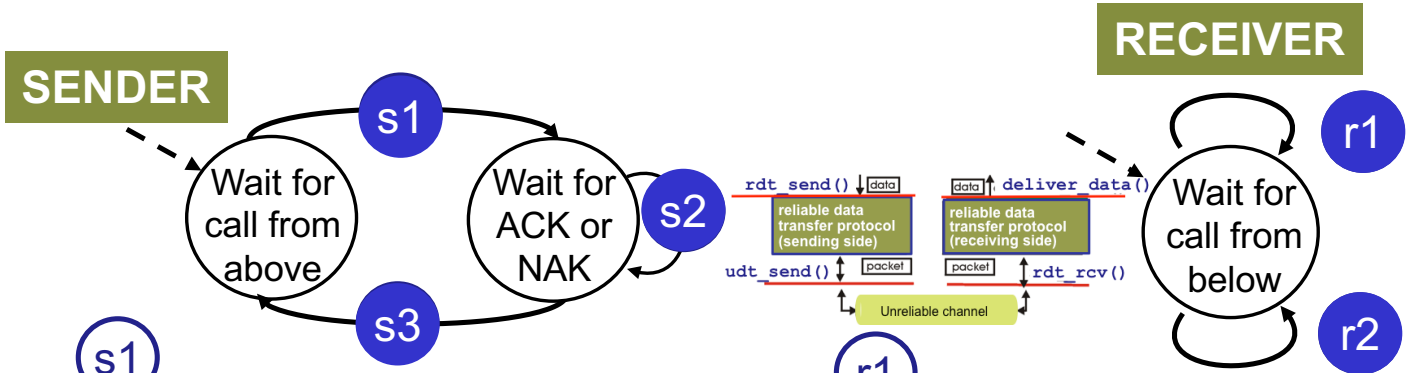
Reliable data transfer 2.0

Finite state machine – stop and wait

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK

event

actions



s1

rdt_send(data)

sndpkt = make_pkt(data, **checksum**)
udt_send(sndpkt)

s2

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

s3

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

r1

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt=make_pkt(**NAK**)
udt_send(sndpkt)

r2

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(**ACK**)
udt_send(sndpkt)

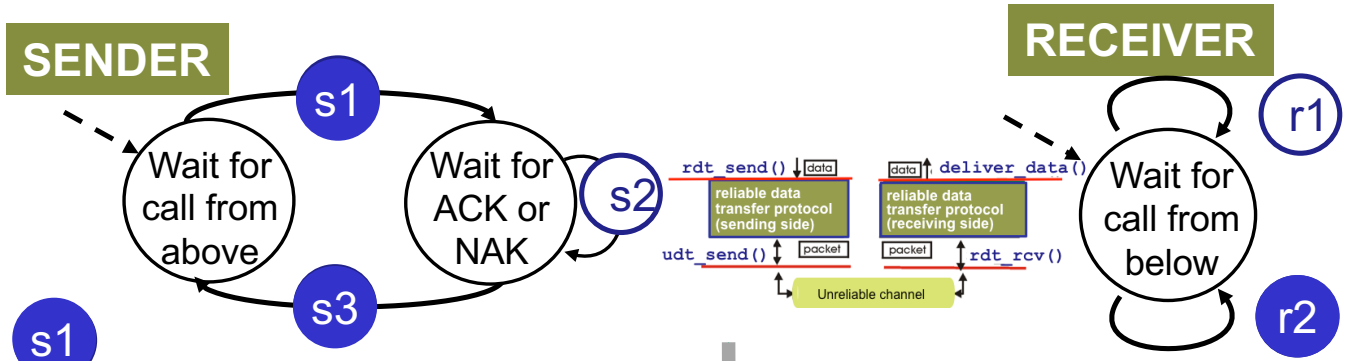
Reliable data transfer 2.0

Communication WITHOUT errors

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK

event

actions



s1

rdt_send(data)

sndpkt = make_pkt(data, **checksum**)
udt_send(sndpkt)

s3

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

r2

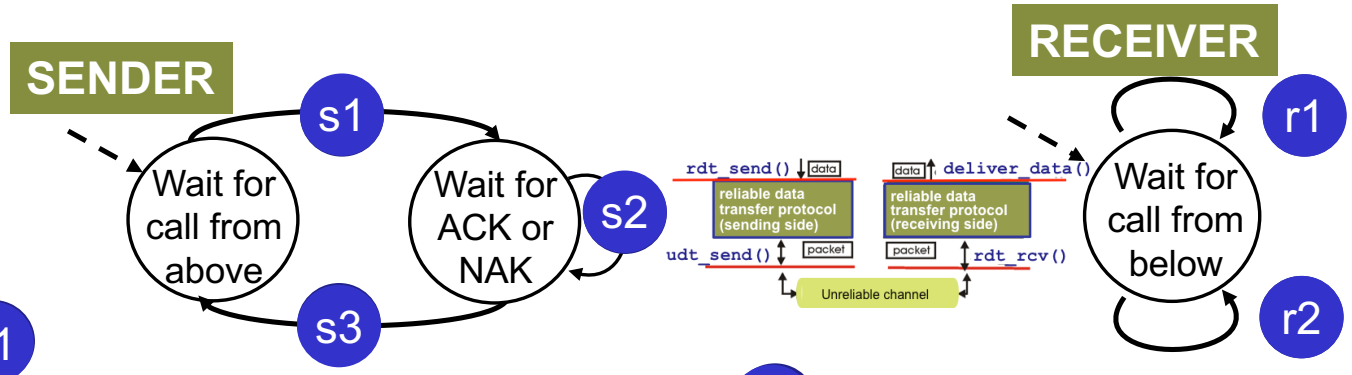
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(**ACK**)
udt_send(sndpkt)

Reliable data transfer 2.0

Communication WITH errors

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK



event

actions

s1

rdt_send(data)

sndpkt = make_pkt(data, **checksum**)
udt_send(sndpkt)

s2

rdt_rcv(rcvpkt) && isNAK(rcvpkt)

udt_send(sndpkt)

s3

rdt_rcv(rcvpkt) && isACK(rcvpkt)

Λ

r1

rdt_rcv(rcvpkt) && corrupt(rcvpkt)

sndpkt=make_pkt(**NAK**)
udt_send(sndpkt)

r2

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt=make_pkt(**ACK**)
udt_send(sndpkt)

NTNU

Rdt2.0 has a fatal flaw: corrupted ACK/NAK is not handled

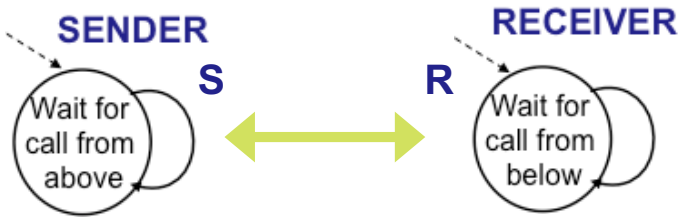
Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK

- What happens if ACK/NAK corrupted?
  - sender doesn't know what happened at receiver!
  - can't just retransmit: **possible duplicate**

Handling duplicates

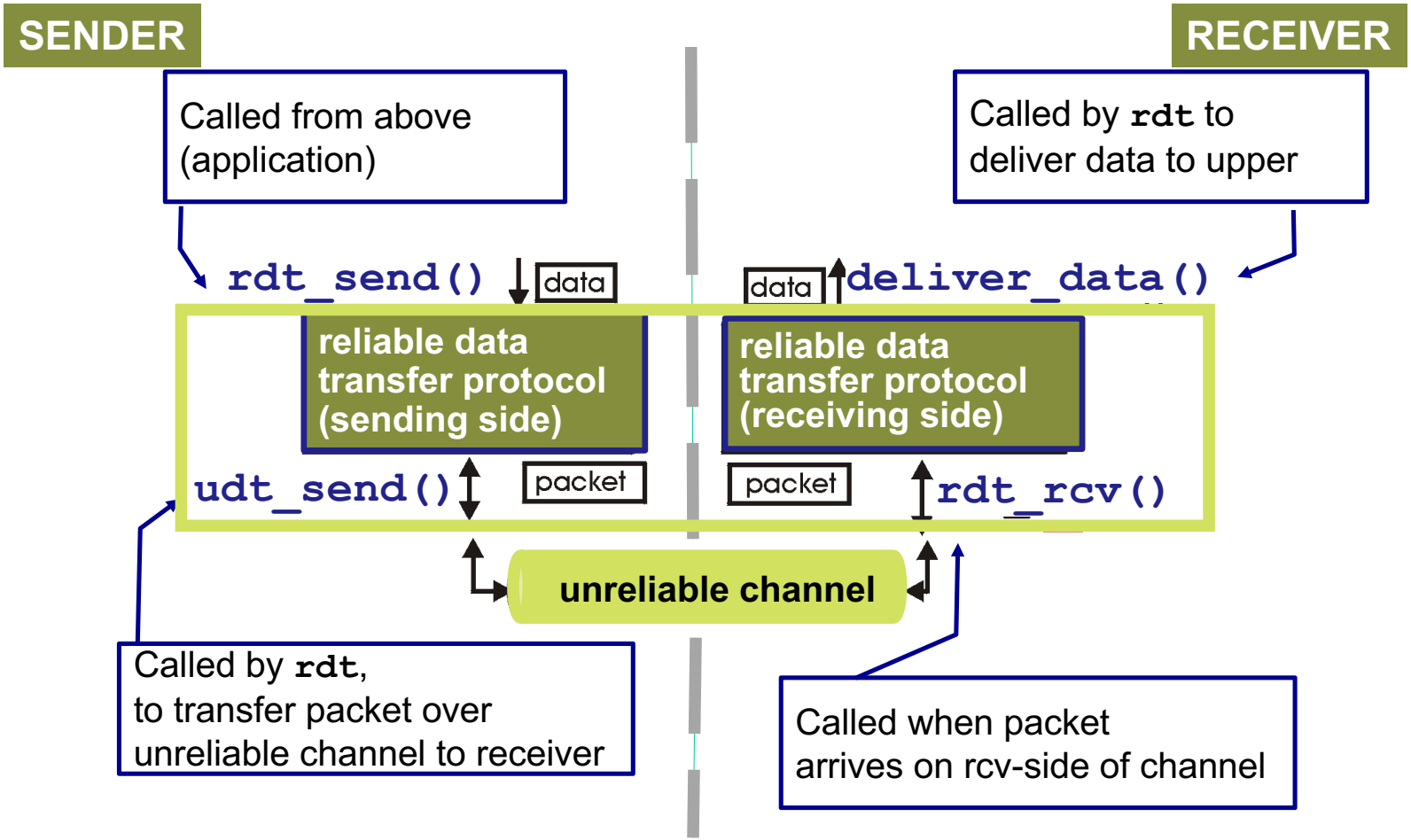
1. sender **retransmits** current packet if ACK/NAK garbled (needs **checksum** on ACK/NAK)
2. sender adds **sequence number** to each packet to detect duplicates
3. receiver discards (doesn't deliver up) duplicate packet

Reliable data transfer (rdt) over various channel characteristics



Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	**checksum**, receiver feedback (**ACK, NAK**), **retransmission** on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK

Reliable data transfer over an unreliable channel



NTNU

Reliable data transfer 2.1: stop-and-wait over unreliable channel

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NACK may also be corrupt	Seq# on data packets, checksum on ACK/NAK

Sender

- **Sequence number** added to packet
- Twice as many states
  - state must “remember” whether “current” pkt sequence number has 0 or 1
- Must **check** if received **ACK/NAK corrupted**
- Two seq. #’s (0,1) will suffice

Receiver

- **Checks** if received packet is **duplicate**
  - state indicates whether 0 or 1 is expected packet sequence number
- Note: receiver cannot know if its last ACK/NAK received OK at sender

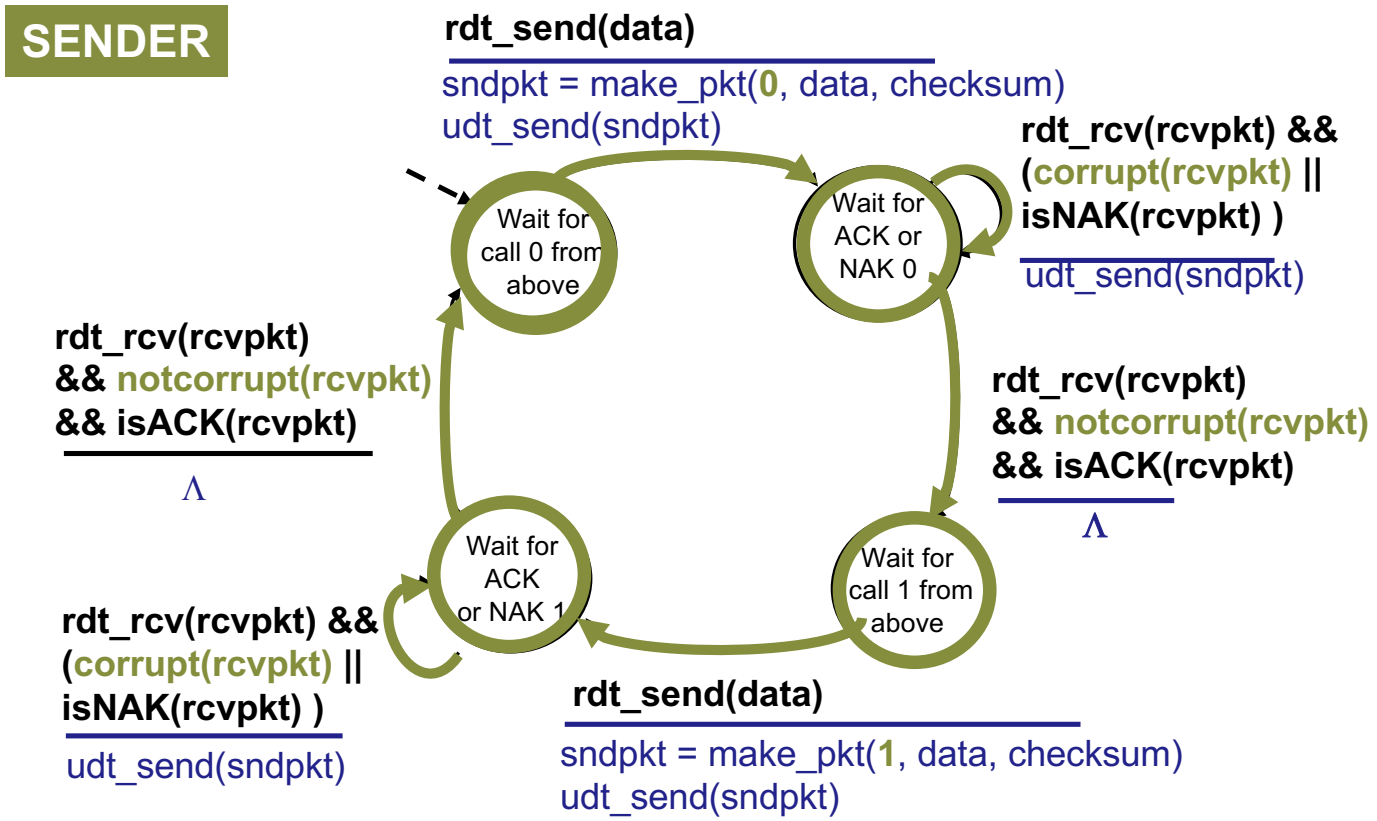
Reliable data transfer 2.1

SENDER includes packet sequence number (0,1) && checks if NAK/ACK corrupt

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	Checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NACK may also be corrupt	Seq# on data packets, checksum on ACK/NAK

event

actions



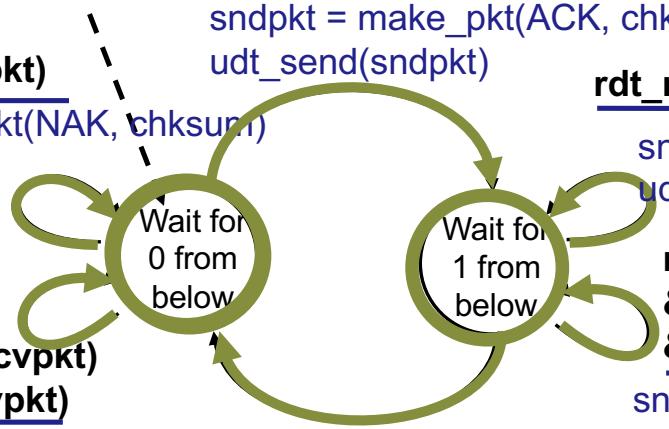
RECEIVER checks on duplicates using the packet sequence number

Rdt 2.1	Bit error S <-> R; ACK NACK may also be corrupt	**Sequence number** on data packets, checksum on ACK/NAK

RECEIVER

event

actions



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && (corrupt(rcvpkt)

sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

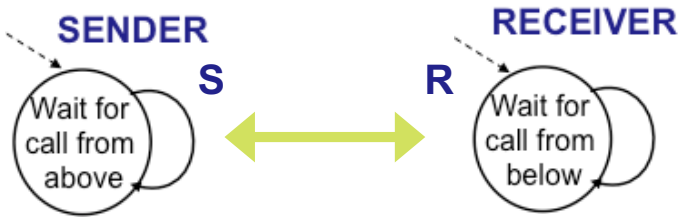
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)

sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)

extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

Reliable data transfer (rdt) over various channel characteristics



Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	**checksum**, receiver feedback (**ACK, NAK**), **retransmission** on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	**seq#** on data packets, **checksum on ACK/NAK**
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK

NTNU

Reliable data transfer 2.2
A NAK-free protocol

- Same functionality as v2.1, but using ACKs only

- Instead of NAK (negative acknowledgement), receiver sends ACK for last packet received OK
  - **receiver must explicitly include sequence number** of packet being ACKed

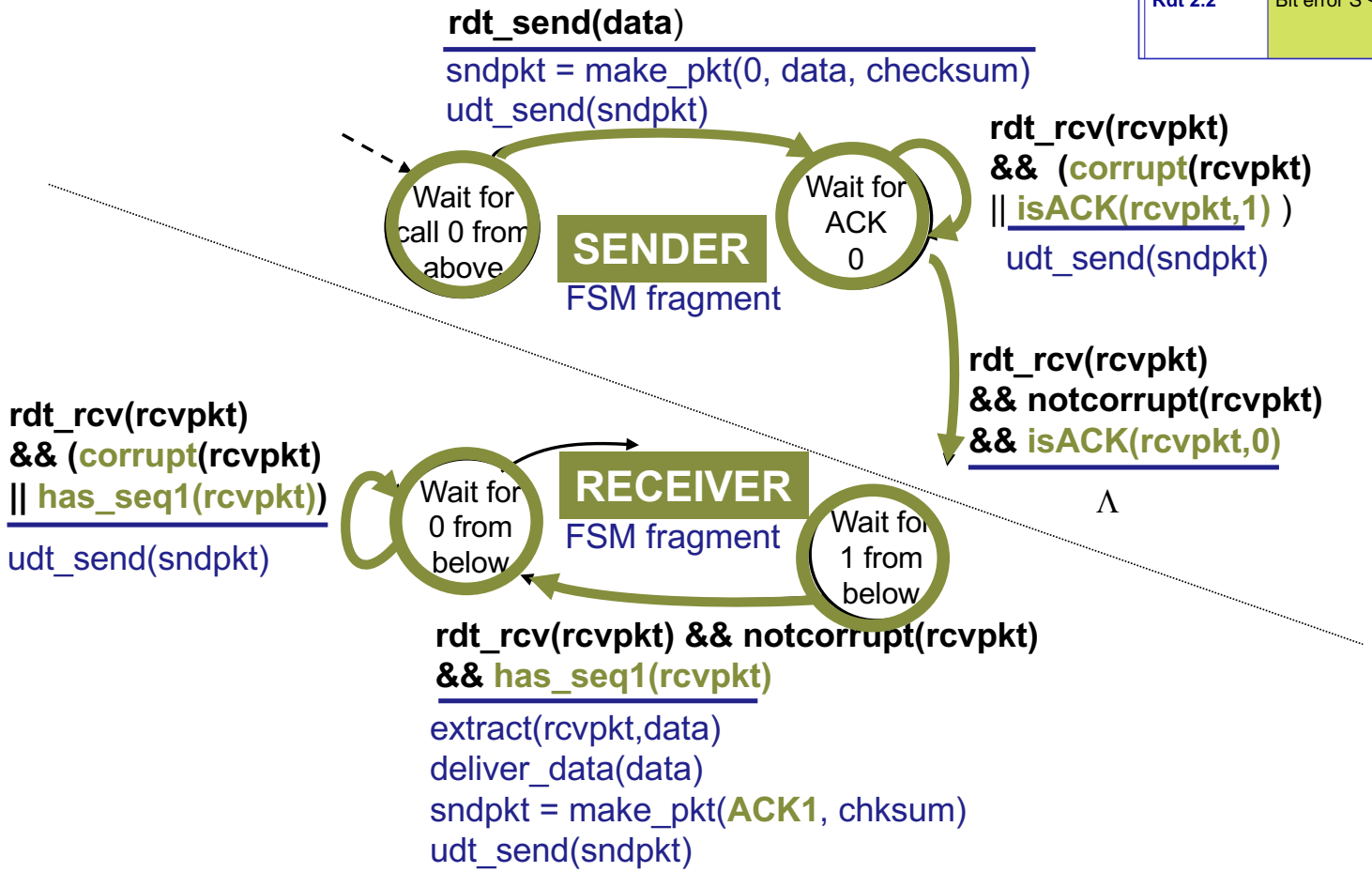
Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK

Receiving duplicate ACK at sender results in same action as receiving NAK: retransmit current packet

Reliable data transfer 2.2 FSM fragments

ACK with sequence number

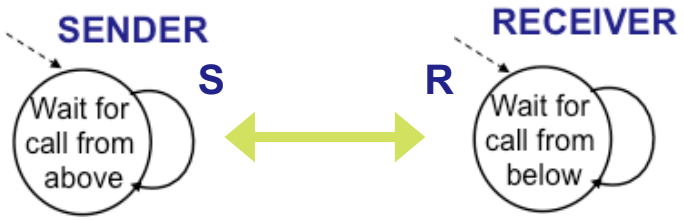
Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK



event

actions

Reliable data transfer (rdt) over various channel characteristics



Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	**checksum**, receiver feedback (**ACK, NAK**), **retransmission** on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	**seq#** on data packets, **checksum on ACK/NAK**
Rdt 2.2	Bit error S <-> R	no NAK, **seq# also in ACKS**, **retransmission on duplicate ACK**
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout

NTNU

Reliable data transfer 3.0

Channels with loss as well as errors

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout

- New assumption: underlying channel can **also loose packets** (data or ACKs)
  - **Checksum**, **sequence numbers**, **ACKs**, **retransmissions** will be of help, but not enough

Approach

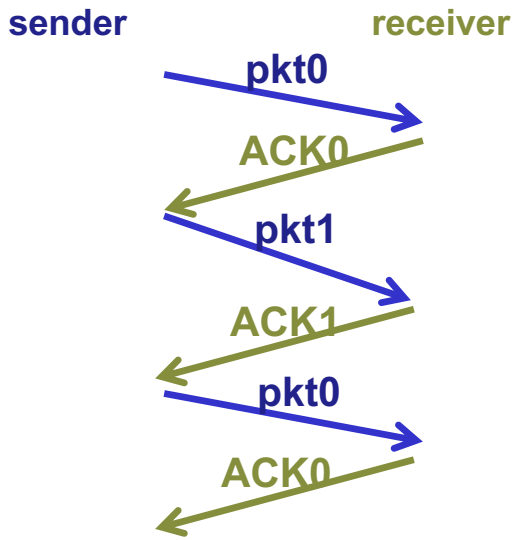
- Sender waits “reasonable” amount of time for ACK
- Retransmits if no ACK received in this time

- If packet (or ACK) just delayed (not lost):
  - retransmission will be duplicate; seq #’s handles this
  - receiver must specify seq # of packet being ACKed
- Requires **countdown timer**

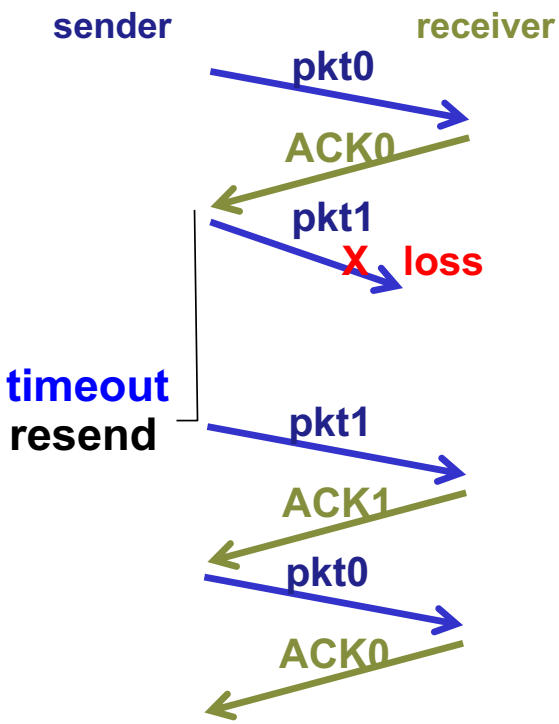
A lost packet is retransmitted at timeout of countdown timer

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout

Operation with no loss



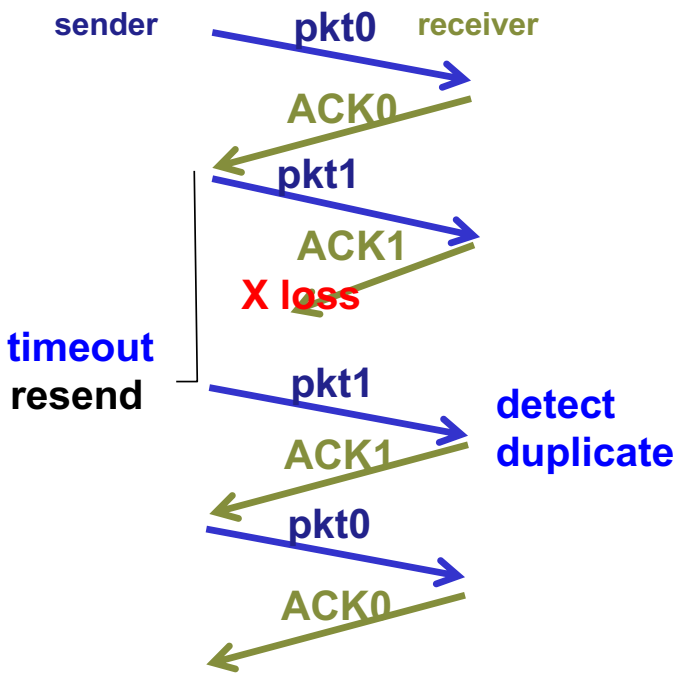
Operation with packet loss



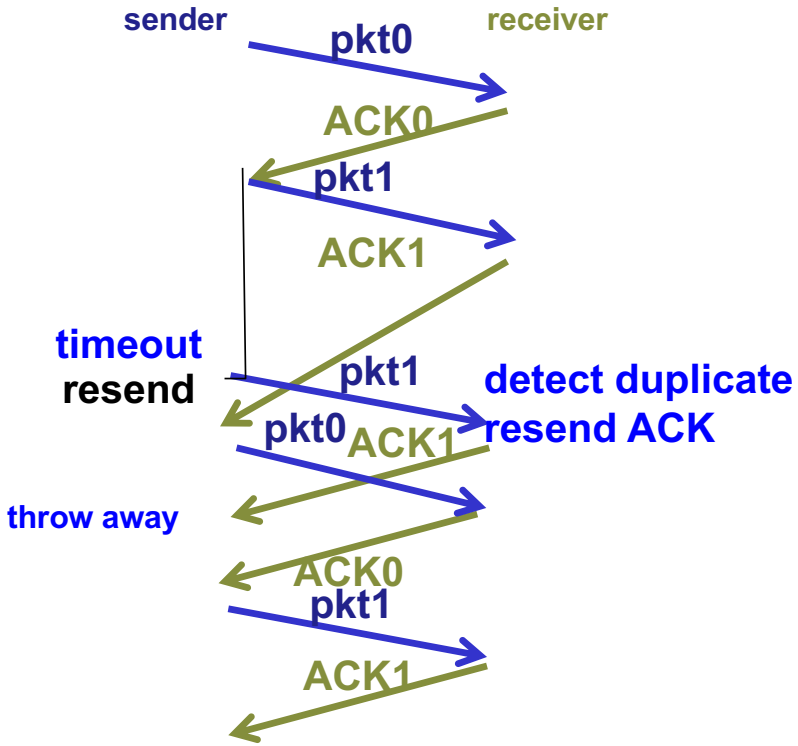
Duplicate packets arrive at the receiver

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout

Operation with lost ACK



Operation with premature timout

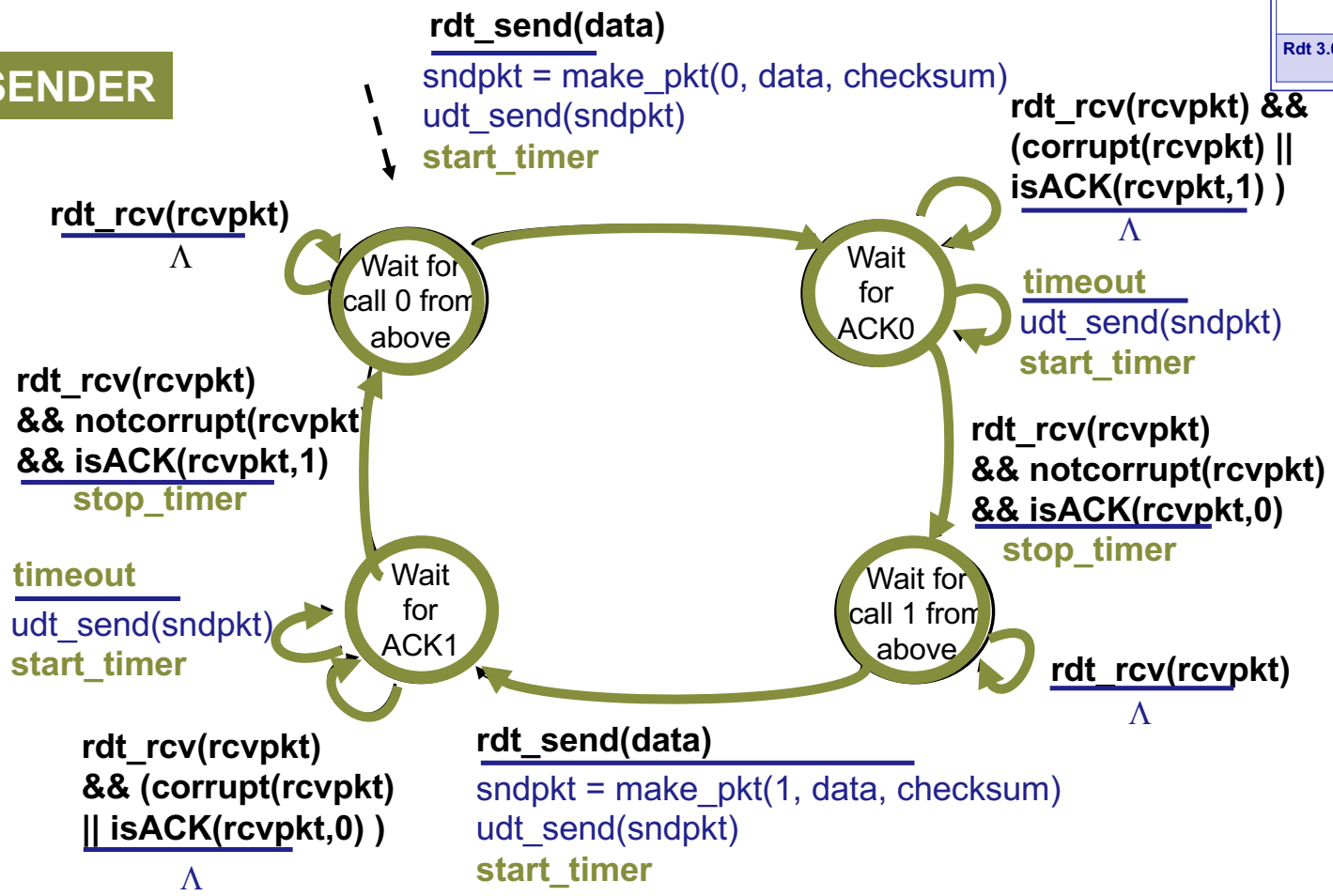


Reliable data transfer 3.0

Sender includes retransmission timer

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout

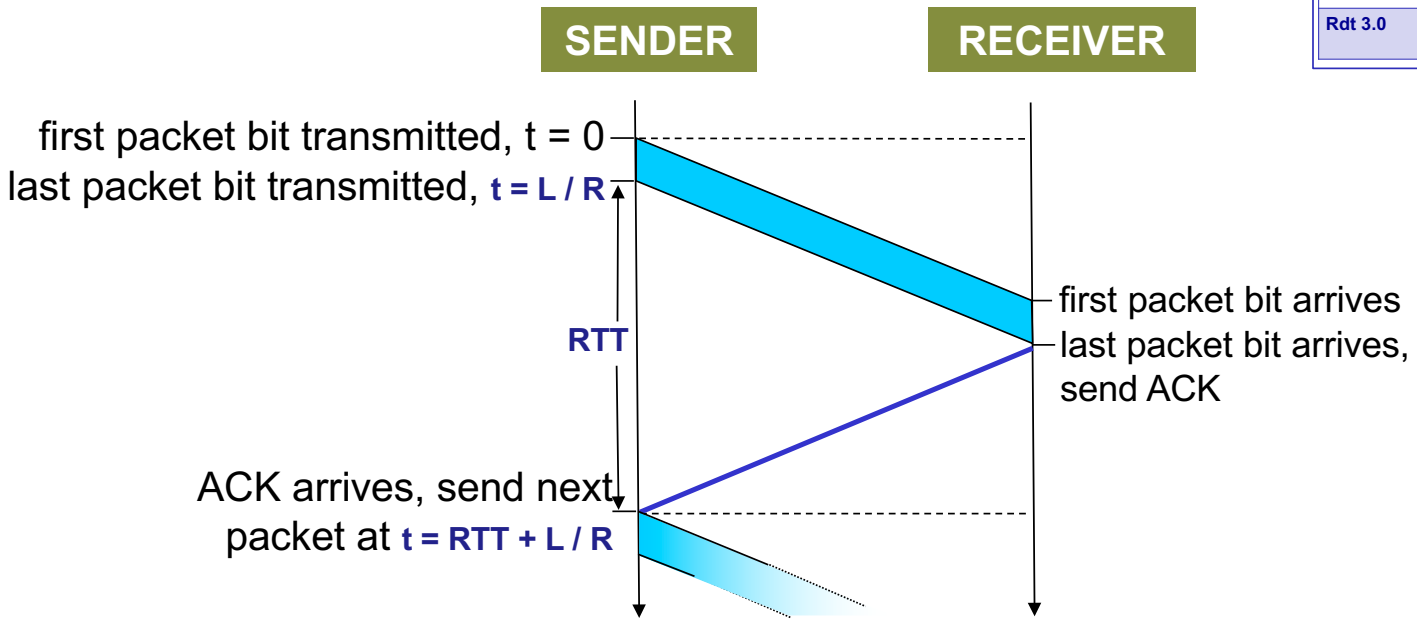
SENDER



event
actions

Rdt3.0: stop-and-wait operation

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout



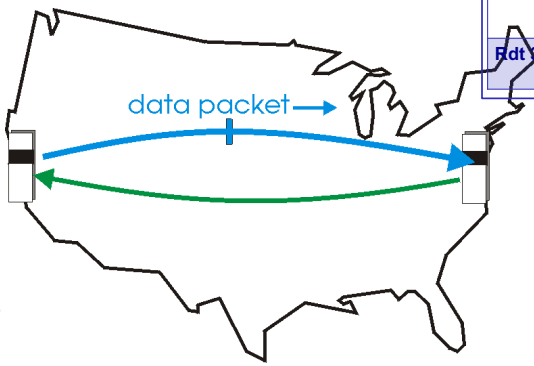
L=Length of packet

R=Rate of transmission

The protocol has a lousy performance

Protocol	Channel	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error S-> R; only data corrupt	checksum, receiver feedback (ACK, NAK), retransmission on NAK
Rdt 2.1	Bit error S <-> R; ACK NAK may also be corrupt	seq# on data packets, checksum on ACK/NAK
Rdt 2.2	Bit error S <-> R	no NAK, seq# also in ACKS, retransmission on duplicate ACK
Rdt 3.0	Bit error S <-> R, packet loss	countdown timer, retransmission on timeout

1 Gbit/s link (R)
15 ms propagation delay (0,5*RTT)
1000 byte (8000 bits) packet length (L)



$$d_{trans} = \frac{L}{R} = \frac{8000\,\text{bits}}{10^9\,\text{bps}} = 8\,\text{microseconds}$$

- **U_{sender}: Utilization** – fraction of time sender busy sending

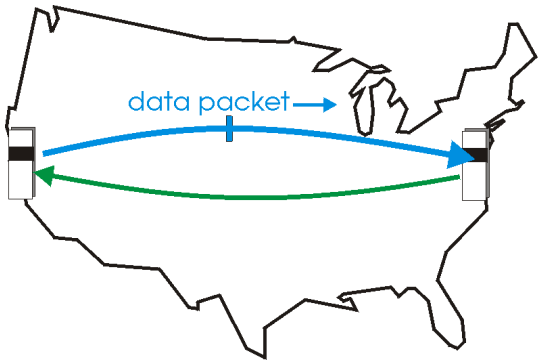
$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = .00027$$

- 1000 bytes every 30 msec
  -> 267 kbit/s throughput over 1 Gbps link

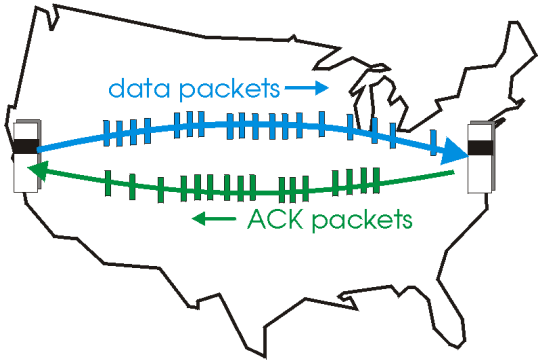
Protocol limits use of physical resources!

Pipelined protocols: sender allows multiple, “in-flight” packets

Stop-and-wait

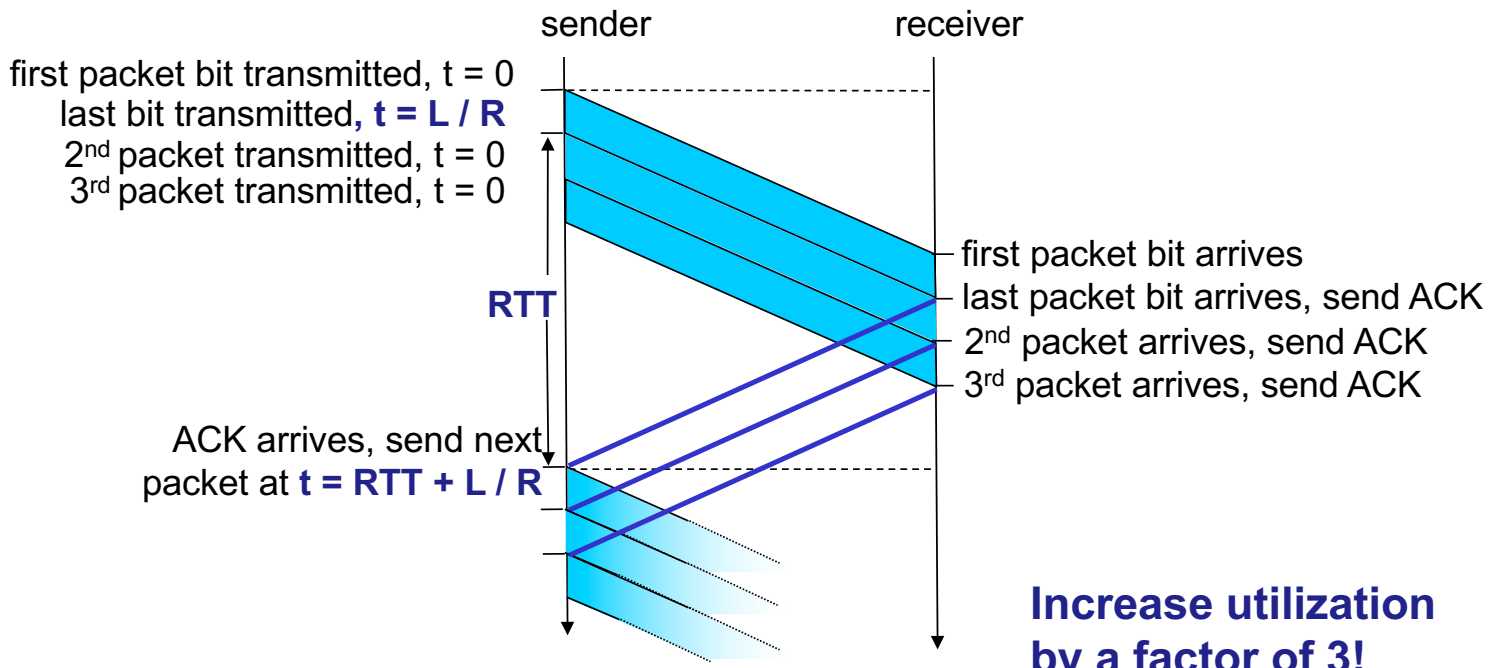


Pipelined



- Packets in flight are yet-to-be acknowledged
- Range of sequence numbers must be increased
- Buffering at sender and/or receiver

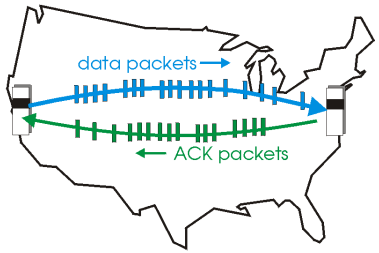
Pipelining: increased utilization



Increase utilization by a factor of 3!

$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

Pipelining protocols – two retransmission strategies



Go-back-N retransmission

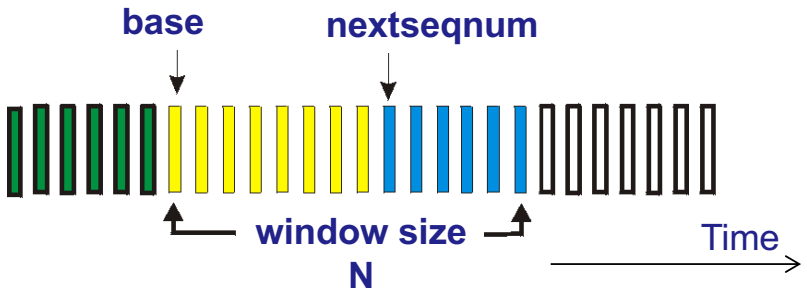
- Sender can have **up to N unacked packets** in pipeline
- Receiver sends **cumulative acks**
  - Doesn't ack packet if there's a gap in received packets
- Sender has **timer for oldest** unacked packet
  - If timer expires, retransmit all unacked packets

Selective repeat retransmission

- Sender can have **up to N unacked packets** in pipeline
- Receiver **acks individual** packets
- Sender maintains **timer for each** unacked packet
  - When timer expires, retransmit only unacked packet

The senders view of sequence numbers

- k-bit sequence number in packet header
- "**Window**" of up to N, consecutive unack'ed packets allowed



- "**Cumulative ACK**": ACKs all packets up to, including sequence number n
  - may receive duplicate ACKs
- Timer for each in-flight packet, only one countdown timer
- **Timeout(n)***:* retransmit packet n and all higher seq # pkts in window

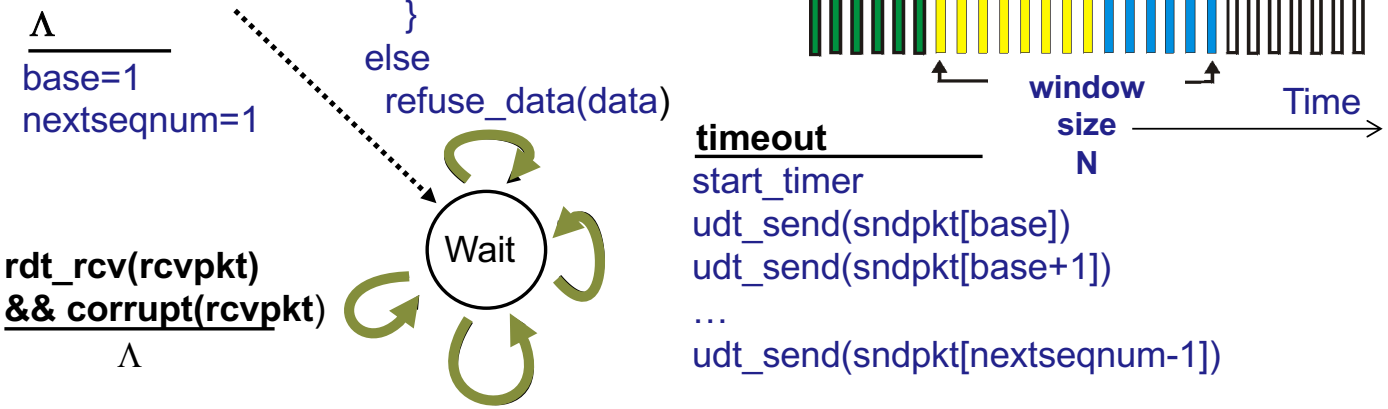
Go-Back-N: sender-extended FSM

event
actions

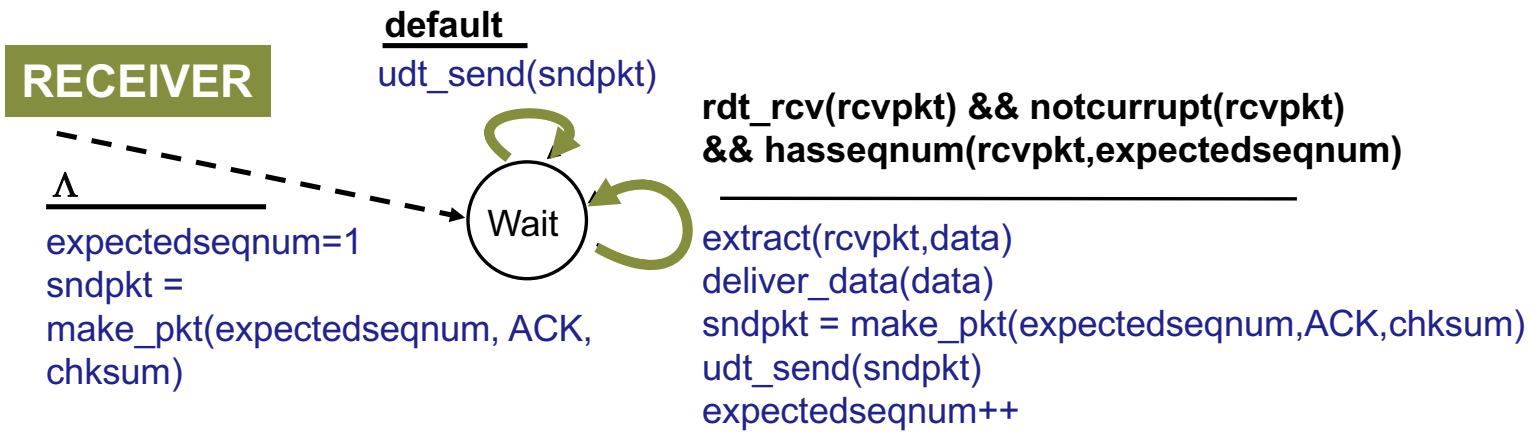
SENDER

rdt_send(data)

```
if (nextseqnum < base+N) {
   sndpkt[nextseqnum] = make_pkt(nextseqnum,data,chksum)
   udt_send(sndpkt[nextseqnum])
   if (base == nextseqnum)
      start_timer
   nextseqnum++
   }
else
  refuse_data(data)
```

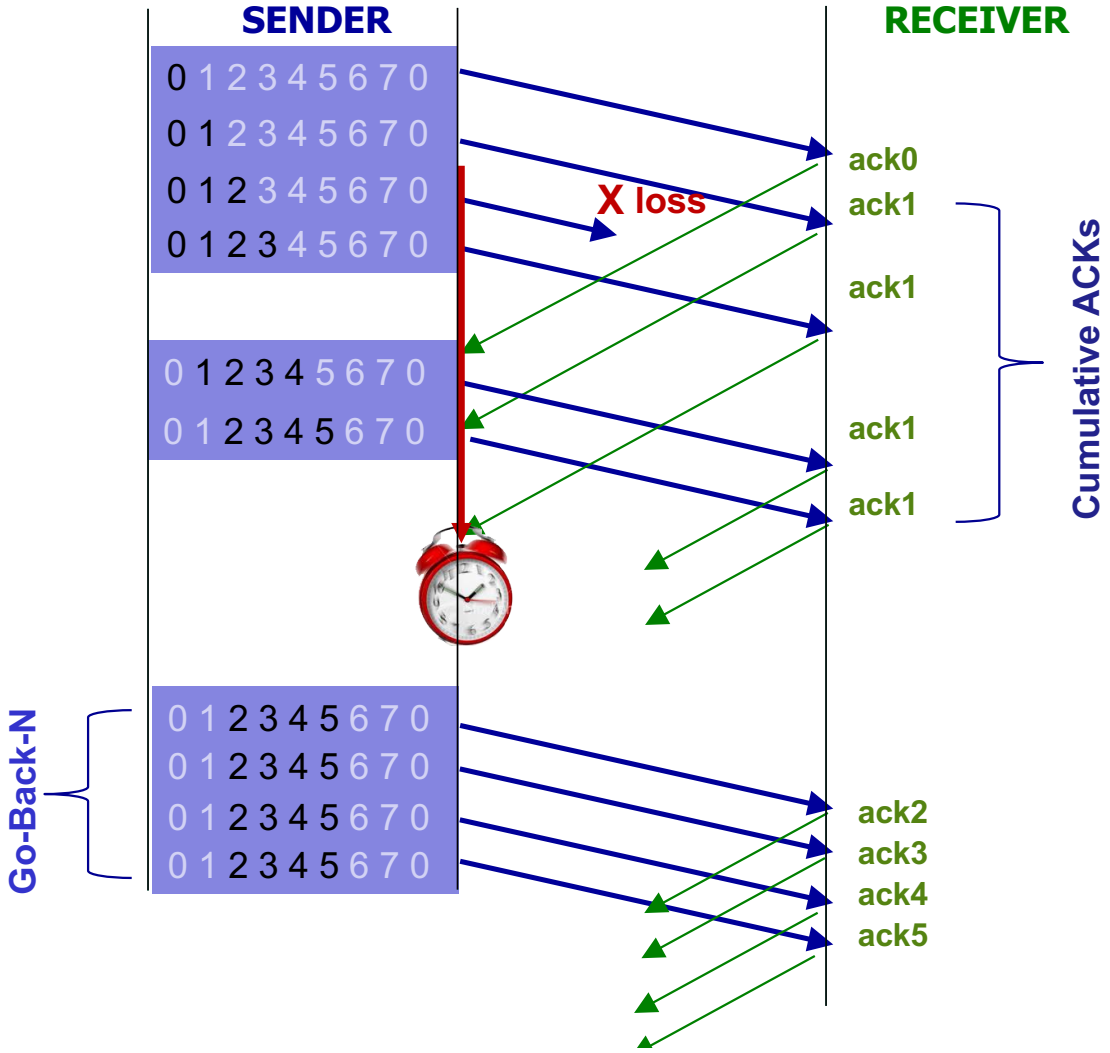


Go-Back-N: receiver-extended FSM



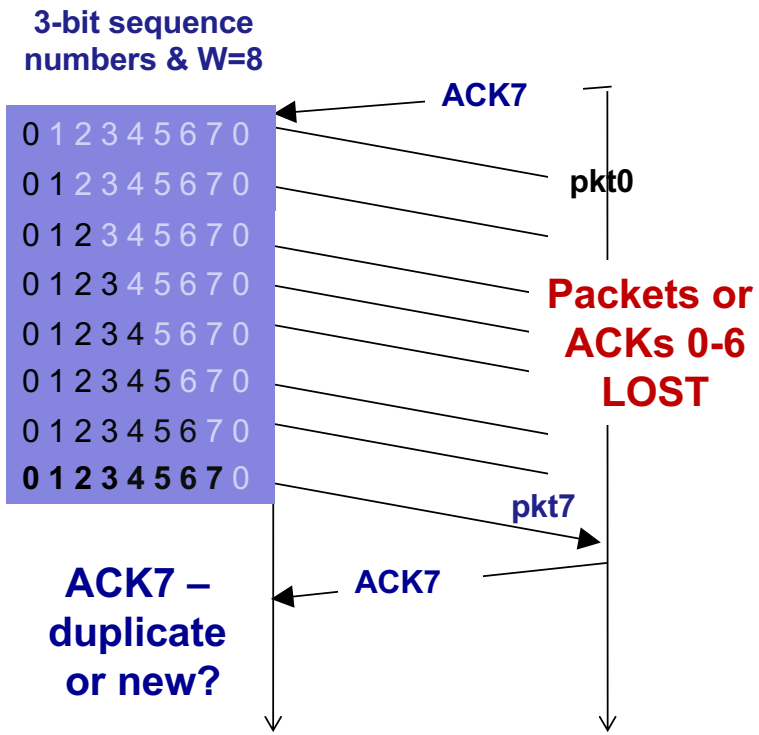
- ACK-only
  - always sends ACK for correctly-received packets with highest in-order sequence number
  - may generate duplicate ACKs
  - needs only remember expected sequence number
- Out-of-order packets
  - Discard (don't buffer) -> no receiver buffering!
  - Re-ACK packets with highest in-order sequence number

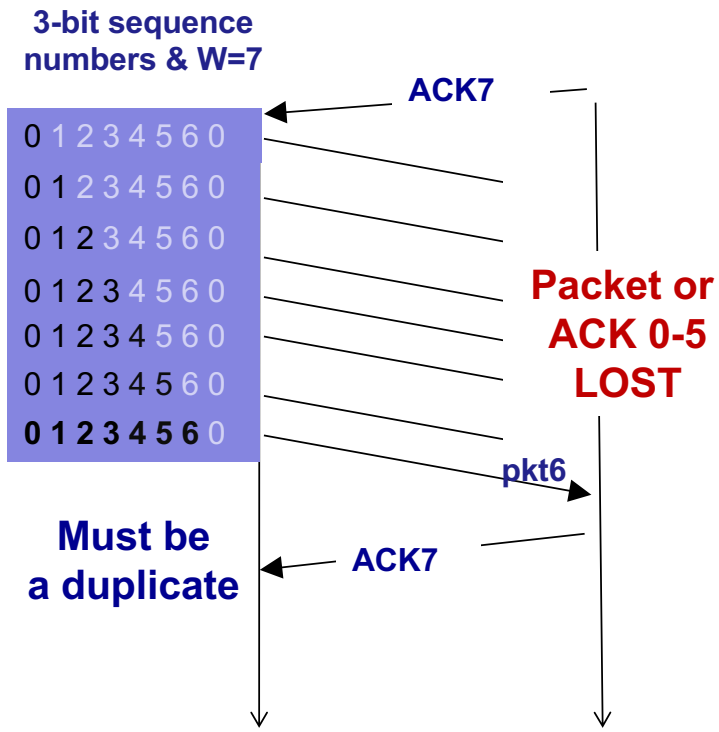
Go-Back-N retransmission in action



Max window =4 packets
3 bit sequence number

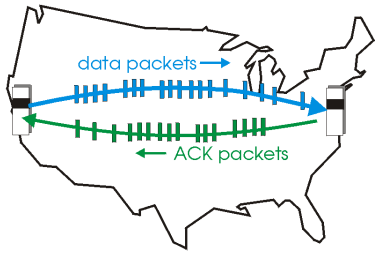
”Go-back-N”: maximum window size < sequence number space





With a n-bit sequence number field, the size of the sequence number space = 2^n. Maximum window = $2^n - 1$

Pipelining protocols – two retransmission strategies



Go-back-N retransmission

- Sender can have **up to N unacked packets** in pipeline
- Receiver sends **cumulative acks**
  - Doesn't ack packet if there's a gap in received packets
- Sender has **timer for oldest** unacked packet
  - If timer expires, retransmit all unacked packets

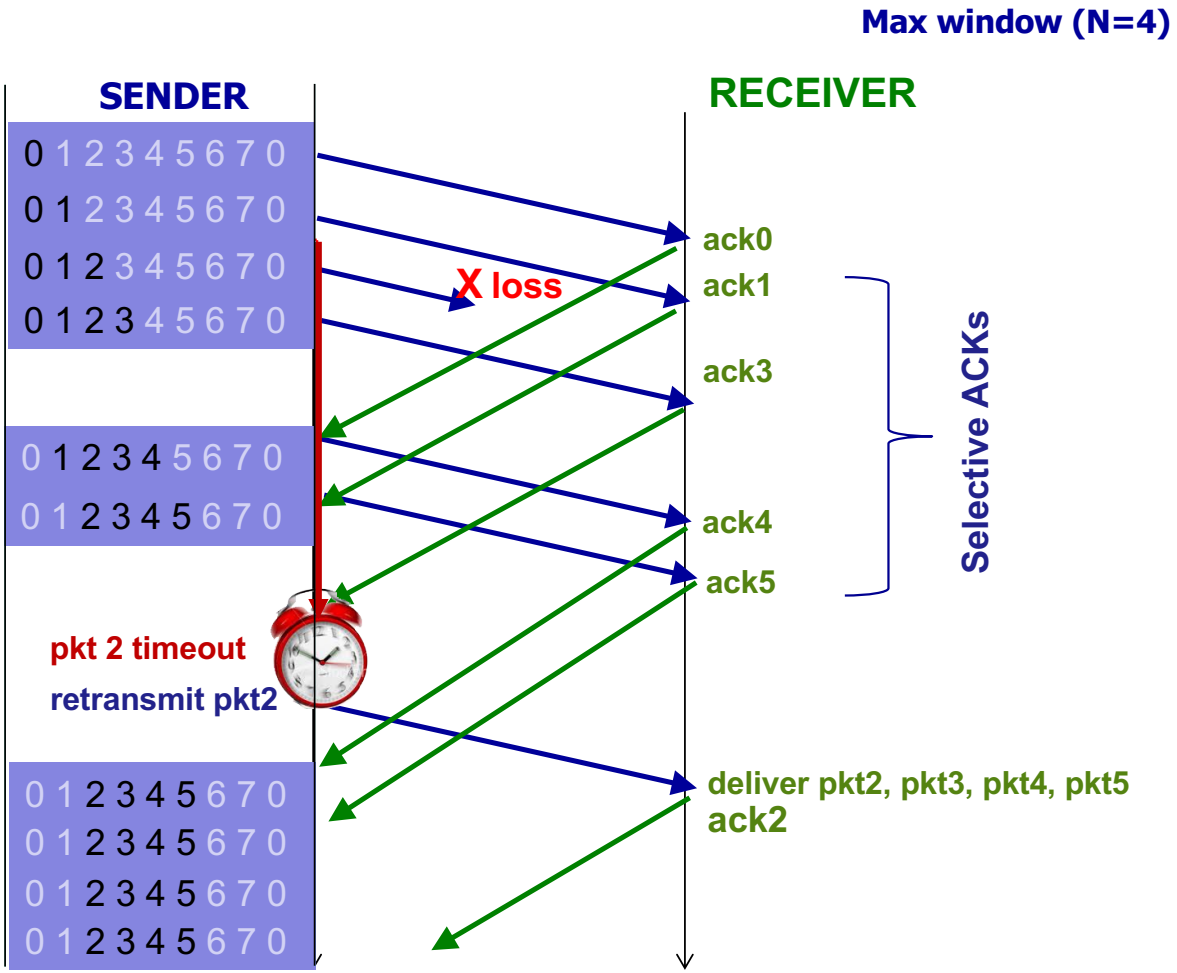
Selective repeat retransmission

- Sender can have **up to N unacked packets** in pipeline
- Receiver **acks individual** packets
- Sender maintains **timer for each** unacked packet
  - When timer expires, retransmit only unacked packet

Selective repeat retransmission: sender only resends unacknowledged packets

- **Sender only resends packets for which ACK not received**
  - sender timer for each unACKed packet
- Sender window
  - N consecutive sequence number
  - again limits sequence numbers of sent, unACKed packets
- **Receiver individually acknowledges all correctly received packets**
  - buffers packets, as needed, for eventual in-order delivery to upper layer

Selective repeat retransmission in action

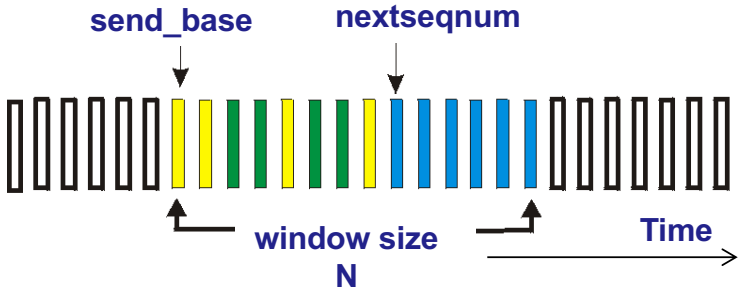


What does sender transmit when ack2 arrives?

Selective repeat retransmission

Events and actions at sender

SENDER view of sequence number



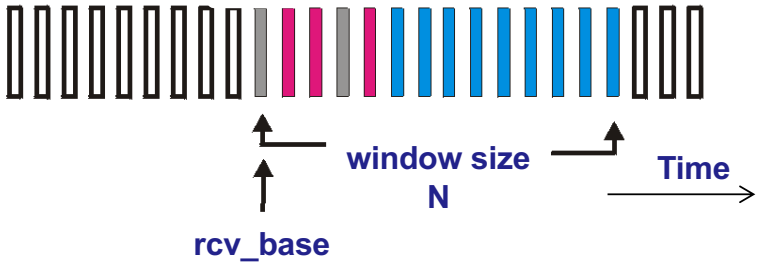
1. Data received from above
   - if next available seq # in window, send packet
2. Timeout(n)
   - resend pkt n, restart timer
3. ACK(n) in [sendbase,sendbase+N]
   - mark pkt n as received (change colour from yellow to green)
   - if n smallest unACKed pkt, advance **send_base** to next unACKed seq #

NTNU

Selective repeat retransmission

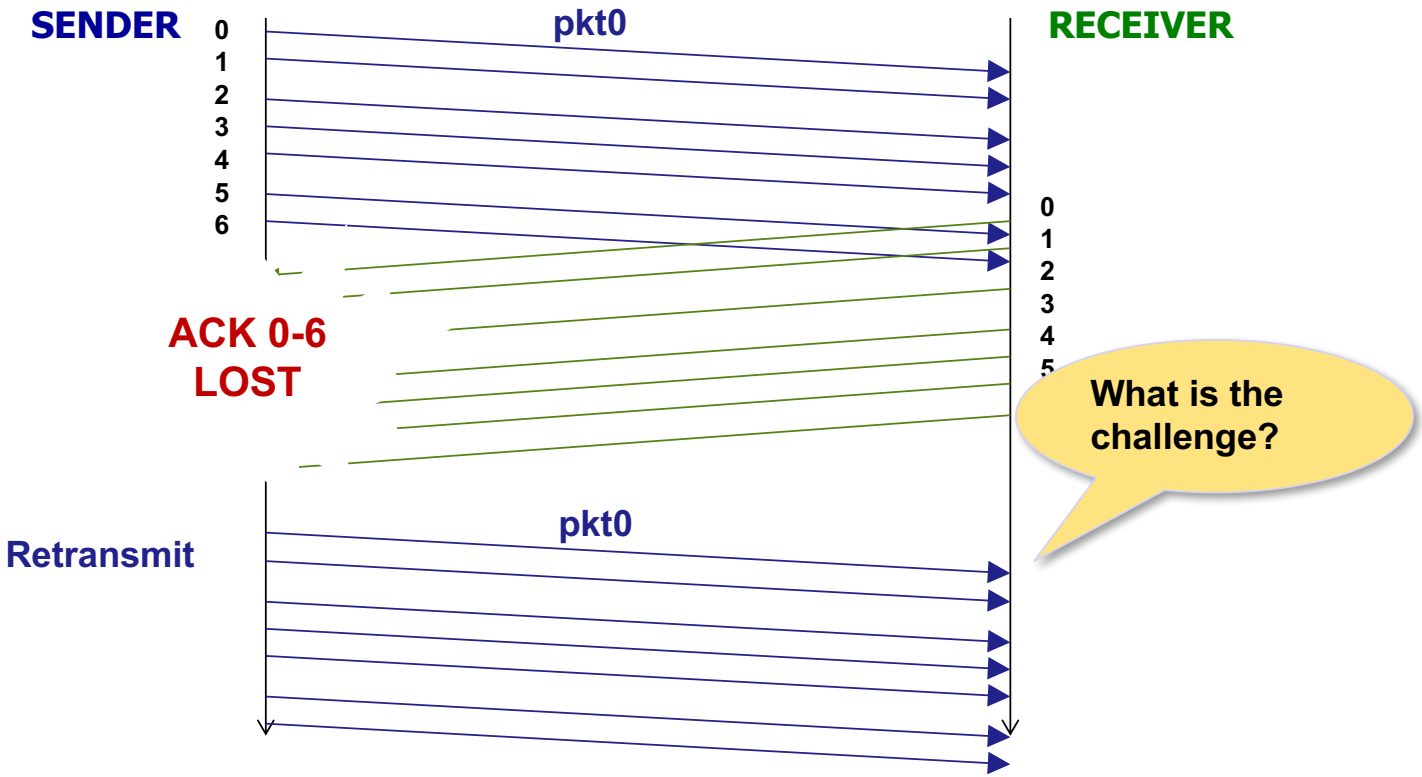
Events and actions at receiver

RECEIVER view of sequence number



1. Packet(n) in [rcvbase, rcvbase+N-1] - within window
   - send ACK(n)
   - out-of-order: buffer packet
   - in-order: deliver all buffered, in-order packets, advance **rcv_base** to next not-yet-received packet
2. Packet(n) in [rcvbase-N, rcvbase-1] - duplicate
   - ACK(n) (also previously acked)
3. Otherwise ignore the packet

Maximum window for selective repeat retransmission is half the sequence number space



With n-bit sequence number field, the size of the sequence number space = 2^n . Maximum window = $2^{(n-1)}$

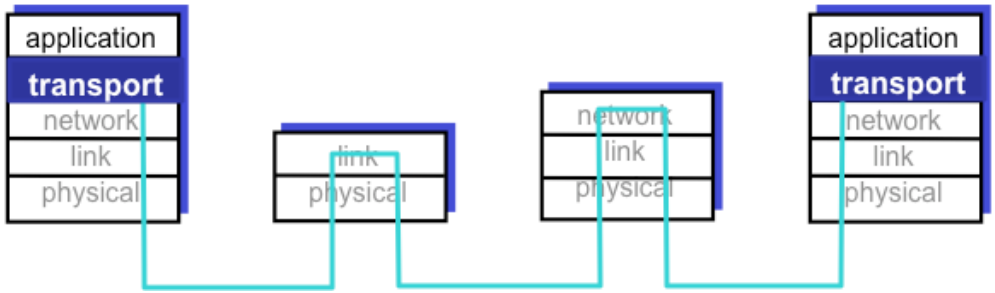
NTNU

Summary: reliable data transfer (rdt) between sender (S) and receiver (R)

Protocol	Channel charact.	Protocol functions added
Rdt 1.0	Error free	
Rdt 2.0	Bit error **S**-> **R**; only data corrupt	**S:** computes **data checksum** **R:** feedback (**ACK**, **NAK**) **S:** **retransmission** on NAK
Rdt 2.1	Bit error **S** <-> **R**; ACK NACK may also be corrupt	**S:** **sequence number** on data packets, **R:** **checksum** on ACK/NAK
Rdt 2.2	Bit error **S** <-> **R**	**no NAK** **R:** **sequence number** also **in ACKs** **S:** **retransmission on duplicate ACK**
Rdt 3.0	**Bit error S <-> R, packet loss**	**S:** **countdown timer** **R:** retransmission on timeout

Window: **for pipelined transmission**

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: Transmission Control Protocol

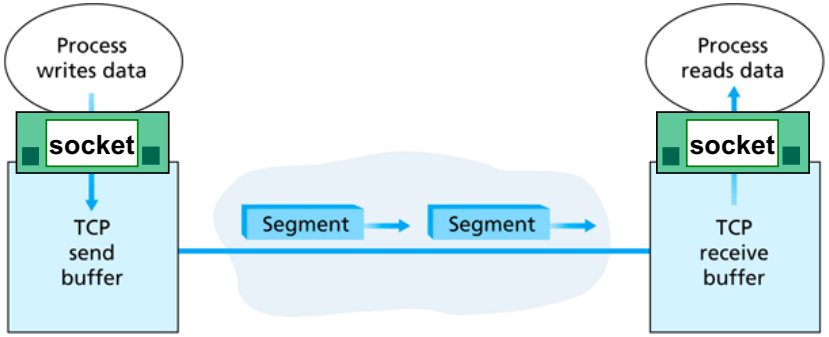
- **TCP segment structure**
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

TCP – Transmission Control Protocol

- **Point-to-point**
  - one sender, one receiver
- **Connection-oriented**
  - handshaking
    initiates sender and receiver state before data exchange
- **Error controlled**
  - bit error, segment error
- **Flow controlled**
  - sender will not
    overwhelm receiver
- **Congestion control**
  - sender adjust transmission dependent on network load

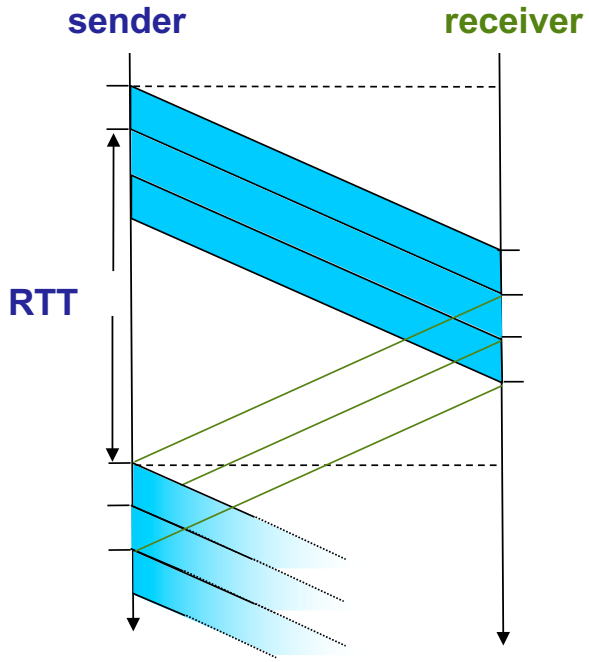


- **Full duplex** data
  - bi-directional data flow in same connection
  - **MSS** = maximum segment size

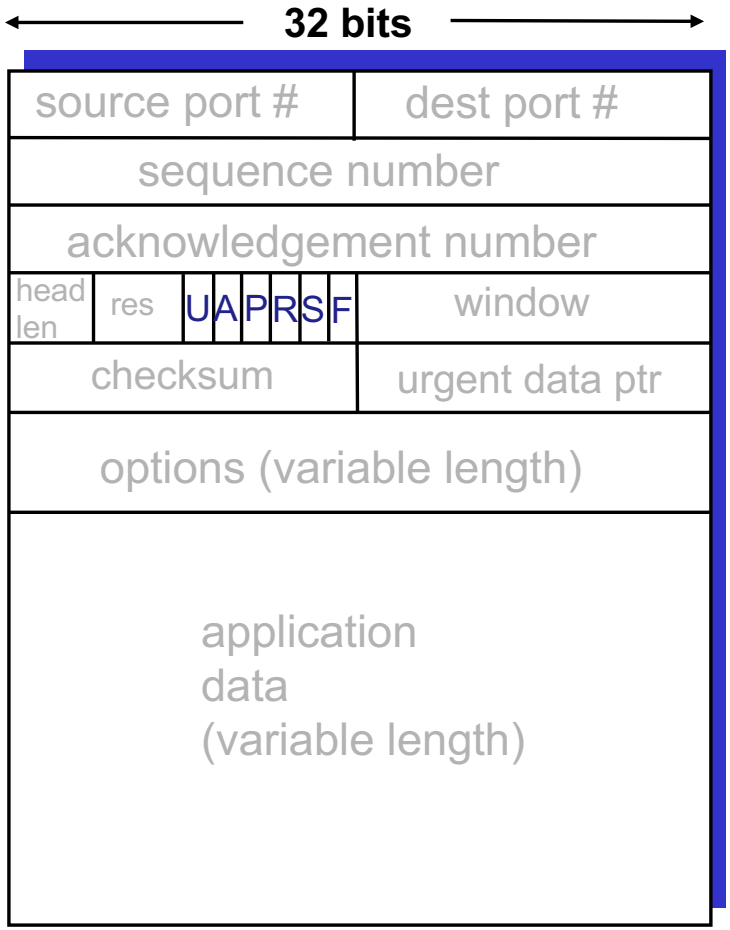
TCP – Transmission Control Protocol

- **Reliable**, in-order **byte stream**
  - no “message boundaries”

- **Pipelined segments**
  - send & receive buffers
  - TCP congestion and error/flow control set window size



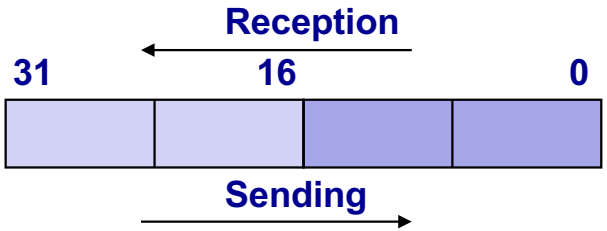
A TCP header is 20 bytes (without options)



- **`SrcPort & DstPort`** identify higher layer protocol/application
- **`Sequence number, Acknowledgement`** number the byte-stream
- **`Head len:`** number of 32-bit words
- **`Window`**: receiver advertised window size
- **`Res`**: reserved
- Flags: **`URG`**, **`ACK`**, **`PUSH`**, **`RESET`**, **`SYN`**, **`FIN`**
- **`Checksum`**: internet checksum (TCP "header" + data + pseudo "header" (IP src, IP dst, IP length))
- **`Urgent data pointer`**: sequence number of the final octet of urgent data if the urgent flag is set (generally not used)

But high bandwidth networks make end systems often use "scalable window" option

- TCP window field is 16 bits
- Scale by doing a shift operation on announced window



source port #	dest port #
sequence number	
acknowledgement number	
	window
checksum	urg pointer

- Shift up to 14 bits, max window = $64k*2^{14}$
  - Internally the window is maintained as a 32-bit value
  - Left shift when receiving the window
  - Right shift when sending the window size
- Scalable window option is sent in SYN segment
  - Scaling factor set when TCP-connection established
  - Scaling factor can be asymmetric

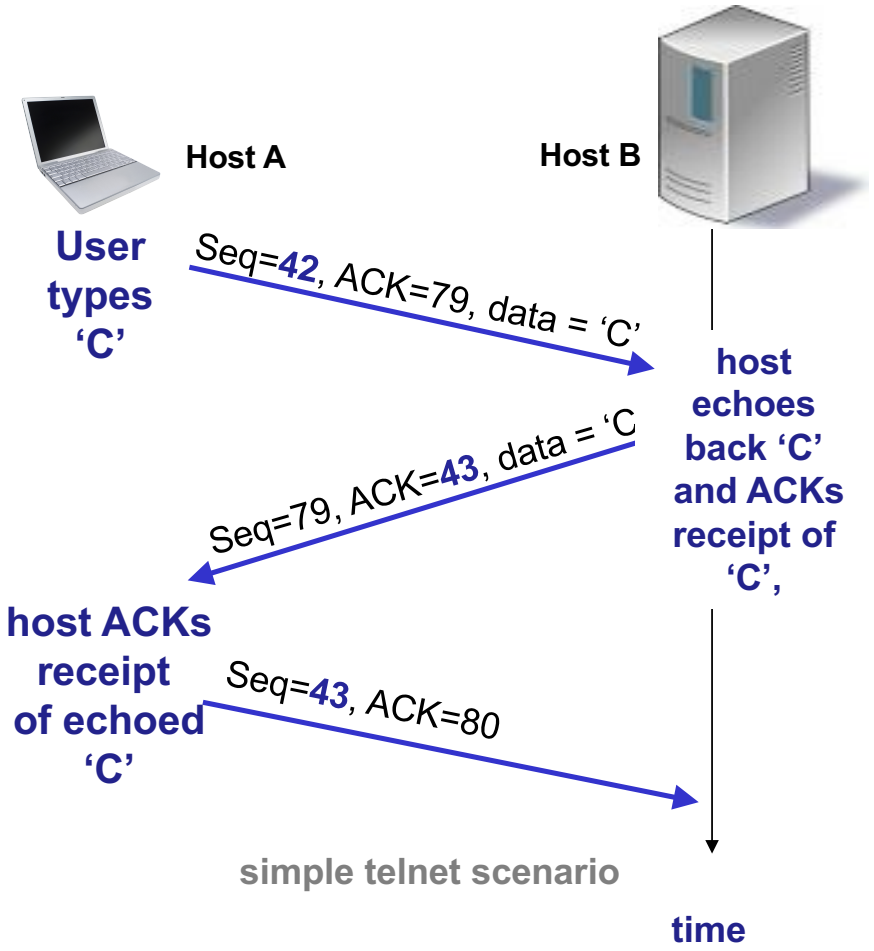
TCP sequence numbers from sender to receiver
TCP ACKs from receiver to sender

Sequence number

- Numbers the **first byte** of the **segment's** data

ACK

- **Cumulative** ACK
- Sequence number of **next byte expected** from other side

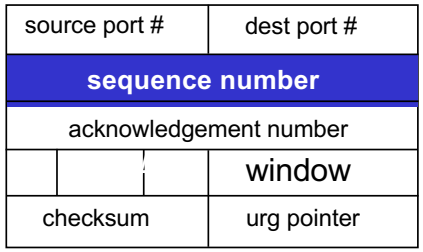


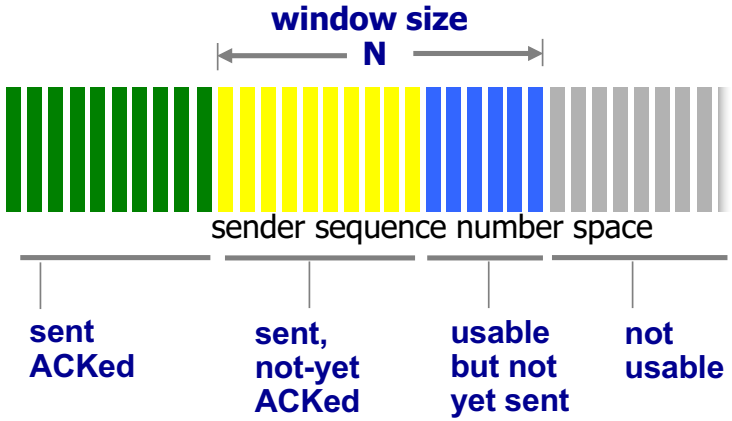
TCP sequence number and TCP ACK enumerate bytes

Sequence number

- Numbers the **first byte** of the **segment's** data

Sender sets the byte stream sequence number

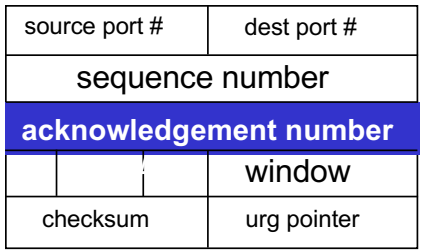




ACK

- **Cumulative** ACK
- Sequence number of **next byte expected** from other side

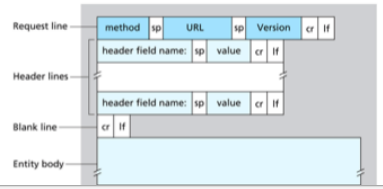
Receiver sets the acknowledgement number



HTTP over TCP

HTTP:

- HTTP: pull
- (non) persistent connections
- each object encapsulated in its own response msg
- **End-of HTTP request:** extra CRLF **"0d 0a"**
- **End-of HTTP response:** `Content-Length`

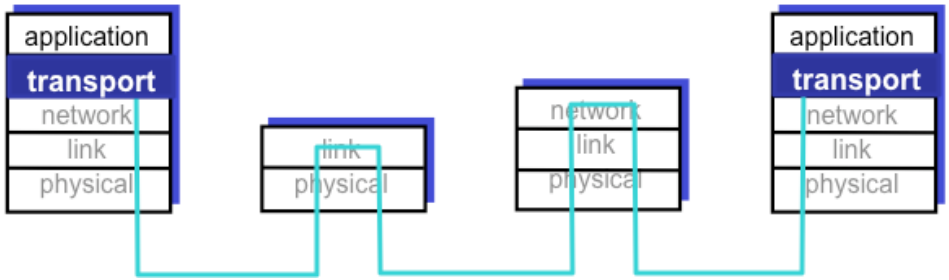


How does HTTP use TCP for transport of messages?

No.	Time	Source	Destination	Protocol	Length	Info
256	13.748301000	192.168.79.165	184.86.13.15	HTTP	978	GET / HTTP/1.1
260	13.775576000	184.86.13.15	192.168.79.165	HTTP	1418	HTTP/1.1 200 OK (text/html)
455	13.869903000	192.168.79.165	80.239.148.216	HTTP	950	GET /v/home/ak/styles/billboard.css HTTP/1.1
466	13.877684000	192.168.79.165	80.239.148.216	HTTP	936	GET /global/scripts/lib/prototype.js HTTP/1.1
467	13.878646000	192.168.79.165	80.239.148.216	HTTP	945	GET /home/styles/billboard.css HTTP/1.1
468	13.880026000	192.168.79.165	80.239.148.216	HTTP	942	GET /global/styles/base.css HTTP/1.1
470	13.881266000	192.168.79.165	80.239.148.216	HTTP	945	GET /v/home/ak/styles/home.css HTTP/1.1
471	13.882137000	192.168.79.165	80.239.148.216	HTTP	952	GET /global/nav/styles/navigation.css HTTP/1.1
476	13.920731000	80.239.148.216	192.168.79.165	HTTP	1061	HTTP/1.1 200 OK (text/css)
483	13.930890000	80.239.148.216	192.168.79.165	HTTP	963	HTTP/1.1 200 OK (text/css)
523	13.940258000	80.239.148.216	192.168.79.165	HTTP	83	HTTP/1.1 200 OK (text/css)
530	13.940263000	80.239.148.216	192.168.79.165	HTTP	919	HTTP/1.1 200 OK (text/css)
547	13.944079000	80.239.148.216	192.168.79.165	HTTP	1461	HTTP/1.1 200 OK (text/css)
564	13.991070000	192.168.79.165	80.239.148.216	HTTP	938	GET /global/ac_base/ac_base.js HTTP/1.1
586	14.051726000	80.239.148.216	192.168.79.165	HTTP	135	HTTP/1.1 200 OK (application/x-javascript)
617	14.647362000	192.168.79.165	80.91.37.212	HTTP	379	GET / HTTP/1.1
620	14.656682000	192.168.79.165	176.34.103.144	HTTP	1202	GET /log?type=activity&sn=38&ct=22718&pi=MYZON3BKPFUQ&mk=DAGBUIQQVCVC
622	14.663780000	80.91.37.212	192.168.79.165	HTTP	701	HTTP/1.1 301 Moved Permanently (text/html)
627	14.686150000	192.168.79.165	80.91.37.212	HTTP	604	GET / HTTP/1.1
639	14.720549000	176.34.103.144	192.168.79.165	HTTP	200	HTTP/1.1 204 No Content
744	14.797502000	80.91.37.212	192.168.79.165	HTTP	810	HTTP/1.1 200 OK (text/html)

How does HTTP pair requests and responses?

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: Transmission Control Protocol

- segment structure
- **TCP reliable data transfer**
- flow control
- connection management

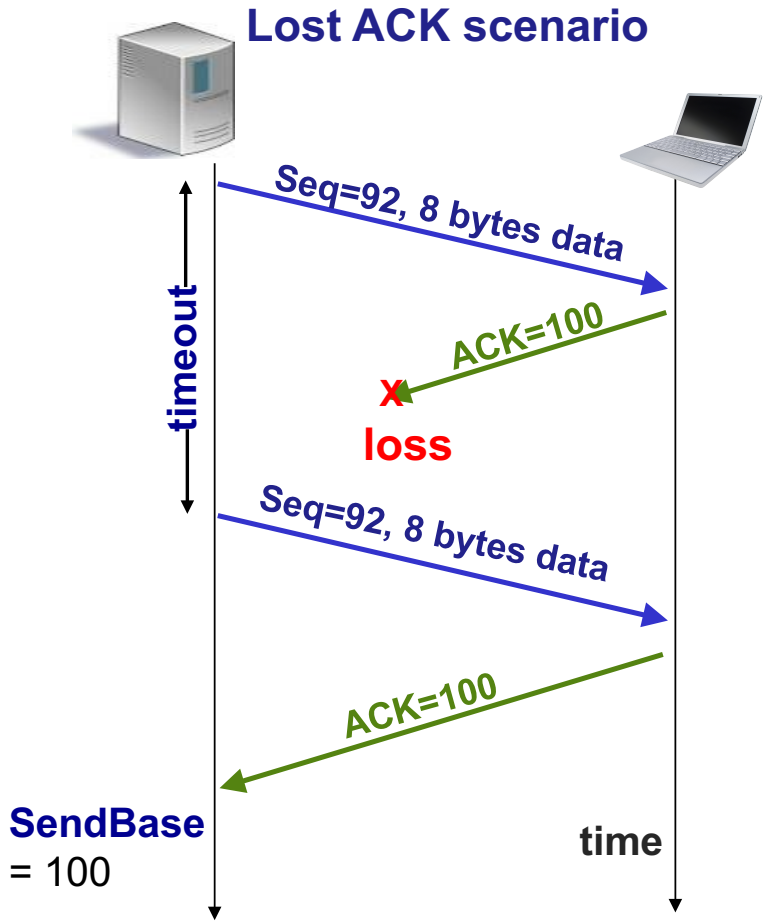
3.6 Principles of congestion control

3.7 TCP congestion control

TCP reliable data transfer on top of IP's unreliable service

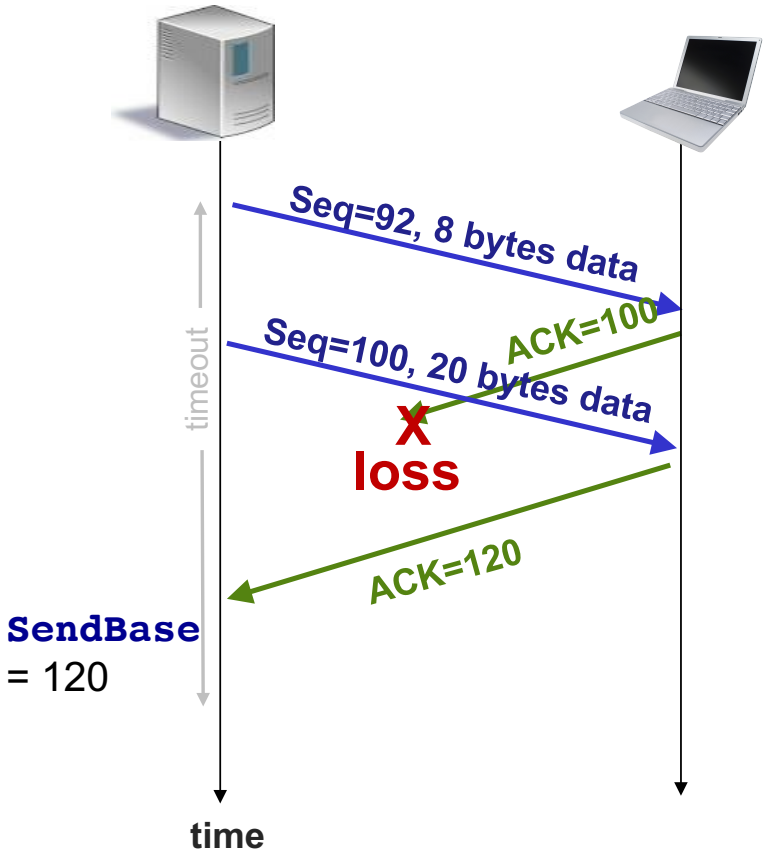
- **Retransmissions** triggered by
  - timeout
  - duplicate acks
- Single retransmission timer
  - Go-back-N retransmission

- Initially consider simplified TCP sender
  - ignore duplicate ACKs
  - ignore flow control, congestion control

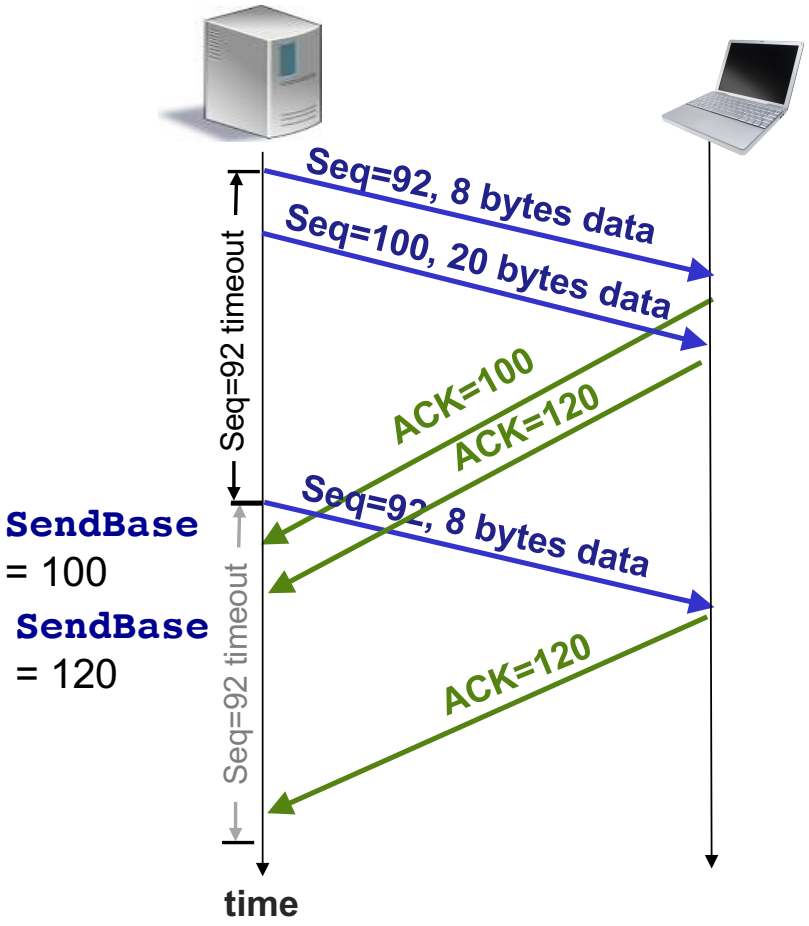


TCP reliable data scenarios

Cumulative ACK scenario



Premature timeout



Receiver events and actions

TCP ACK generation at a glance

event
actions

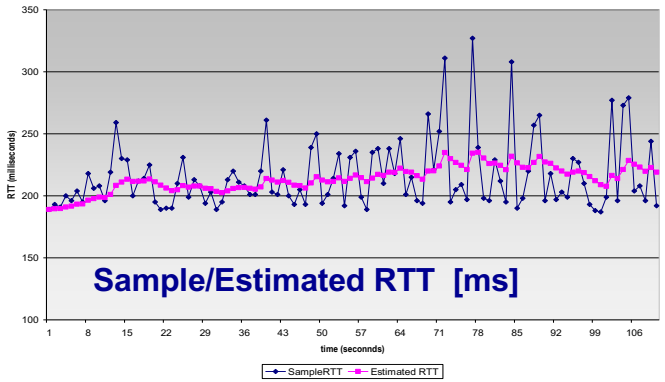
Event at TCP receiver	Receiver TCP actions
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed.	**Delay ACK**, wait up to 500 ms for next in-order segment. If no next segment, send ACK.
Arrival of in-order segment with expected seq #. One other segment has ACK pending.	Immediately send single **cumulative ACK**, ACKing both in-order segments.
Arrival of out-of-order segment higher-than-expect seq # . Gap detected.	Immediately send ***duplicate ACK***, indicating seq # of next expected byte (lower end of gap).
Arrival of segment that partially or completely fills gap.	Immediate send ACK, provided that segment starts at lower end of gap.

[RFC 1122, RFC 2581]

TCP round trip time (RTT) and timeout

How to set TCP timeout value?

- Longer than RTT (that varies)
- Too short: premature timeout unnecessary retransmission
- Too long: slow reaction to loss



How to estimate RTT?

- **`SampleRTT`**: measured time from segment transmission until ACK receipt
  - ignore retransmissions
- **`SampleRTT`** will vary, want estimated RTT "smoother"
  - exponential weighted moving average: average recent measurements

$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$

(typically, $\alpha = 0.125$)

TCP round trip time, timeout interval

- Timeout interval: **EstimatedRTT** plus "safety margin"
  - large variation in **EstimatedRTT** -> larger safety margin

- Estimate **SampleRTT** deviation, **DevRTT**, from **EstimatedRTT**

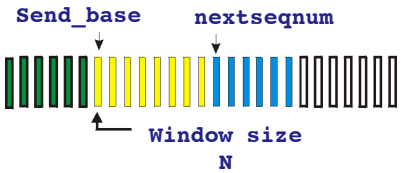
$$DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|$$

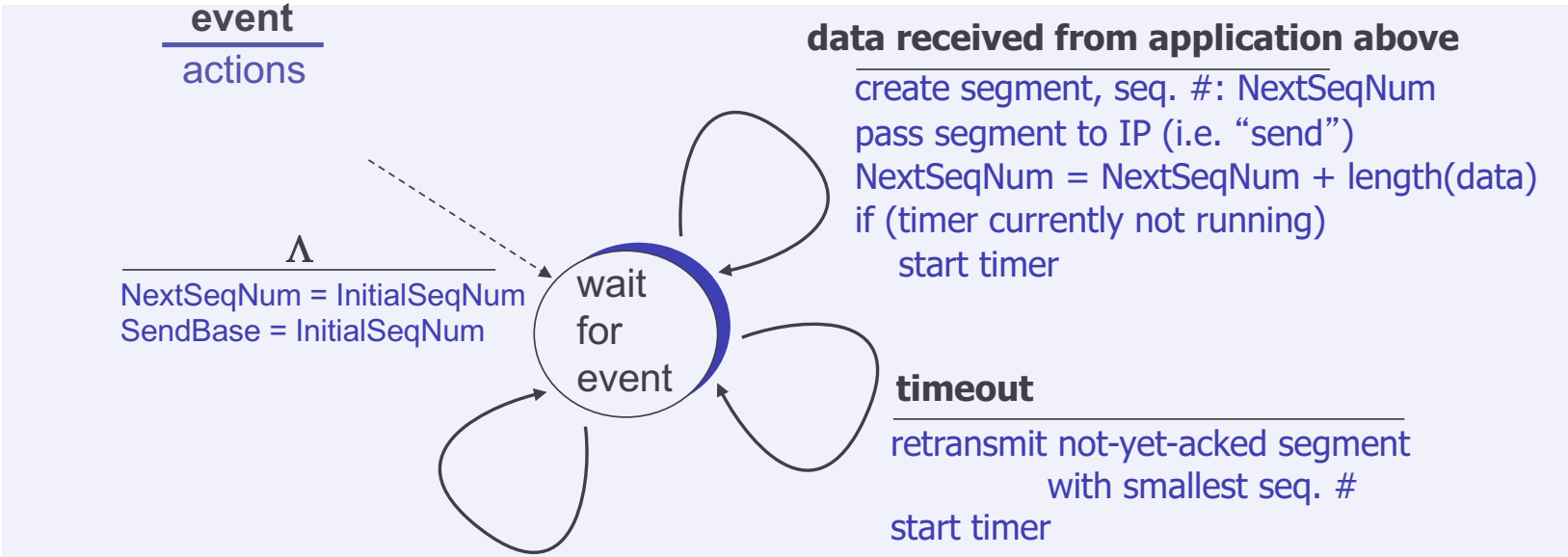
(typically, $\beta = 0.25$)

"safety margin"

$$TimeoutInterval = EstimatedRTT + 4*DevRTT$$

TCP sender (simplified)

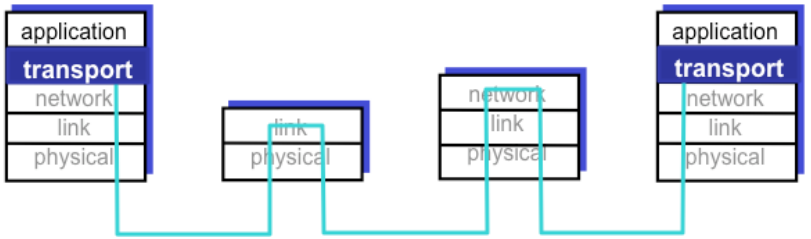




ACK received, with ACK field value y

```
if (y > SendBase) {
   SendBase = y
   /* SendBase–1: last cumulatively ACKed byte */
   if (there are currently not-yet-acked segments)
      start timer
     else stop timer
    }
```

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: Transmission Control Protocol

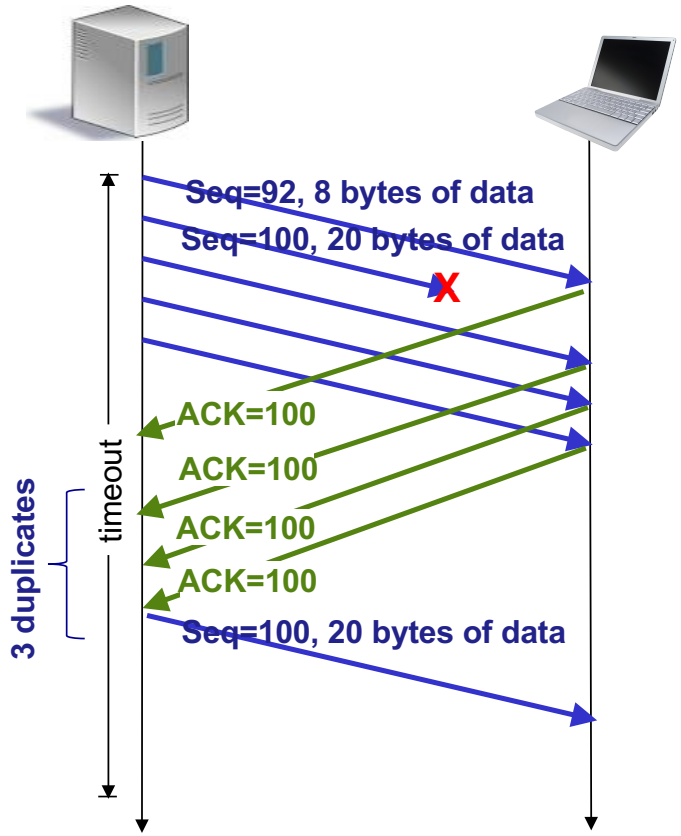
- TCP segment structure
- **reliable data transfer**
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

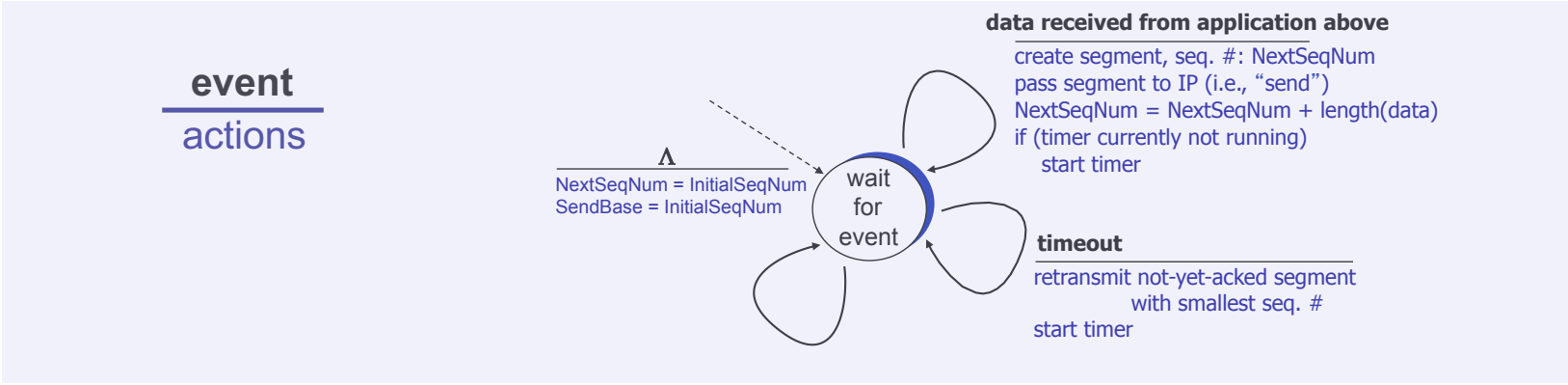
TCP fast retransmit

- **Challenge**: Time-out period often relatively long
  - long delay before resending lost packet
- **Solution**: Detect lost segments via duplicate ACKs
  - sender often sends many segments back-to-back
  - if one segment is lost, there will likely be many duplicate ACKs



Fast retransmit after sender receipt of triple duplicate ACKs

TCP sender handling fast retransmit



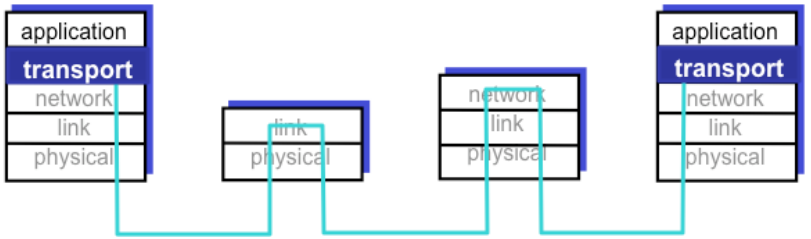
event: ACK received, with ACK field value of y

```
if (y > SendBase) {
 SendBase = y
 if (there are currently not-yet-acknowledged segments)
      start timer
}
else {
      increment count of dup ACKs received for y
      if (count of dup ACKs received for y = 3) {
           resend segment with sequence number y
       }
```

a duplicate ACK for already ACKed segment

fast retransmit

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: Transmission Control Protocol

- segment structure
- reliable data transfer
- **TCP flow control**
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

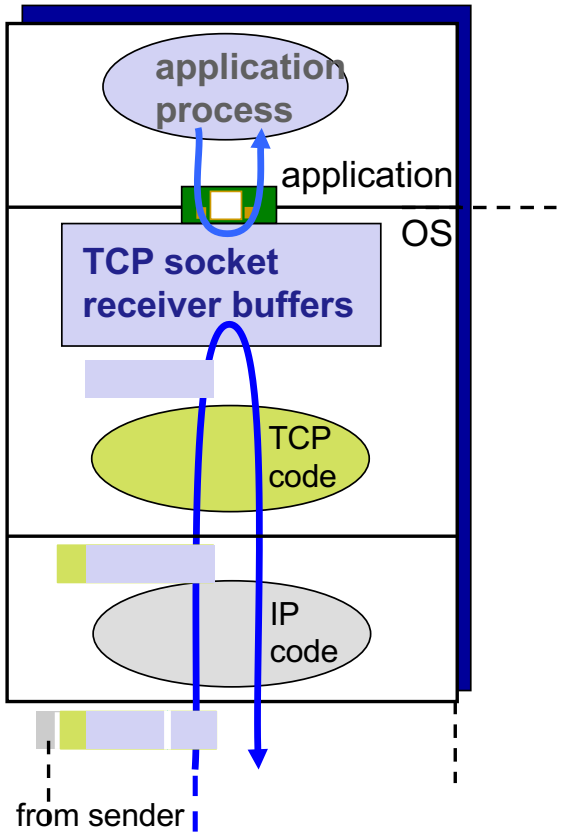
TCP flow control assures no overflow of receiver buffer

flow control

receiver controls sender, so sender won't overflow receiver's buffer by transmitting too much, too fast

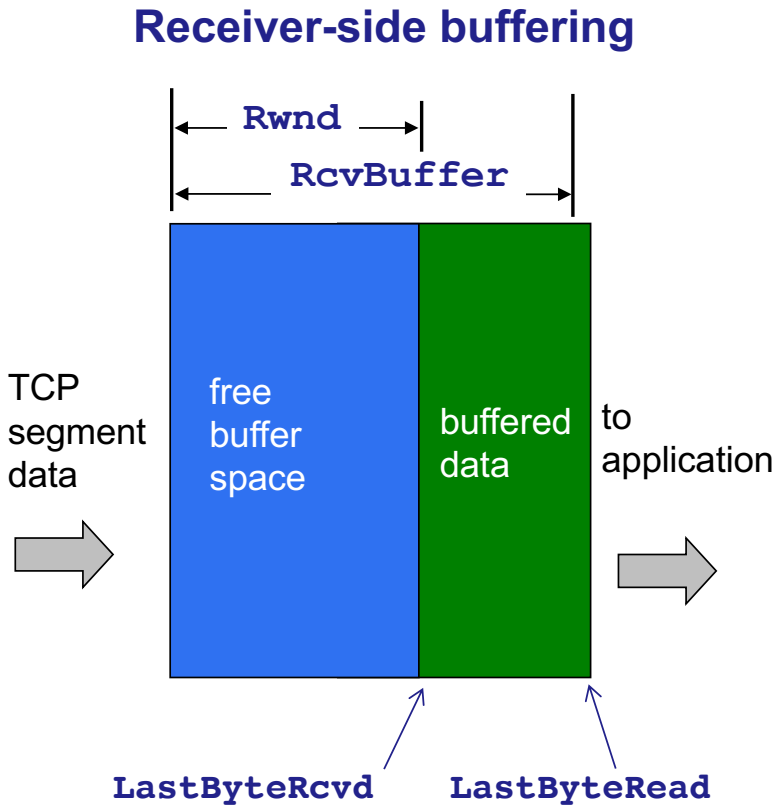
- TCP receiver has a receive buffer
- Application process may be slow at reading from buffer

Receiver protocol stack



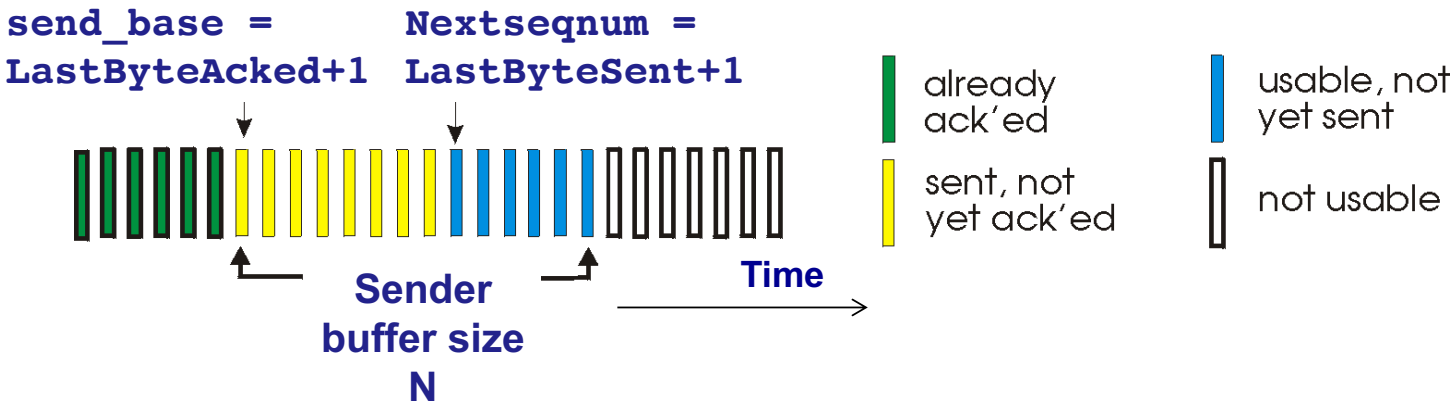
Available space in receiver buffer is communicated to sender

- **RcvBuffer** size is set via socket options
  - typical default is 4096 bytes
  - Many operating systems autoadjust **RcvBuffer**
- Spare room in buffer =
  window size announced by receiver to sender =
  Rwnd
- **Rwnd** = **RcvBuffer-**
  [LastByteRcv - LastByteRead]

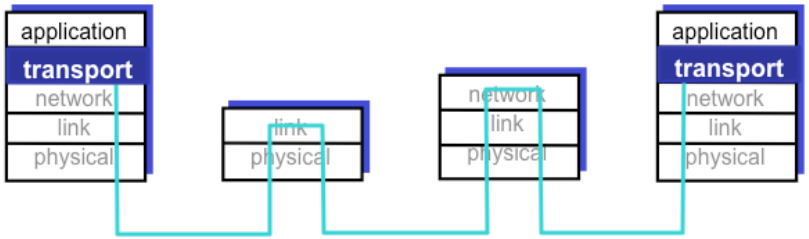


Sender limits amount of unacked ("in-flight") data to receiver's `Rwnd` value

- Guarantees receive buffer will not overflow
- Receiver announces free buffer space, **`Rwnd`**, in TCP header **`window`** field
- Sender cannot send more than receiver can receive
  - **`LastByteSent – LastByteAcked <= Rwnd`**



Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: Transmission Control Protocol

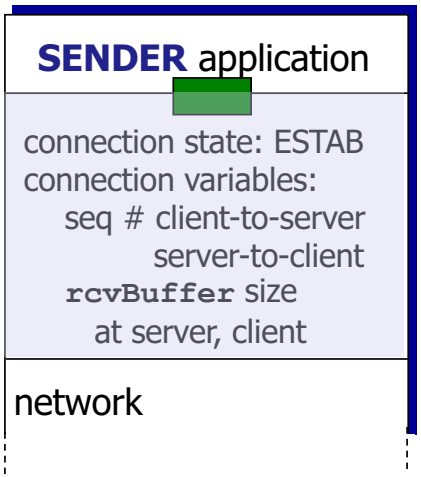
- segment structure
- reliable data transfer
- flow control
- **TCP connection management**

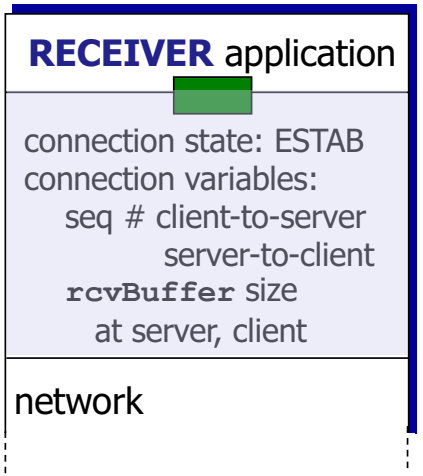
3.6 Principles of congestion control

3.7 TCP congestion control

TCP connection management

- Before exchanging data, sender/receiver “handshake”
  - Agree to establish connection
  - Agree on connection parameters



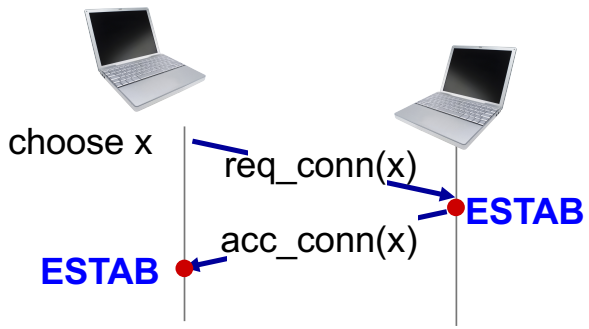



```
Socket clientSocket = new
Socket("hostname","portnumber");
```

```
Socket connectionSocket =
welcomeSocket.accept();
```

Agreeing to establish a connection

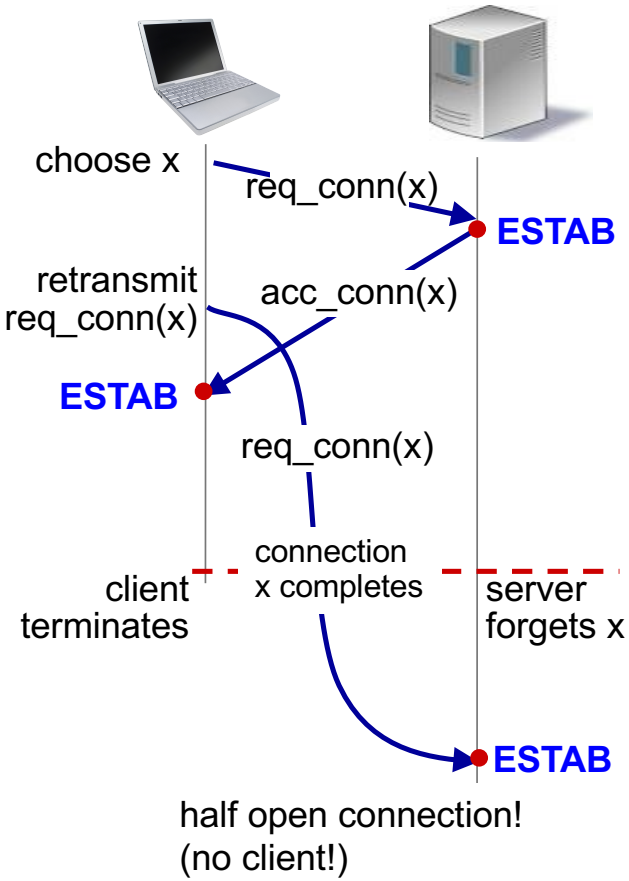
2-way handshake:

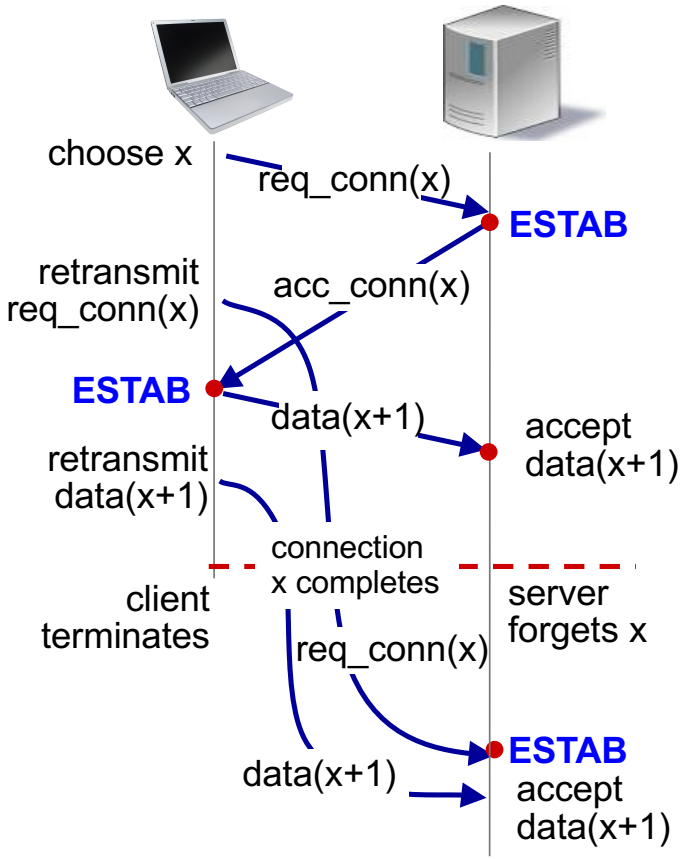


Will 2-way handshake always work in network with

- variable delays
- retransmitted messages due to message loss
- message reordering
- can't "see" other side?

2-way handshake failure scenarios



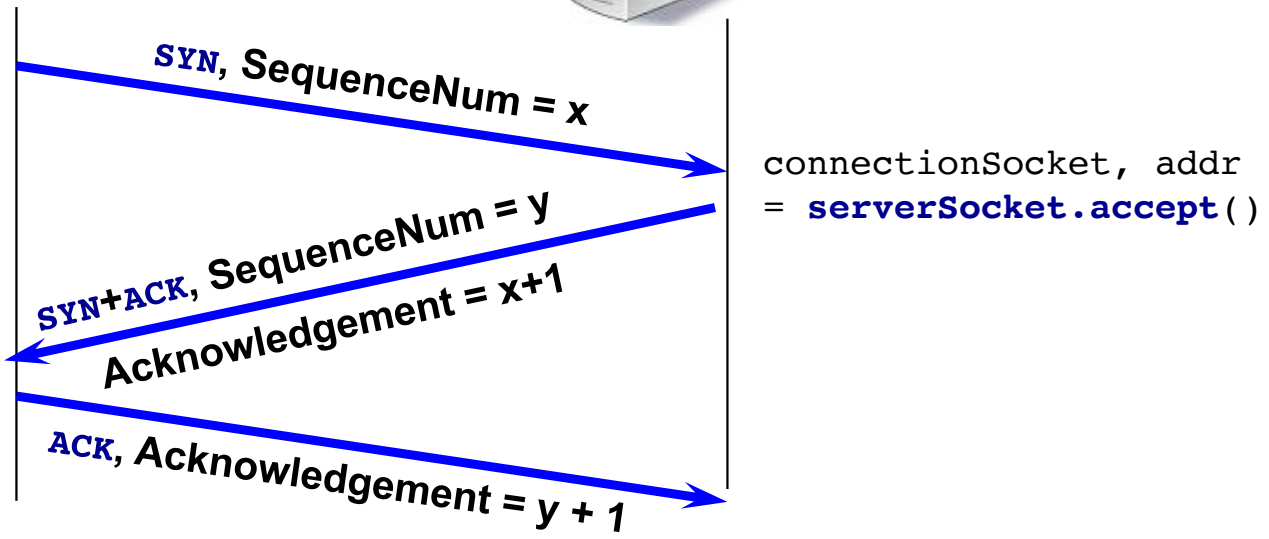


TCP connection set-up through a three-way handshake

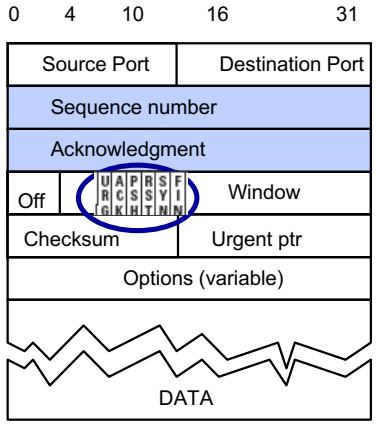
Active participant (**client**)

Passive participant (**server**)

```
clientSocket.connect
("hostname",
"port number");
```




```
connectionSocket, addr
= serverSocket.accept()
```



TCP connection set-up and byte-stream segmentation

Source	Destination	Protocol	Info
172.16.3.226	158.39.11.240	TCP	61797 > http [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=3 TSV=739718215 T
158.39.11.240	172.16.3.226	TCP	http > 61797 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1460 WS=0 TSV=
172.16.3.226	158.39.11.240	TCP	61797 > http [ACK] Seq=1 Ack=1 Win=524280 Len=0 TSV=739718216 TSER=0
172.16.3.226	158.39.11.240	HTTP	GET /00_ROOT/links.htm HTTP/1.1
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
172.16.3.226	158.39.11.240	TCP	61797 > http [ACK] Seq=634 Ack=2897 Win=524280 Len=0 TSV=739718217 TSE
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
172.16.3.226	158.39.11.240	TCP	61797 > http [ACK] Seq=634 Ack=5793 Win=522728 Len=0 TSV=739718218 TSE
172.16.3.226	158.39.11.240	TCP	61797 > http [ACK] Seq=634 Ack=7241 Win=524280 Len=0 TSV=739718218 TSE
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
158.39.11.240	172.16.3.226	TCP	[TCP segment of a reassembled PDU]
172.16.3.226	158.39.11.240	TCP	61797 > http [ACK] Seq=634 Ack=8689 Win=524176 Len=0 TSV=739718218 TSE

Connection-oriented TCP demultiplexes connections

- TCP socket identified by 4-tuple
  - **IP source address**
  - **IP destination address**
  - **TCP source port number**
  - **TCP destination port number**
- Receiver uses all four values to direct segment to appropriate socket

- Server host may support many simultaneous TCP sockets/connections
  - each socket identified by its own 4-tuple
- Web servers **create different sockets for each connecting client**
  - non-persistent HTTP will have different socket for each request

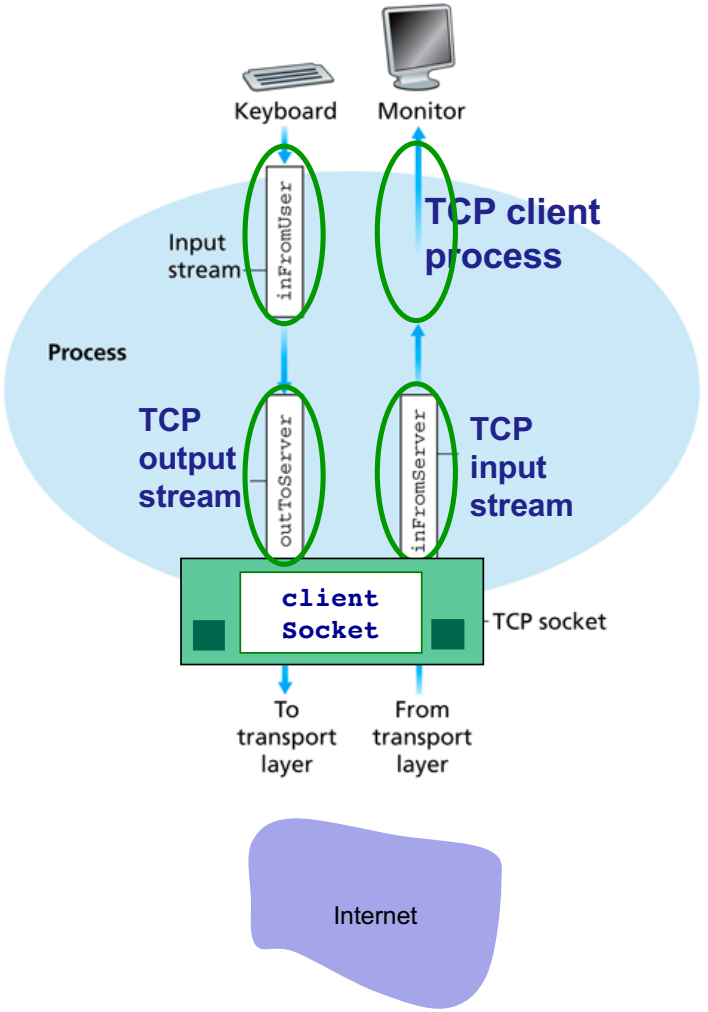
E.g. IP datagram for HTTP request:

```
Ethernet II, Src: Apple_ee:45:5a (c8:bc:c8:ee:45:5a), Dst: ZyxelCom_11:88:e3 (cc:5d:4e:11:88:e3)
Internet Protocol Version 4, Src: 10.0.0.37, Dst: 8.254.164.254
Transmission Control Protocol, Src Port: 64491 (64491), Dst Port: http (80), Seq: 1, Ack: 1, Len: 431
Hypertext Transfer Protocol
```

NTNU

CLIENT TCP

- As stream is a sequence of characters that flows into or out of a process
- An **input stream** is attached to some input source for the process, e.g. keyboard or socket
- An **output stream** is attached to an output source, e.g. monitor or socket

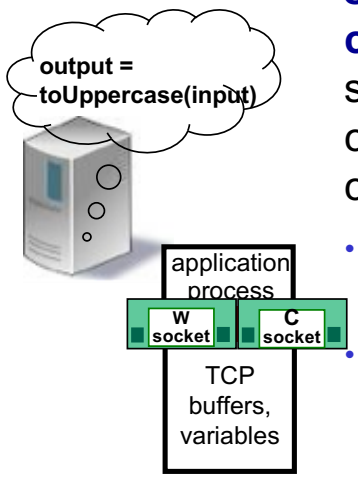


Socket programming with TCP

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

- server listens on **welcoming socket**
- **server TCP creates new connection socket** for server process to communicate with contacting client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients

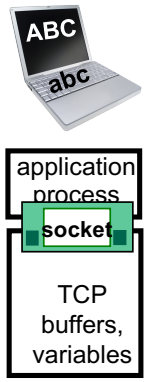


client must contact server

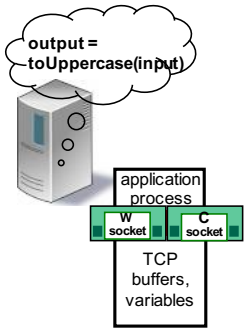
- server process must be running
- server must have created socket to welcome client's contact

client contacts server by

- creating TCP socket
- specifying IP address & port number of server process
- establishing TCP connection to server – connect



Client/server socket interaction: TCP



Server (running on `serverIP`)

Create socket, port=`X`, for incoming request:
WelcomeServerSocket = socket()

Wait for incoming connection request
connectionSocket = WelcomeServerSocket.accept()

Read request from **connectionSocket**

Write reply to **connectionSocket**

Close **connectionSocket**

TCP connection setup

TCP data

TCP data

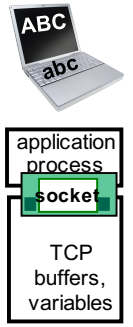
client

Create socket, connect to `serverIP`, port=`X`
clientSocket = socket()

Send request using **clientSocket**

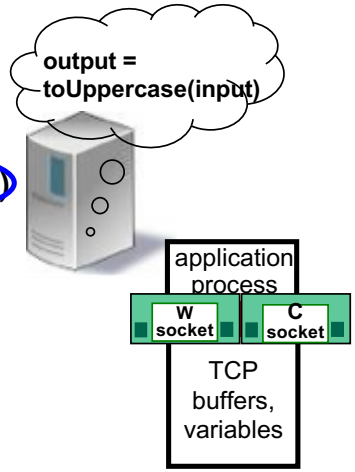
Read reply from **clientSocket**

Close **clientSocket**



Example: Python TCP server

% Ch 2
Socket API



create TCP **welcoming** socket

server begins listening for incoming TCP requests

loop forever

server waits on accept() for incoming requests, new **socket created** on return

read bytes from socket (but not address as in UDP)

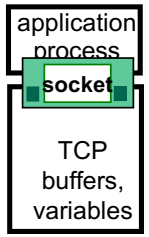
close connection to this client (but *not* welcoming socket)

```
from socket import *
serverPort = 1030
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))

serverSocket.listen(1)

print 'The server is ready to receive'

while 1:
    connectionSocket, addr = serverSocket.accept()

    sentence = connectionSocket.recv(1024)
    capitalizedSentence = sentence.upper()

    connectionSocket.send(capitalizedSentence)

    connectionSocket.close()
```

http://docs.python.org/2/library/socket.html

Example: Python TCP client

% Ch 2
Socket API



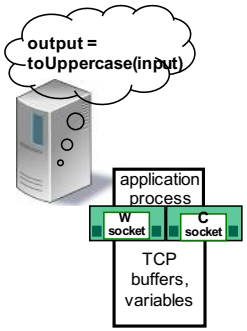
create TCP socket (SOCK_STREAM) for server, remote port 1030

No need to attach server name, port

```
from socket import *
serverName = 'servername'
serverPort = 1030
clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName,serverPort))

sentence = raw_input('Input lowercase sentence:')

clientSocket.send(sentence)

modifiedSentence = clientSocket.recv(1024)

print 'From Server:', modifiedSentence

clientSocket.close()
```

http://docs.python.org/2/library/socket.html

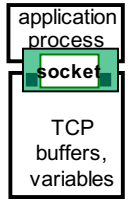
Socket methods to initialize connection



Server (running on 10.0.0.76)

```
serverPort = 1030
serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('',serverPort))

serverSocket.listen(1)
```

Client (on 10.0.0.37)




```
serverName = '10.0.0.76'
serverPort = 1030
clientSocket = socket(AF_INET, SOCK_STREAM)

clientSocket.connect((serverName,serverPort))
```

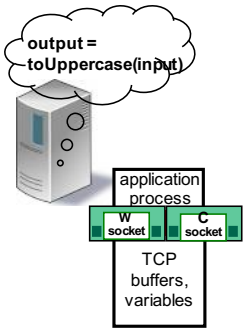
TCP connection setup

```
connectionSocket, addr = serverSocket.accept()
```

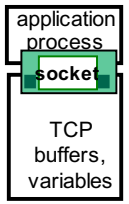
TCP connection setup

```
34 9.981273000  10.0.0.37  10.0.0.76   TCP    78 51211 > iad1 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=16 TSval=57082033 TSecr=0 SACK_PERM=1
35 9.984233000  10.0.0.76  10.0.0.37   TCP    78 iad1 > 51211 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=16 TSval=622449672 TSecr=57082033 SACK_PERM=1
36 9.984495000  10.0.0.37  10.0.0.76   TCP    66 51211 > iad1 [ACK] Seq=1 Ack=1 Win=131760 Len=0 TSval=57082062 TSecr=622449672
```

Socket methods to send and receive data



Client (on 10.0.0.37)




```
Client: Skriv din melding
Kan du oversette: En jeger gikk i skogen
Klient: Avventer svar fra server
```

```
sentence = raw_input('Input lowercase sentence:')
clientSocket.send(sentence)
```

Server (running on 10.0.0.76)

```
Melding fra klient: Kan du oversette: En jeger gikk i skogen
Svar sendt til klient: KAN DU OVERSETTE: EN JEGER GIKK I SKOGEN
```

```
sentence = connectionSocket.recv(1024)

capitalizedSentence = sentence.upper()

connectionSocket.send(capitalizedSentence)
```

TCP data

```
Svar mottatt fra server: KAN DU OVERSETTE: EN JEGER GIKK I SKOGEN
```

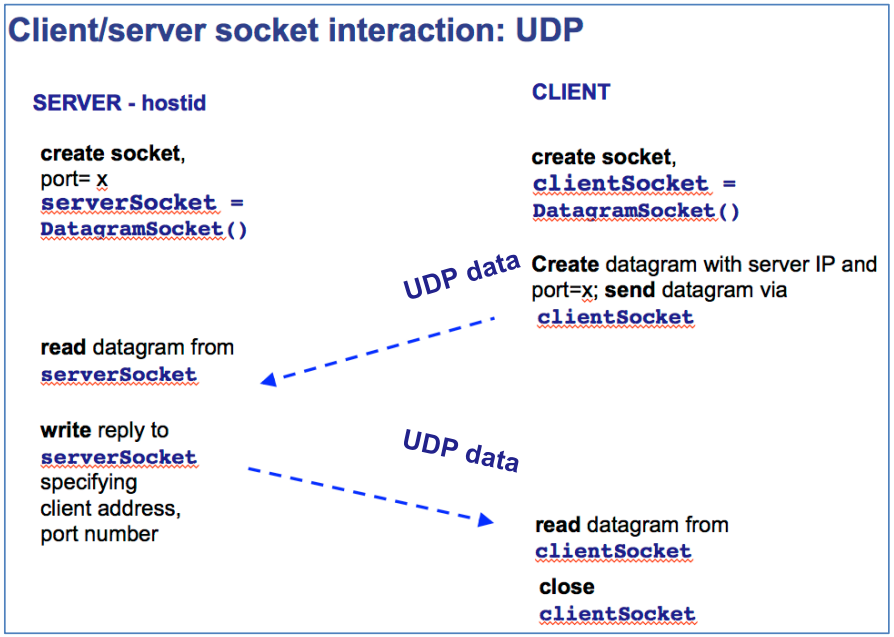
	From	To	Protocol	Length	PortS	PortD (iad1 = 1030)	
42	22.126653000	10.0.0.37	10.0.0.76	TCP	67	51211 > iad1	[PSH, ACK] Seq=1 Ack=1 Win=131760 Len=1 TSval=57094164 TSecr=622449674
43	22.273157000	10.0.0.76	10.0.0.37	TCP	66	iad1 > 51211	[ACK] Seq=1 Ack=2 Win=131760 Len=0 TSval=622461927 TSecr=57094164
44	22.273391000	10.0.0.37	10.0.0.76	TCP	106	51211 > iad1	[PSH, ACK] Seq=2 Ack=1 Win=131760 Len=40 TSval=57094310 TSecr=622461927
45	22.275080000	10.0.0.76	10.0.0.37	TCP	66	iad1 > 51211	[ACK] Seq=1 Ack=42 Win=131712 Len=0 TSval=622461929 TSecr=57094310
46	22.275759000	10.0.0.76	10.0.0.37	TCP	67	iad1 > 51211	[PSH, ACK] Seq=1 Ack=42 Win=131712 Len=1 TSval=622461929 TSecr=57094310
47	22.276009000	10.0.0.37	10.0.0.76	TCP	66	51211 > iad1	[ACK] Seq=42 Ack=2 Win=131760 Len=0 TSval=57094311 TSecr=622461929
48	22.277655000	10.0.0.76	10.0.0.37	TCP	106	iad1 > 51211	[PSH, ACK] Seq=2 Ack=42 Win=131712 Len=40 TSval=622461931 TSecr=57094311
49	22.277710000	10.0.0.37	10.0.0.76	TCP	66	51211 > iad1	[ACK] Seq=42 Ack=42 Win=131712 Len=0 TSval=57094313 TSecr=622461931

NTNU

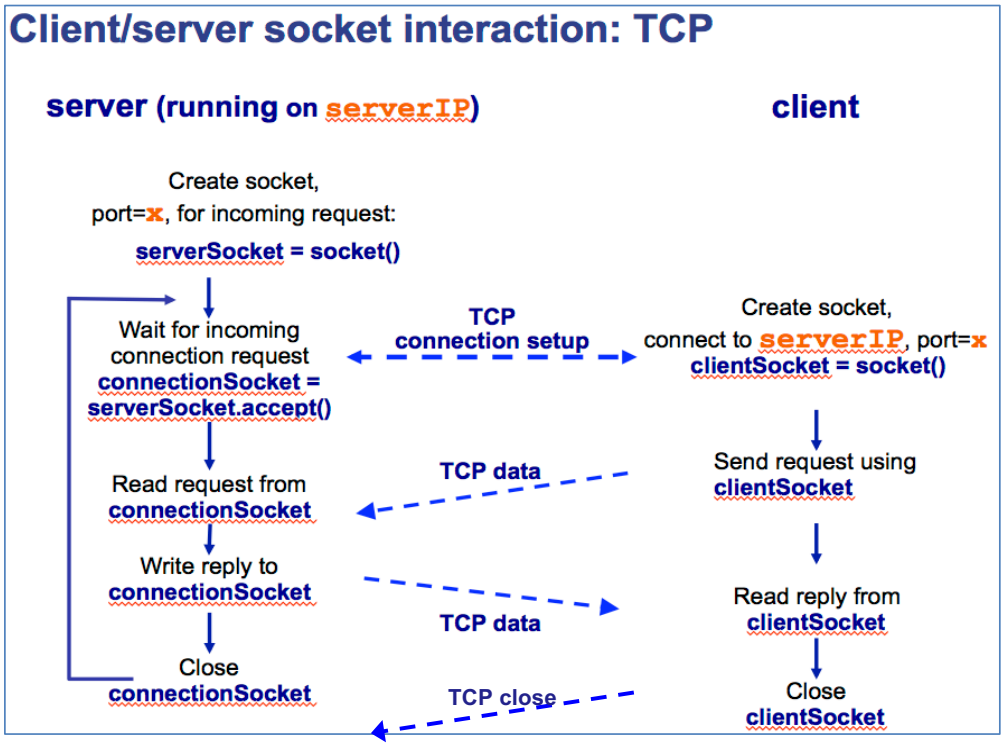
Summary

Socket API and UDP vs TCP transmission

Client/server socket interaction: UDP

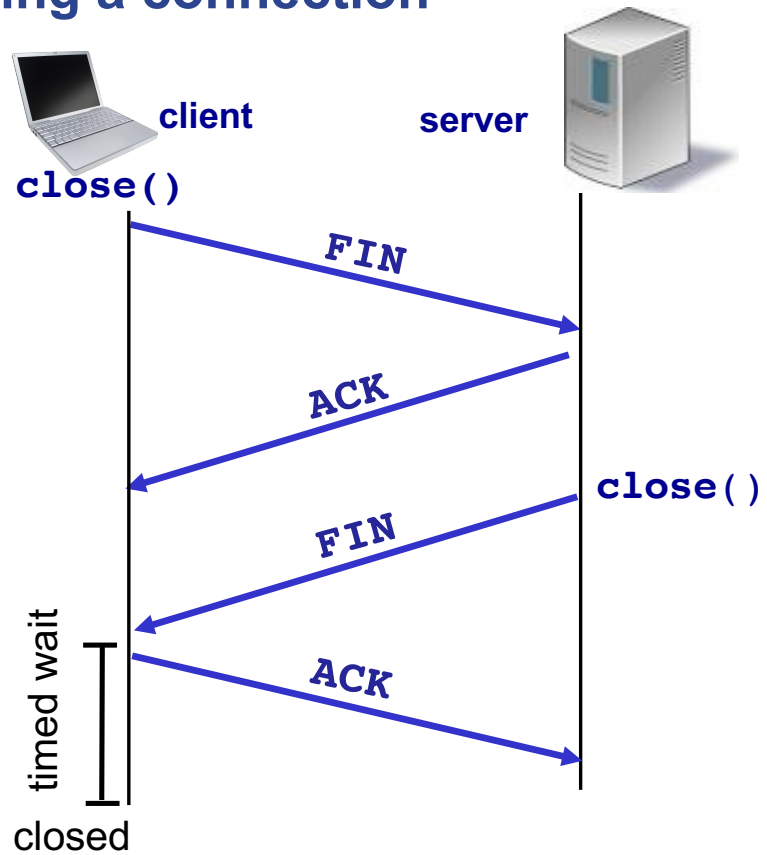


Client/server socket interaction: TCP



TCP connection management – closing a connection

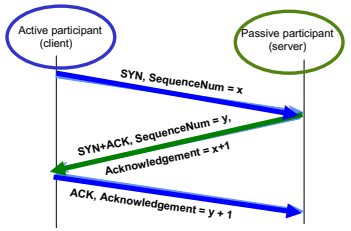
- Client closes socket
  `clientSocket.close();`
  FIN
- Server returns **ACK**
- Server closes connection, **FIN**
- Client sends **ACK** enters TIME_WAIT state to be able to resend **ACK** to received **FINs**
- Connection closed at server when **ACK** received

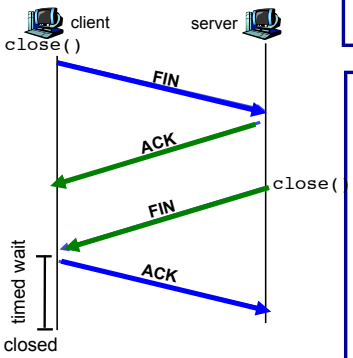


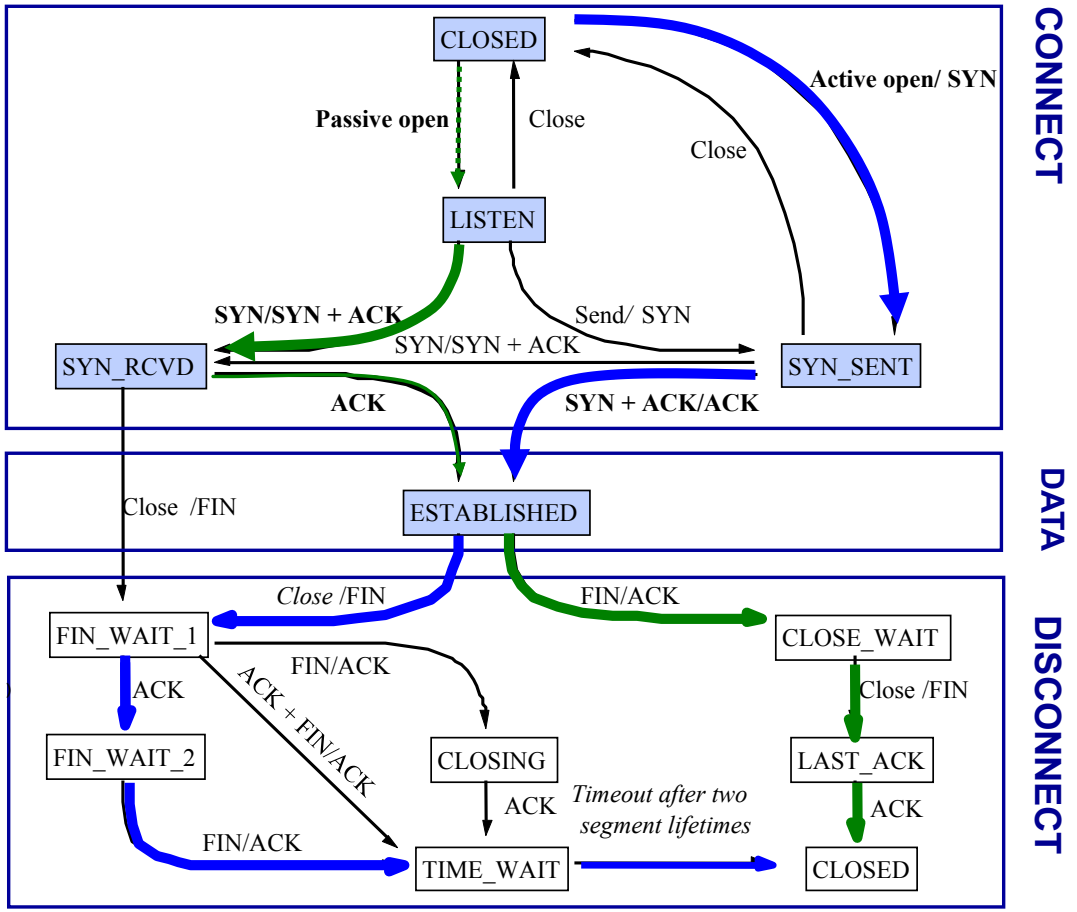
Time	Source	Destination	Protocol	Length	Info
122.260653	10.0.0.103	193.213.115.9	TCP	54	49441 → pop3s [FIN, ACK] Seq=795 Ack=1824 Win=65535 Len=0
122.266248	193.213.115.9	10.0.0.103	TCP	54	pop3s → 49441 [ACK] Seq=1824 Ack=796 Win=49640 Len=0
122.266322	10.0.0.103	193.213.115.9	TCP	54	[TCP Dup ACK 1110#1] 49441 → pop3s [ACK] Seq=796 Ack=1824 Win=65535 Len=0
122.266409	193.213.115.9	10.0.0.103	TCP	54	pop3s → 49441 [FIN, ACK] Seq=1824 Ack=796 Win=49640 Len=0
122.266440	10.0.0.103	193.213.115.9	TCP	54	49441 → pop3s [ACK] Seq=796 Ack=1825 Win=65535 Len=0

TCP: State diagram

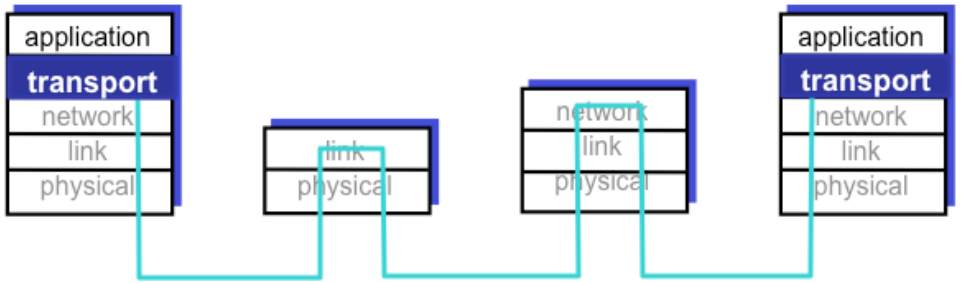
"Peer-to-peer" and service interface







Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP (Transmission Control Protocol)

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

Principles of congestion control

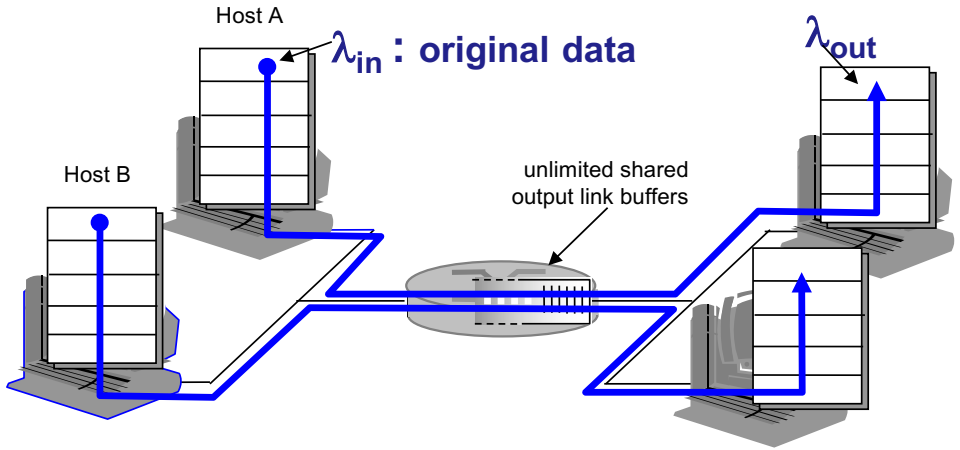
- **Congestion**: “too many sources sending too much data too fast for network to handle”
- Different from flow control!
- Manifestations
  - **lost packets** (buffer overflow at routers)
  - **long delays** (queuing in router buffers)

Costs of congestion

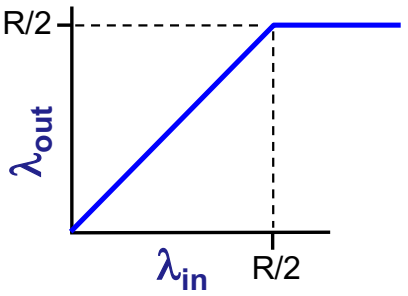
- More work (retransmission) for given “goodput”
- Unneeded retransmissions: link carries multiple copies of packet - decreasing goodput

Large queuing delay when packet rate nears link capacity

- Two senders, two receivers share link capacity R
- One router, **unlimited buffer**
- **No retransmission**

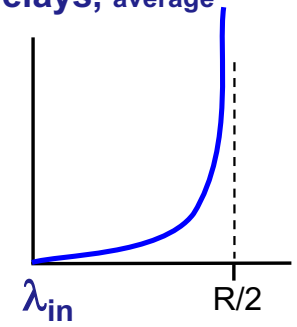


Throughput, per connection



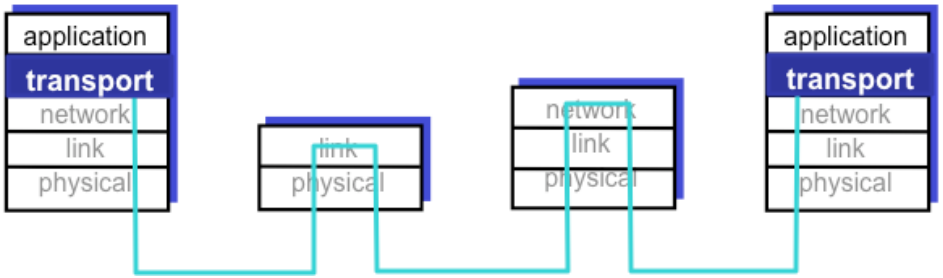
maximum per-connection throughput: R/2

Delays, average



large delays as arrival rate, λ_{in} approaches capacity

Transport layer



3.1 Transport-layer services

3.2 Multiplexing and demultiplexing

3.3 Connectionless transport: UDP (User Datagram Protocol)

3.4 Principles of reliable data transfer

3.5 Connection-oriented transport: TCP (Transmission Control Protocol)

- segment structure
- reliable data transfer
- flow control
- connection management

3.6 Principles of congestion control

3.7 TCP congestion control

TCP congestion control: loss event = congestion

- **Loss** event = retransmission **timeout** or **3 duplicate acks**

- TCP sender reduces sending rate after loss event
- A variable **`CongWin`** is used to control the number of sent, and unacknowledged segments

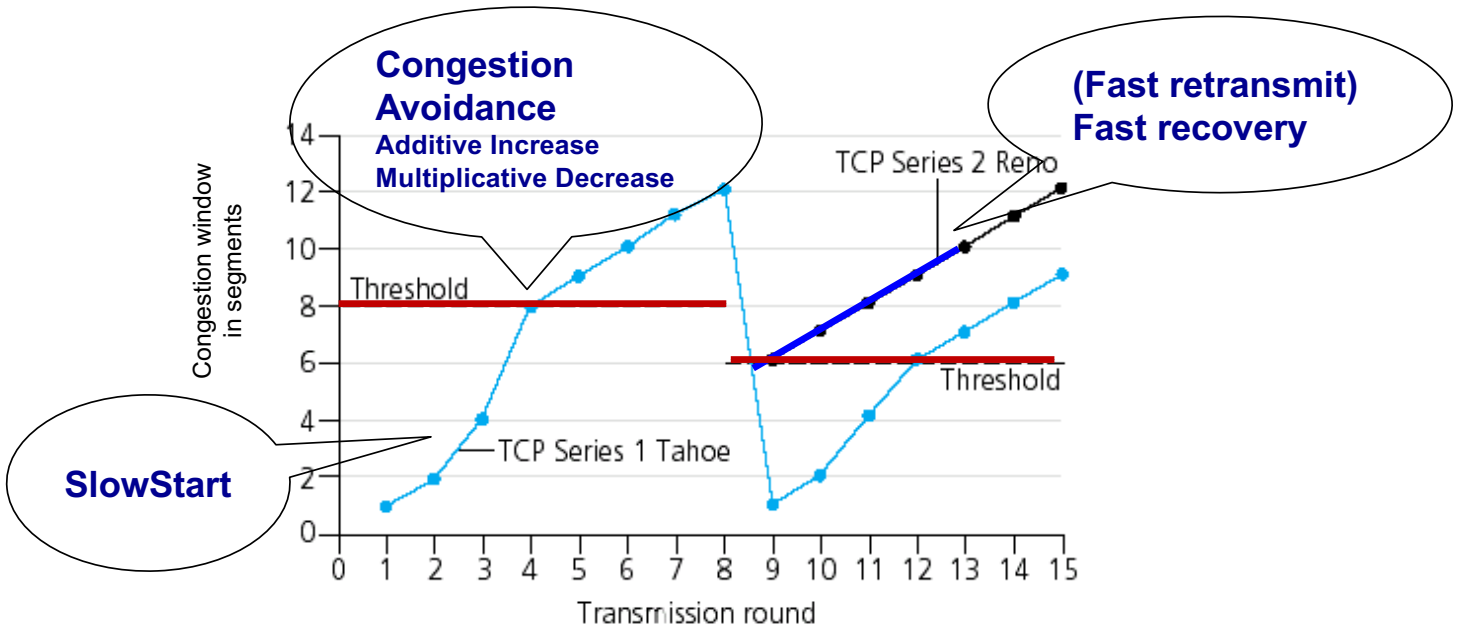
$$\text{rate} = \frac{\texttt{CongWin}}{\text{RTT}} \text{ bytes/s}$$

- **`CongWin`** is dynamic, and the function of perceived network congestion as seen from the sender

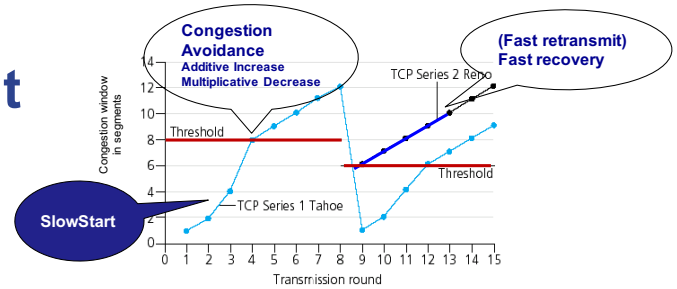
- Three mechanisms
  1. **Slow start**
  2. **Congestion avoidance: Additive Increase Multiplicative Decrease**
  3. **Fast retransmit, fast recovery – conservative after timeout events**

The three mechanisms of TCP congestion control

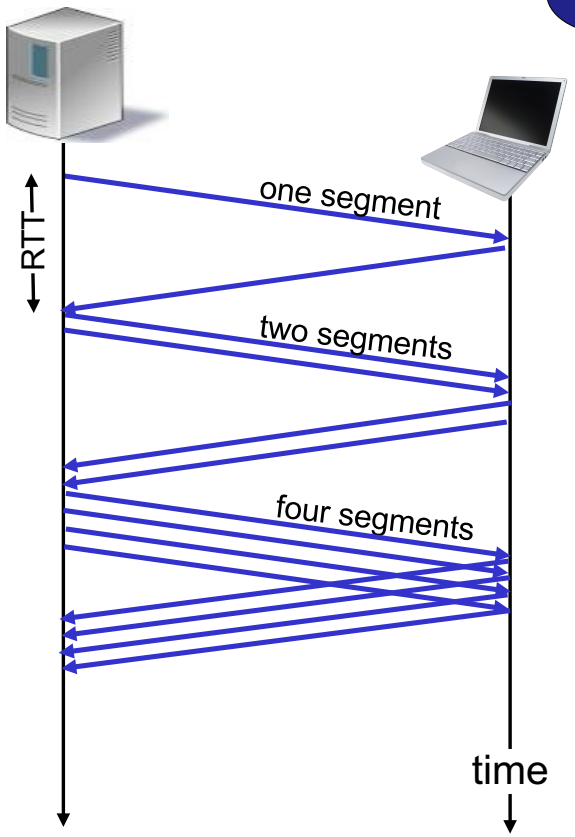
- When **`CongWin`** gets to 1/2 of its value before timeout: change from exponential to linear increase
- Implementation: variable **`Threshold`**
- At loss event, **`Threshold`** is set to 1/2 of **`CongWin`** just before loss event



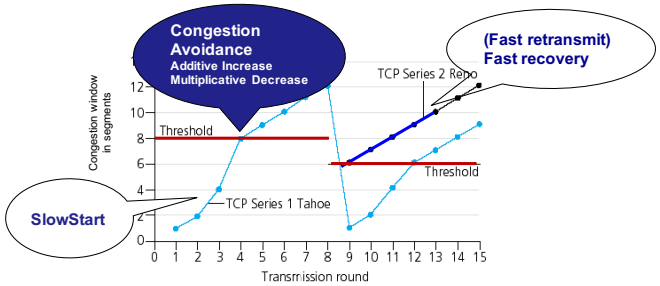
TCP slow start: after connection set-up and timeout increase sending rate exponentially



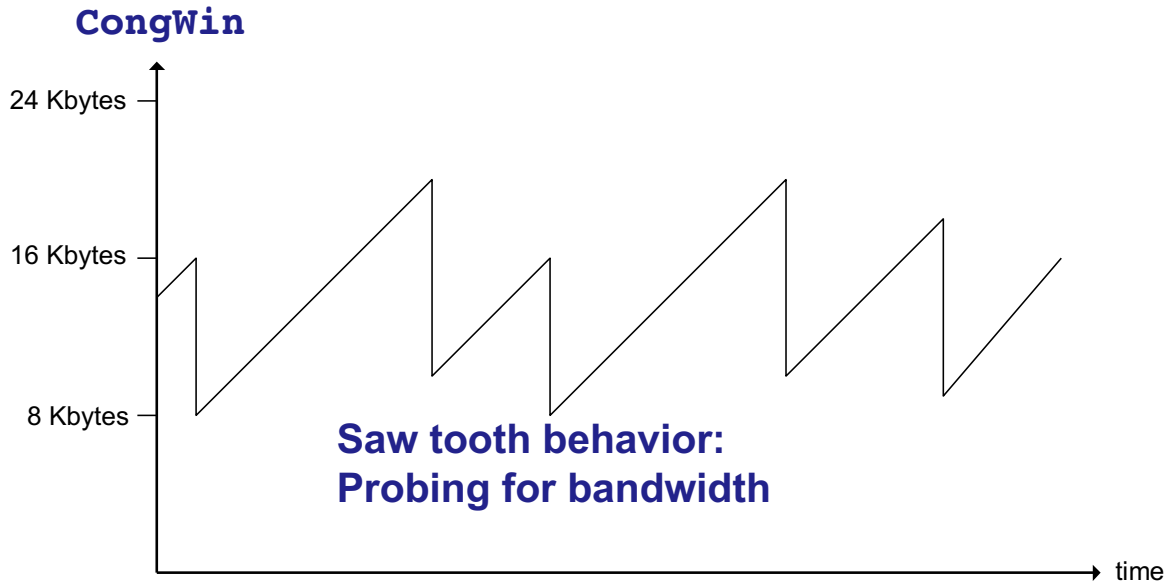
- At start of connection **`CongWin`** = 1 MSS
  - If MSS = 500 bytes & RTT = 200 msec initial rate = MSS/RTT =20 kbps
- Available bandwidth may be >> MSS/RTT
  - desirable to quickly ramp up to respectable rate
  - double **`CongWin`** every RTT
  - done by incrementing **`CongWin`** for every ACK received



TCP congestion avoidance: additive increase, multiplicative decrease

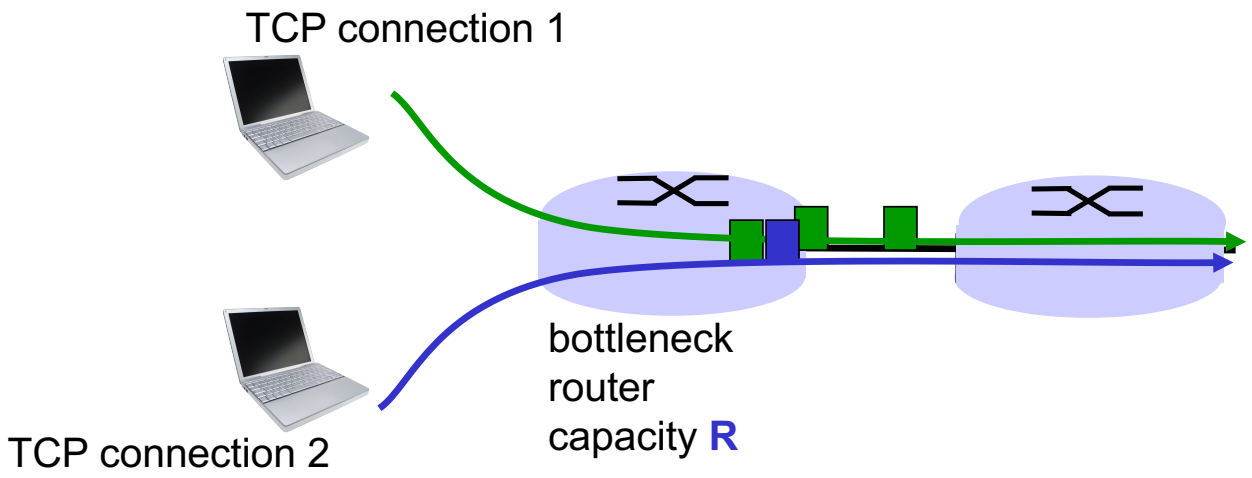


- Increase transmission rate, probing for usable bandwidth, until loss occurs
  - **additive increase:** increase `CongWin` by 1 MSS (maximum segment size) every RTT until loss detected
  - **multiplicative decrease:** cut `CongWin` in half after loss



Fairness: each connection gets an equal share of available bandwidth

Fairness goal: if K TCP sessions share same bottleneck link of bandwidth R, each should have average rate of R/K

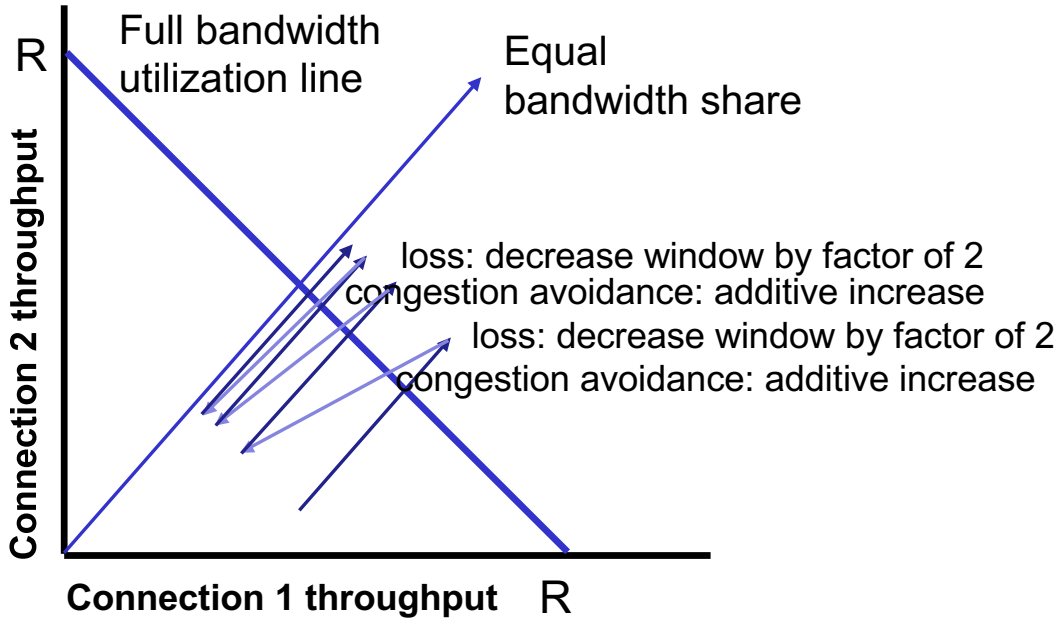


Why TCP fair – assuming AIMD

Additive Increase Multiplicative Decrease

Two competing sessions:

- Additive increase gives slope of 1, as throughput increases
- Multiplicative decrease decreases throughput proportionally



But fairness is not always the case

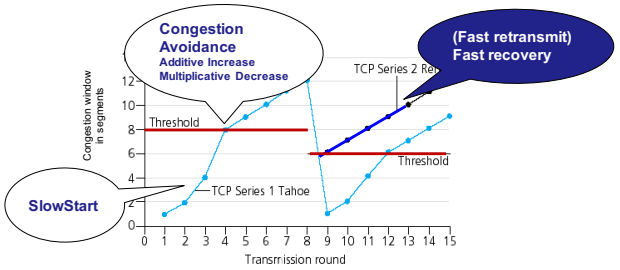
Fairness and UDP

- Multimedia applications often do not use TCP
  - do not want rate throttled by congestion control
- Instead use UDP
  - pump audio/video at constant rate, tolerate packet loss
- Research area: **TCP friendly congestion control**

Fairness and parallel TCP connections

- Nothing prevents applications from opening **parallel connections** between 2 hosts
  - web browsers do this
- Example: link of rate R supporting 9 connections;
  - new app asks for 1 TCP, gets rate R/10
  - new app asks for 11 TCPs, gets > R/2 !

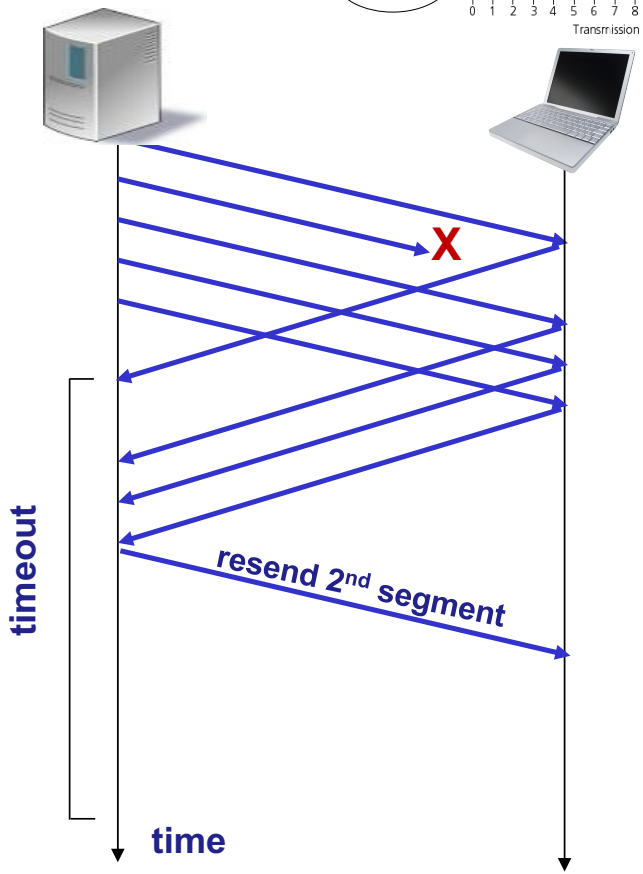
TCP fast recovery:
3 duplicate ACKs is a packet loss = congestion



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 (+3MSS)
  - **CongWin** then grows linearly
- After timeout event
  - **CongWin** is set to 1 MSS;
  - window then grows exponentially
  - to a threshold, then grows linearly



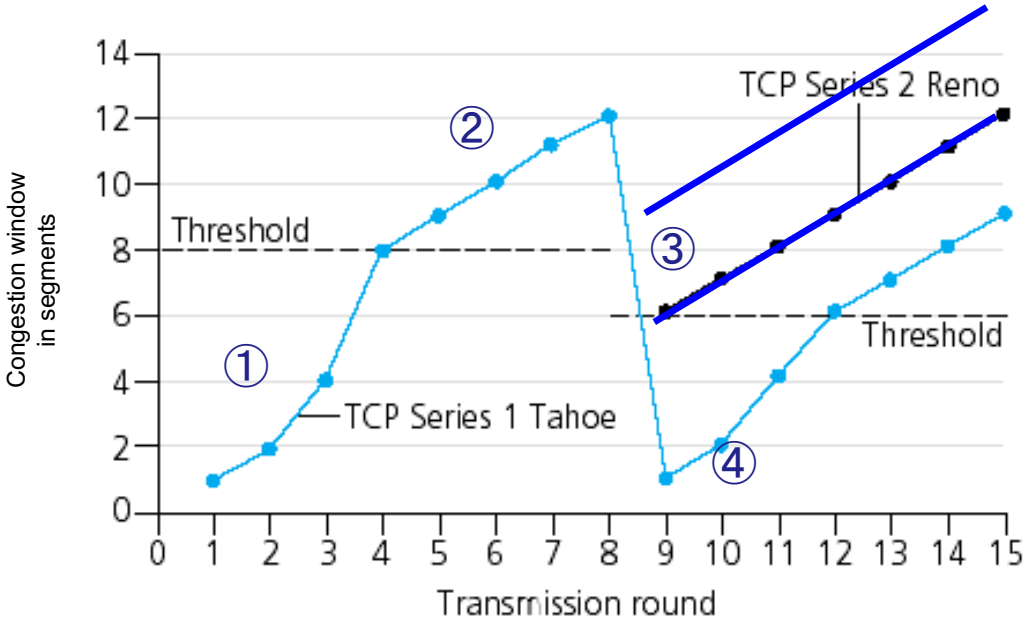
TCP congestion control - summary

① When `CongWin` is below `Threshold`, sender in **slow-start** phase, window grows exponentially.

② When `CongWin` is above `Threshold`, sender is in **congestion-avoidance** phase, window grows linearly.

③ When a **triple duplicate ACK** occurs, `Threshold` set to `CongWin`/2 and `CongWin` set to `Threshold (+3MSS)`.

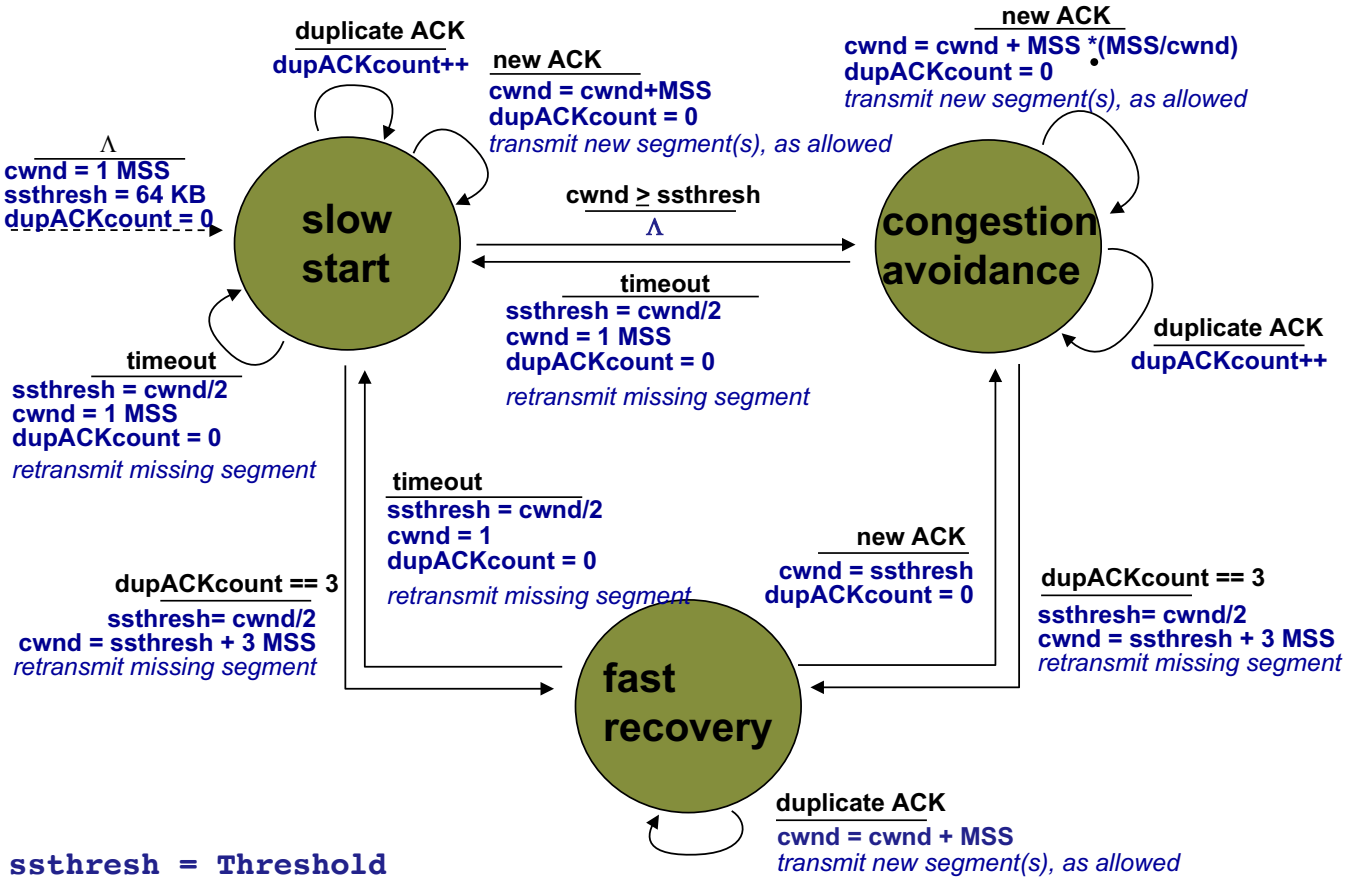
④ When timeout occurs, `Threshold` set to `CongWin`/2 and `CongWin` is set to 1 MSS.



TCP sender congestion control – states

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
	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 (Congestion Avoidance)	**Loss event detected by triple duplicate ACK**	`Threshold = CongWin/2`, `CongWin = Threshold+3MSS` 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

TCP sender congestion control – FSM (Finite state machine)



á la carte

TCP congestion control RFCs (requests for comments)

- V. Jacobson, "Congestion Avoidance and Control," Computer Communication Review, vol. 18, no. 4, pp. 314-329, Aug. 1988
- RFC2211 Requirements for Internet Hosts -- Communication Layers, October 1989
- V. Jacobson, "Modified TCP Congestion Avoidance Algorithm," end2end-interest mailing list, April 30, 1990.
- RFC 2001 TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997
- RFC 2414 Increasing TCP's Initial Window, September 1998
- TCP 2581 TCP Congestion Control, April 1999
- RFC 2582 The NewReno Modification to TCP's Fast Recovery Algorithm, April 1999
- RFC 3390 Increasing TCP's Initial Window, October 2002
- TFC 3782 The NewReno Modification to TCP's Fast Recovery Algorithm, April 2004
- RFC 5681 TCP Congestion Control (Obsoletes 2581), September 2009
- RFC 6582 The NewReno Modification to TCP's Fast Recovery Algorithm, April 2012

Source: http://www.rfc-editor.org/rfc-index.html

NTNU

Summary

Transport layer

Principles behind transport layer services:

- multiplexing, demultiplexing
- reliable data transfer
- flow control
- congestion control

Port numbers in transport protocol header

checksum, ACKs, timeout, sequence numbers, window, retransmission: go-back-N, selective

Receiver announces available buffer, Rwnd

Cwnd variable

- Instantiation and implementation in the Internet
  - UDP
  - TCP

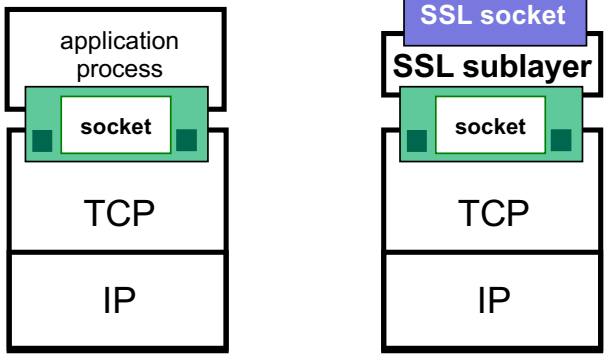
Connectionless and unreliable

Connection oriented, reliable byte stream

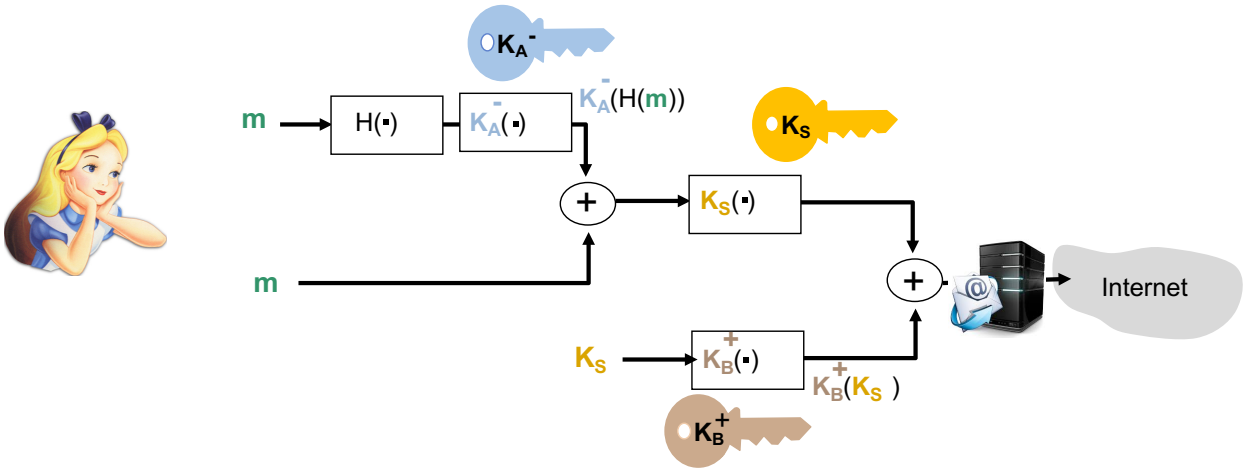
Securing TCP connections
Secure Sockets Layer (SSL)

- Widely deployed security protocol
  - supported by almost all browsers, web servers
  - **https**
  - billions $/year over **SSL**
- Variation - **TLS**: transport layer security [RFC 2246]
- SSL provides application programming interface (API) to applications
  - Available to all TCP applications
  - Secure socket interface

- Original goal: Web e-commerce transactions
- SSL enhances TCP with
  - Confidentiality - encryption (eg. credit-card numbers)
  - Integrity
  - Authentication Web-server/optional client



Secure a TCP connection not a single e-mail message

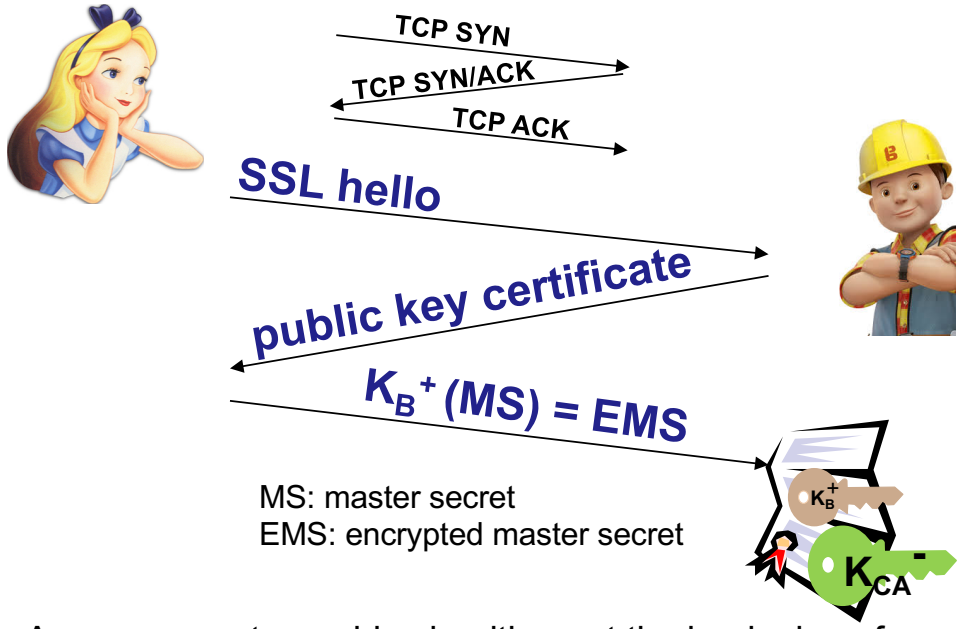


Authenticated (integrity) and confidential (encrypted) email

- Want to send byte streams & interactive data
- Want a set of secret keys for entire connection
- Want certificate exchange as part of protocol: handshake phase

SSL: securing the TCP data stream

- **Handshake**: Alice and Bob use their certificates, private keys to authenticate each other and exchange shared secret
- **Key derivation**: Alice and Bob use shared secret to derive set of keys
- **Data transfer**: data to be transferred is broken up into series of records
- **Connection closure**: special messages to securely close connection



- Agree on cryptographic algorithms at the beginning of the SSL session,

Real SSL: handshake starts with server authentication

- Negotiation: agree on crypto algorithms
  - client sends list of algorithms it supports, along with client nonce
  - server chooses algorithms from list; sends back: choice + certificate + server nonce
  - client verifies certificate, extracts server's public key, generates pre_master_secret, encrypts with server's public key, sends to server
- Establish keys: client and server independently compute encryption and MAC keys from pre_master_secret and nonces
- Client authentication (optional)
- Client and server independently compute encryption and MAC keys from pre_master_secret and nonces
  - client sends a MAC of all the handshake messages
  - server sends a MAC of all the handshake messages

Common SSL symmetric ciphers
- DES – Data Encryption Standard: block
- 3DES – Triple strength: block
- RC2 – Rivest Cipher 2: block
- RC4 – Rivest Cipher 4: stream

SSL public key encryption
- RSA

Hash algorithms
- Secure Hash Algorithm (SHA-1/2)
  Message Digest 5 (MD5)

SSL – TLS Handshake: Client and Server Hello

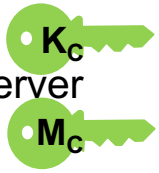
```
▶ Transmission Control Protocol, Src Port: 49442 (49442), Dst Port: pop3s (995), Seq: 1, Ack: 1, Len: 140
▼ Secure Sockets Layer
  ▼ TLSv1 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 135
    ▼ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 131
        Version: TLS 1.0 (0x0301)
      ▼ Random
          GMT Unix Time: Jan 21, 2013 08:37:06.000000000 CET
          Random Bytes: 25d4e8604e35179ed5312ef1a43fa563f5476d3fa5e2ec71...
        Session ID Length: 0
        Cipher Suites Length: 50
      ▼ Cipher Suites (25 suites)
          Cipher Suite: TLS_EMPTY_RENEGOTIATION_INFO_SCSV (0x00ff)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_RC4_128_SHA (0xc007)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc008)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
          Cipher Suite: TLS_ECDHE_RSA_WITH_RC4_128_SHA (0xc011)
          Cipher Suite: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA (0xc012)
          Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_128_CBC_SHA (0xc004)
          Cipher Suite: TLS_ECDH_ECDSA_WITH_AES_256_CBC_SHA (0xc005)
          Cipher Suite: TLS_ECDH_ECDSA_WITH_RC4_128_SHA (0xc002)
          Cipher Suite: TLS_ECDH_ECDSA_WITH_3DES_EDE_CBC_SHA (0xc003)
          Cipher Suite: TLS_ECDH_RSA_WITH_AES_128_CBC_SHA (0xc00e)
          Cipher Suite: TLS_ECDH_RSA_WITH_AES_256_CBC_SHA (0xc00f)
          Cipher Suite: TLS_ECDH_RSA_WITH_RC4_128_SHA (0xc00c)
          Cipher Suite: TLS_ECDH_RSA_WITH_3DES_EDE_CBC_SHA (0xc00d)
          Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
          Cipher Suite: TLS_RSA_WITH_RC4_128_SHA (0x0005)
          Cipher Suite: TLS_RSA_WITH_RC4_128_MD5 (0x0004)
          Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
          Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
          Cipher Suite: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA (0x0016)
```

```
Transmission Control Protocol, Src Port: pop3s (995), Dst Port: 49442 (49442), Seq: 1, Ack: 141, Len: 1276
Secure Sockets Layer
▼ TLSv1 Record Layer: Handshake Protocol: Server Hello
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 74
  ▼ Handshake Protocol: Server Hello
      Handshake Type: Server Hello (2)
      Length: 70
      Version: TLS 1.0 (0x0301)
    ▼ Random
        GMT Unix Time: Jan 21, 2013 08:37:05.000000000 CET
        Random Bytes: 2ff96fdfc039e91b234a30c74b638d4e6dca21da006a5453...
      Session ID Length: 32
      Session ID: 838b816e616ebfe6c7aa96341070664a0272fa5dfb4ffd89...
      Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
      Compression Method: null (0)
▼ TLSv1 Record Layer: Handshake Protocol: Certificate
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 1183
  ▼ Handshake Protocol: Certificate
      Handshake Type: Certificate (11)
      Length: 1179
      Certificates Length: 1176
    ▶ Certificates (1176 bytes)
▼ TLSv1 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 4
  ▼ Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

SSL: securing the TCP data stream

- **Handshake**: Alice and Bob use their certificates and private keys to authenticate each other and exchange shared secret
- **Key derivation**: Alice and Bob use shared secret to derive set of keys
- **Data transfer**: data to be transferred is broken up into series of records
- **Connection closure**: special messages to securely close connection

- Keys derived by the SSL key derivation function using a master secret (computed from a pre-master secret and some additional random data (nonce)
- 4 different keys for message authentication code (MAC) and encryption
  - Kc = encryption key for data from client to server
  - Mc = MAC key for data from client to server
  - Ks = encryption key for data from server to client
  - Ms = MAC key for data from server to client



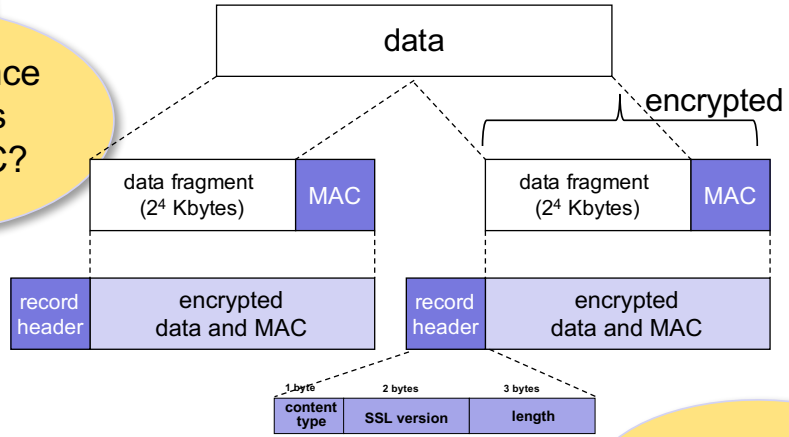
- If **cipher block chaining** a random *k*-bit string, Initialization Vector (IV) is also generated
  - IV mix some randomness into the ciphertext so that identical plaintext blocks produce different ciphertext blocks.

SSL: securing the TCP data stream

- **Handshake**: Alice and Bob use their certificates and private keys to authenticate each other and exchange shared secret
- **Key derivation**: Alice and Bob use shared secret to derive set of keys
- **Data transfer**: data to be transferred is broken up into series of records
- **Connection closure**: special messages to securely close connection

- Break stream in series of records
  - each record carries a MAC (message authentication code)
  - MAC is a hash of the data + the MAC key M_B + sequence number + type
- Receiver needs to distinguish MAC from data

Why sequence number as part of MAC?



Why a type and a length field?

SSL: securing the TCP data stream

- **Handshake**: Alice and Bob use their certificates and private keys to authenticate each other and exchange shared secret
- **Key derivation**: Alice and Bob use shared secret to derive set of keys
- **Data transfer**: data to be transferred is broken up into series of records
- **Connection closure**: special messages to securely close connection

- **Type field** indicates whether the record terminates the SSL session.
- The SSL type is sent in the clear, and authenticated at the receiver using the record's MAC.)
- MAC = MAC(MAC key, sequence||type||data)
- This avoids **truncation attack** where attacker in the middle of ongoing SSL session forges TCP connection close segment (TCP FIN)
  - one or both sides would think there was less data than actually

NTNU

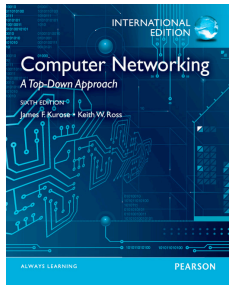
Attack challenges mitigated by SSL

Challenge	Response
Attacker can re-order records – replay of individual packets	Put sequence number into MAC: • MAC = MAC(Mx, sequence||data) • note: no sequence number field
Attacker can capture and replay records – connection replay attack	Use nonce for randomness. • Receiver sends different random nonces which causes encryption keys to be different for different connections • Replay of messages will fail integrity check
Attacker could tamper with handshake messages (e.g. delete stronger algorithm from crypto algorithm list)	Client and server sends a MAC of all the handshake messages
Attacker forges TCP connection close segment	Record types, with one type for closure • type 0 for data; type 1 for closure • MAC = MAC(Mx, sequence||type||data)

Real SSL connection

Source	Destination	Protocol	Length	Info
10.0.0.103	193.213.115.9	TCP	78	49441 → pop3s [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=16 TSval=
193.213.115.9	10.0.0.103	TCP	58	pop3s → 49441 [SYN, ACK] Seq=0 Ack=1 Win=49640 Len=0 MSS=1460
10.0.0.103	193.213.115.9	TCP	54	49441 → pop3s [ACK] Seq=1 Ack=1 Win=65535 Len=0
10.0.0.103	193.213.115.9	TLSv1	194	Client Hello
193.213.115.9	10.0.0.103	TCP	54	pop3s → 49441 [ACK] Seq=1 Ack=141 Win=49640 Len=0
193.213.115.9	10.0.0.103	TLSv1	1330	Server Hello, Certificate, Server Hello Done
10.0.0.103	193.213.115.9	TCP	54	49441 → pop3s [ACK] Seq=141 Ack=1277 Win=65535 Len=0
10.0.0.103	193.213.115.9	TLSv1	321	Client Key Exchange
193.213.115.9	10.0.0.103	TCP	54	pop3s → 49441 [ACK] Seq=1277 Ack=408 Win=49640 Len=0
10.0.0.103	193.213.115.9	TLSv1	113	Change Cipher Spec, Encrypted Handshake Message
193.213.115.9	everything henceforth is encrypted	TCP	54	pop3s → 49441 [ACK] Seq=1277 Ack=467 Win=49640 Len=0
193.213.115.9		TLSv1	113	Change Cipher Spec, Encrypted Handshake Message
10.0.0.103		TCP	54	49441 → pop3s [ACK] Seq=467 Ack=1336 Win=65535 Len=0
193.213.115.9		TLSv1	123	Application Data
10.0.0.103	193.213.115.9	TCP	54	49441 → pop3s [ACK] Seq=467 Ack=1405 Win=65535 Len=0
10.0.0.103	193.213.115.9	TLSv1	91	Application Data
193.213.115.9	10.0.0.103	TLSv1	139	Application Data
10.0.0.103	193.213.115.9	TCP	54	49441 → pop3s [ACK] Seq=504 Ack=1490 Win=65535 Len=0
10.0.0.103	193.213.115.9	TLSv1	107	Application Data
193.213.115.9	10.0.0.103	TLSv1	91	Application Data

“Network layer” fra 3. februar



4	Thursday 12:15 – 14:00 Thursday 14:15 – 15:00	Transport Layer Theory Assignment 2: *Application Layer*	R1 R1	Kjersti Assistants/ Ida/Norvald	Chapter 3 One must deliver and pass at least 5 of the 8 theory assignments.
4	Friday 09:15 – 11:00	Transport Layer (cont)	R1	Kjersti	Chapter 3
5	Thursday 12:15 – 14:00	Transport Layer (cont)	R1	Kjersti	Chapter 3 Chapter 8.6
5	Friday 09:15 – 11:00	Network Layer	R1	Kjersti	Chapter 4
6	Wednesday 18:15 – 20:00	Python Crash Course	F1	Magnus/Bank?	Install Python before the crash course.
6	Thursday 12:15 – 14:00 Thursday 14:15 – 15:00	Network Layer (cont) Theory Assignment 3: *Transport Layer* Wireshark Lab 2: *TCP (optional but highly recommended!)*	R1 R1	Kjersti Assistants/ Ida/Norvald	Chapter 4 One must deliver and pass at least 5 of the 8 theory assignments.
6	Friday 09:15 – 11:00	Network Layer (cont)	R1	Kjersti	Chapter 4 Chapter 8.7 and 8.9, 8.9.1