DIPARTIMENTO DI INFORMATICA
Via Salaria 113 - 00198, Roma

SAPIENZA UNIVERSITÀ DI ROMA

Chapter 3
Transport Layer

Reti di Elaboratori
Corso di Laurea in Informatica
Università degli Studi di Roma "La Sapienza"

Prof.ssa Chiara Petrioli

Parte di queste slide sono state prese dal materiale associato al libro *Computer Networking: A Top Down Approach* , 5th edition.

Thanks also to Antonio Capone, Politecnico di Milano, Giuseppe Bianchi and Francesco LoPresti, Un. di Roma Tor Vergata

Chapter 3: Transport Layer

Our goals:

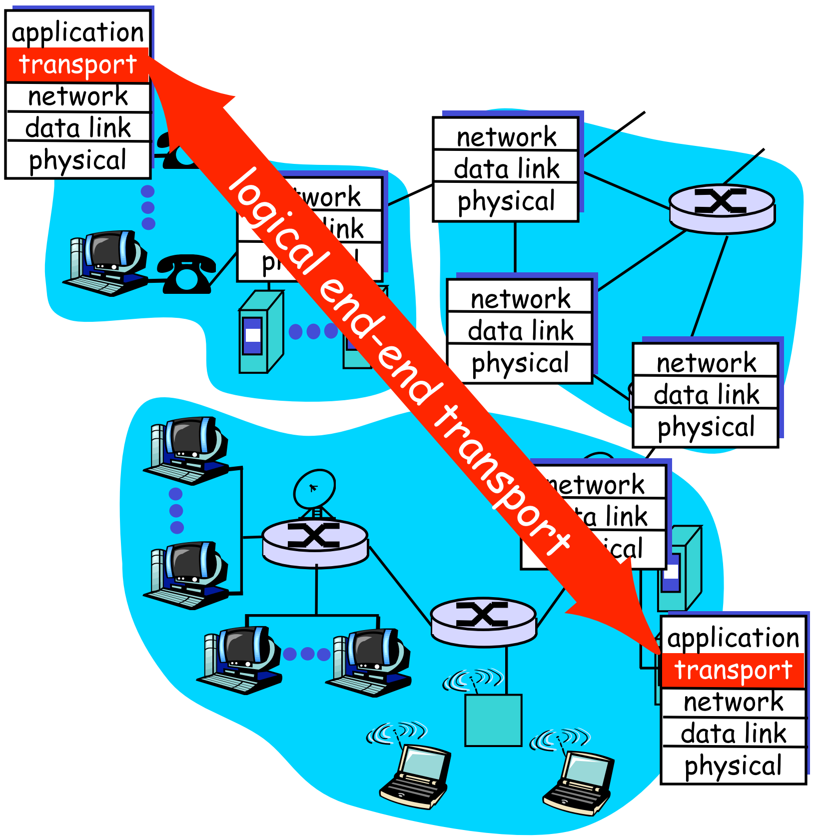
- understand principles behind transport layer services:
  - multiplexing/ demultiplexing
  - reliable data transfer
  - flow control
  - congestion control
- learn about transport layer protocols in the Internet:
  - UDP: connectionless transport
  - TCP: connection-oriented transport
  - TCP congestion control

Chapter 3 outline

- 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

Transport services and protocols

- provide *logical communication* between app processes running on different hosts
- transport protocols run in end systems
  - send side: breaks app messages into segments, passes to network layer
  - rcv side: reassembles segments into messages, passes to app layer
- more than one transport protocol available to apps
  - Internet: TCP and UDP



Transport vs. network layer

- *network layer:* logical communication between hosts
- *transport layer:* logical communication between processes
  - relies on, enhances, network layer services

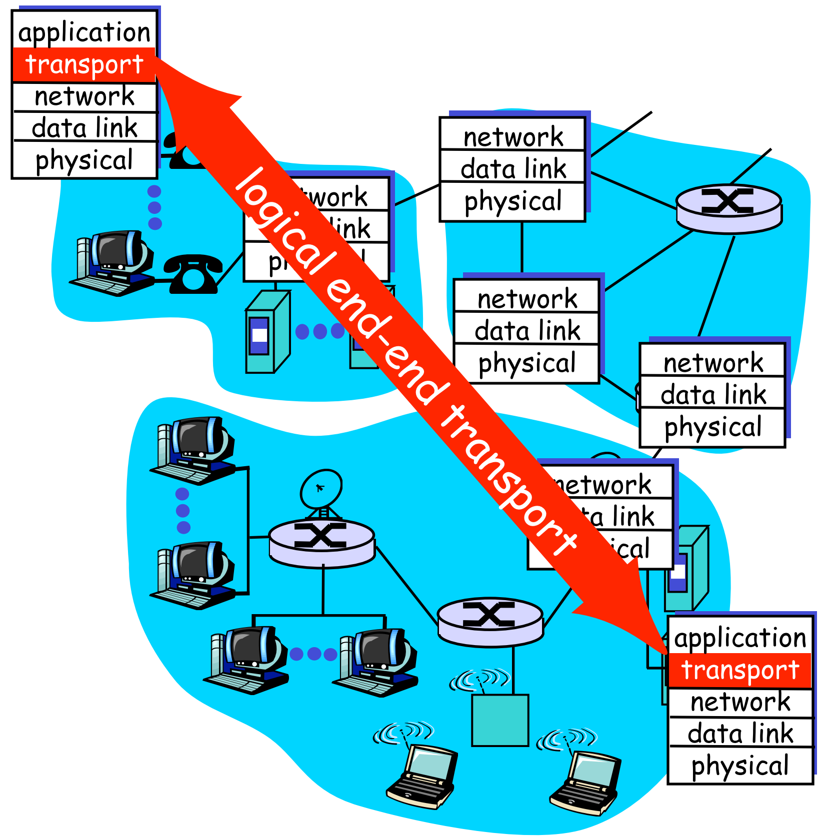
Household analogy:

12 kids sending letters to 12 kids

- processes = kids
- app messages = letters in envelopes
- hosts = houses
- transport protocol = Ann and Bill
- network-layer protocol = postal service

Internet transport-layer protocols

- reliable, in-order delivery (TCP)
  - congestion control
  - flow control
  - connection setup
- unreliable, unordered delivery: UDP
  - no-frills extension of "best-effort" IP
- services not available:
  - delay guarantees
  - bandwidth guarantees

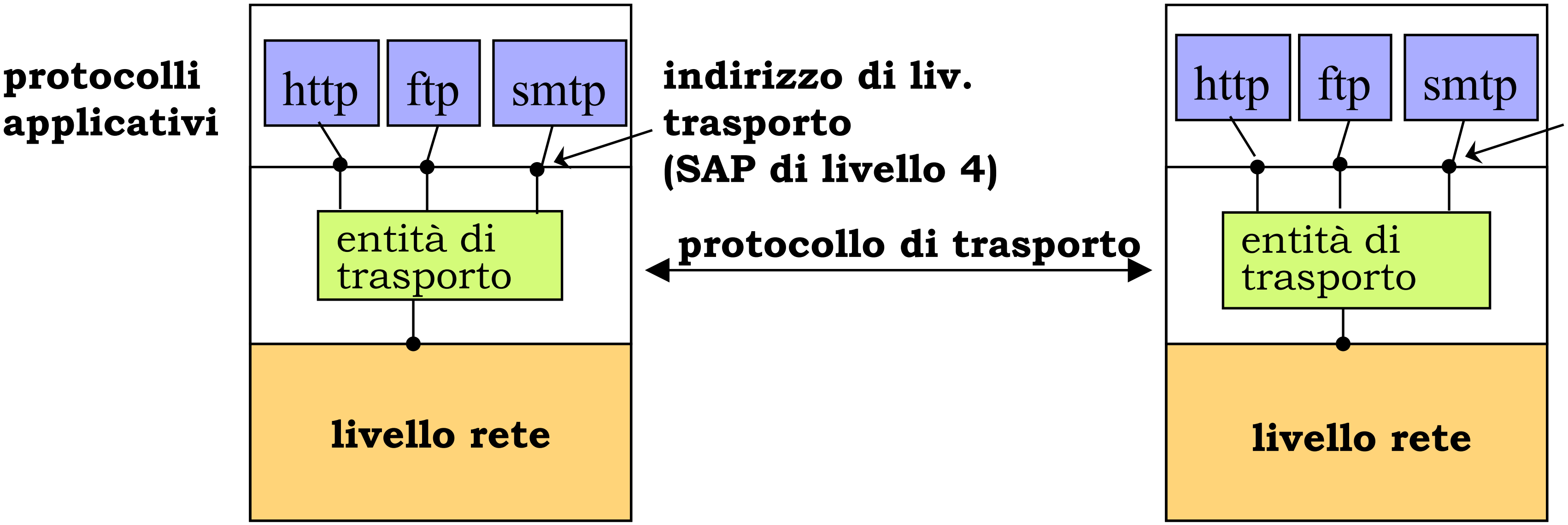


Chapter 3 outline

- 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

Servizio di trasporto

- Più applicazioni possono essere attive su un end system
  - il livello di trasporto svolge funzioni di multiplexing/demultiplexing
  - ciascun collegamento logico tra applicazioni è indirizzato dal livello di trasporto



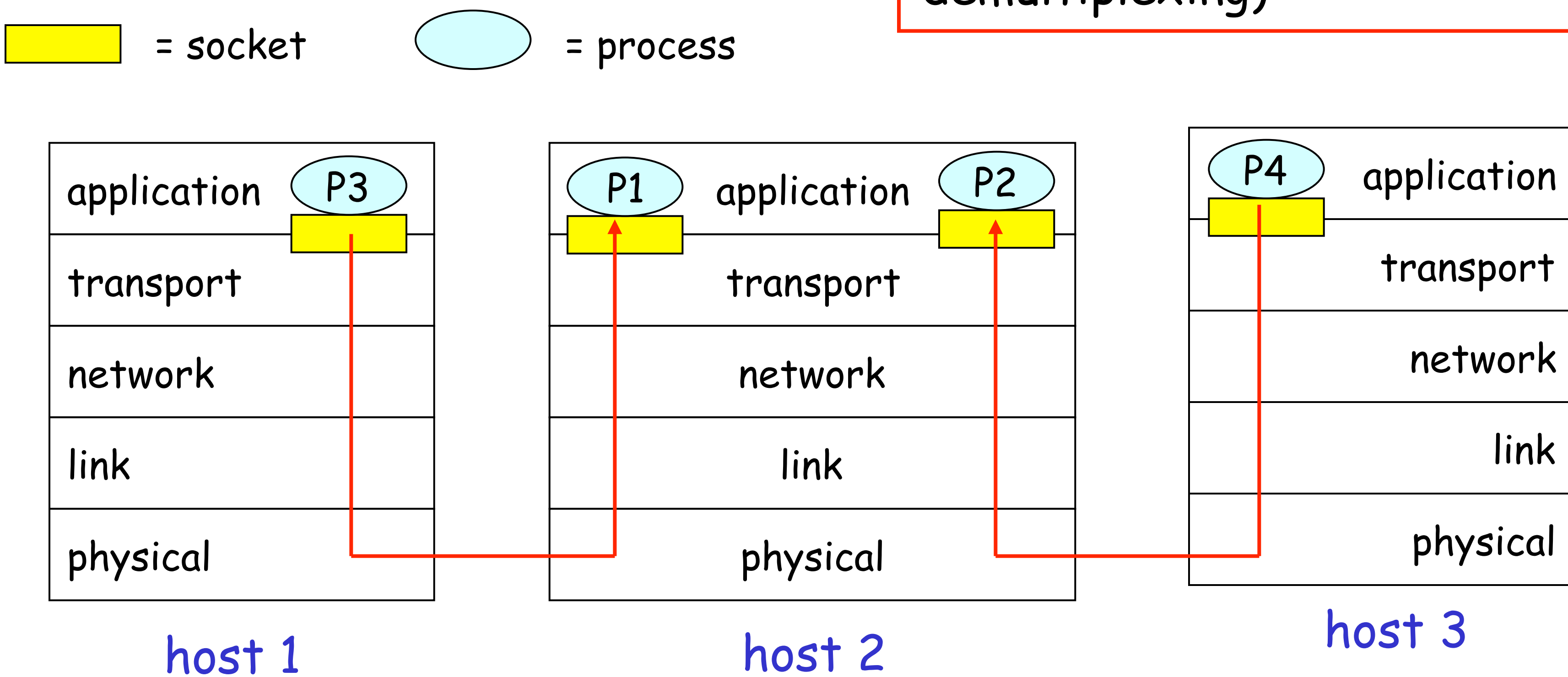
Multiplexing/demultiplexing

Demultiplexing at rcv host:

delivering received segments to correct socket

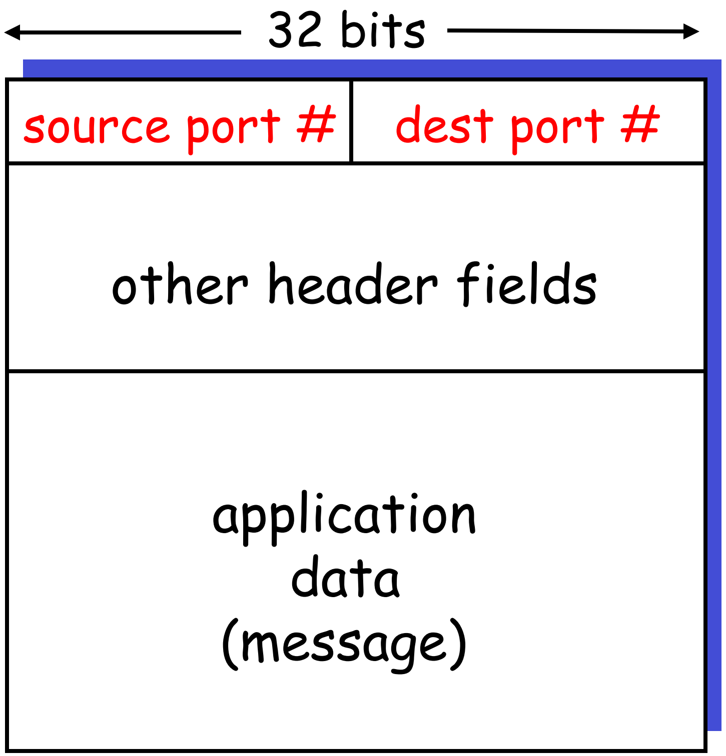
Multiplexing at send host:

gathering data from multiple sockets, enveloping data with header (later used for demultiplexing)



How demultiplexing works

- host receives IP datagrams
  - each datagram has source IP address, destination IP address
  - each datagram carries 1 transport-layer segment
  - each segment has source, destination port number (recall: well-known port numbers for specific applications)
- host uses IP addresses & port numbers to direct segment to appropriate socket



TCP/UDP segment format

Connectionless demultiplexing

- Create sockets with port numbers:

```
DatagramSocket mySocket1 = new
   DatagramSocket(99111);
DatagramSocket mySocket2 = new
   DatagramSocket(99222);
```

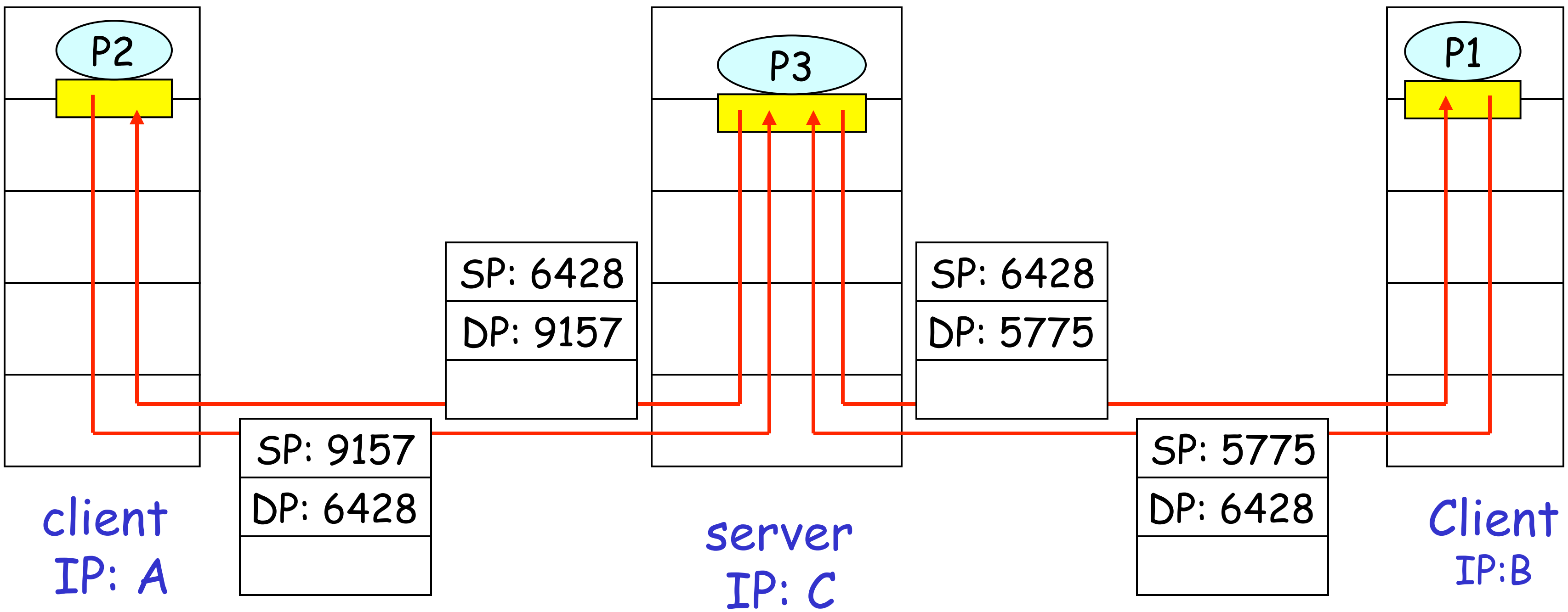
- UDP socket identified by two-tuple:

(dest IP address, dest port number)

- When host receives UDP segment:
  - checks destination port number in segment
  - directs UDP segment to socket with that port number
- IP datagrams with different source IP addresses and/or source port numbers directed to same socket

Connectionless demux (cont)

```
DatagramSocket serverSocket = new DatagramSocket(6428);
```

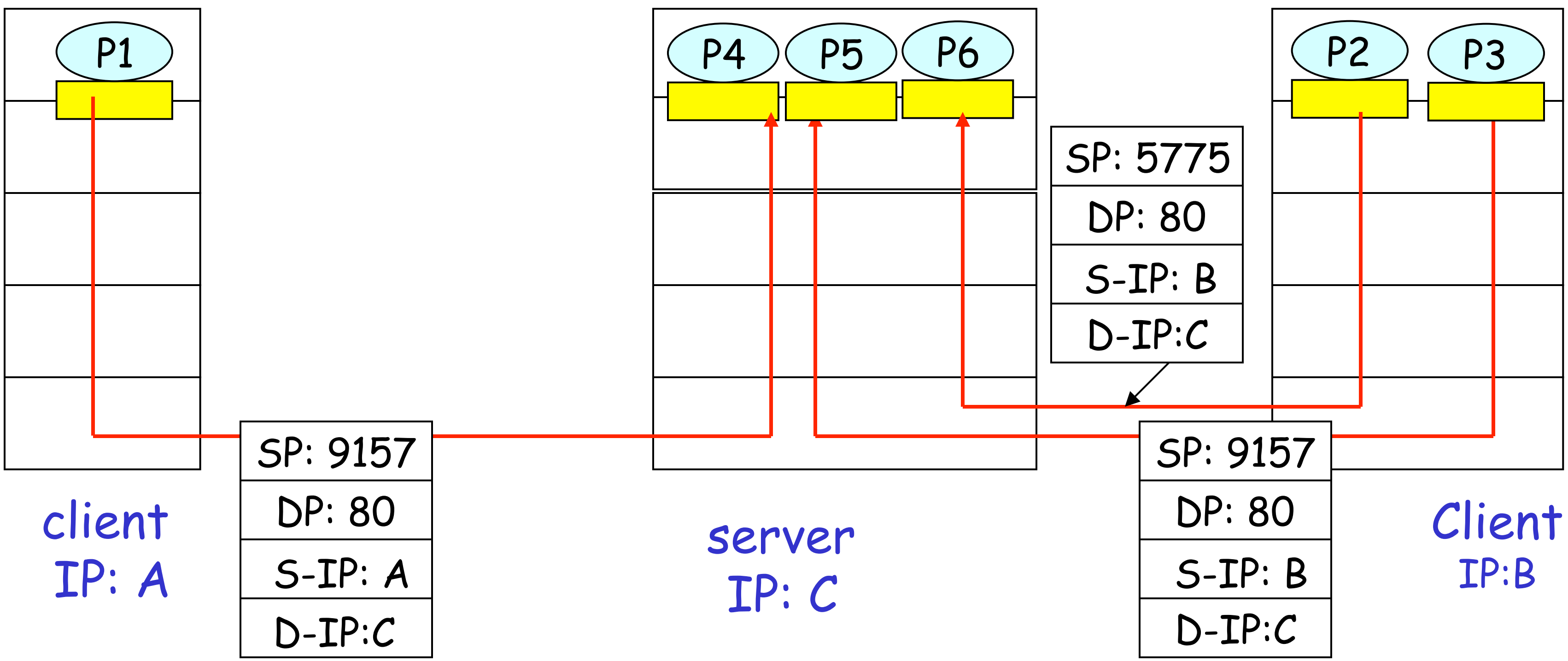


SP provides "return address"

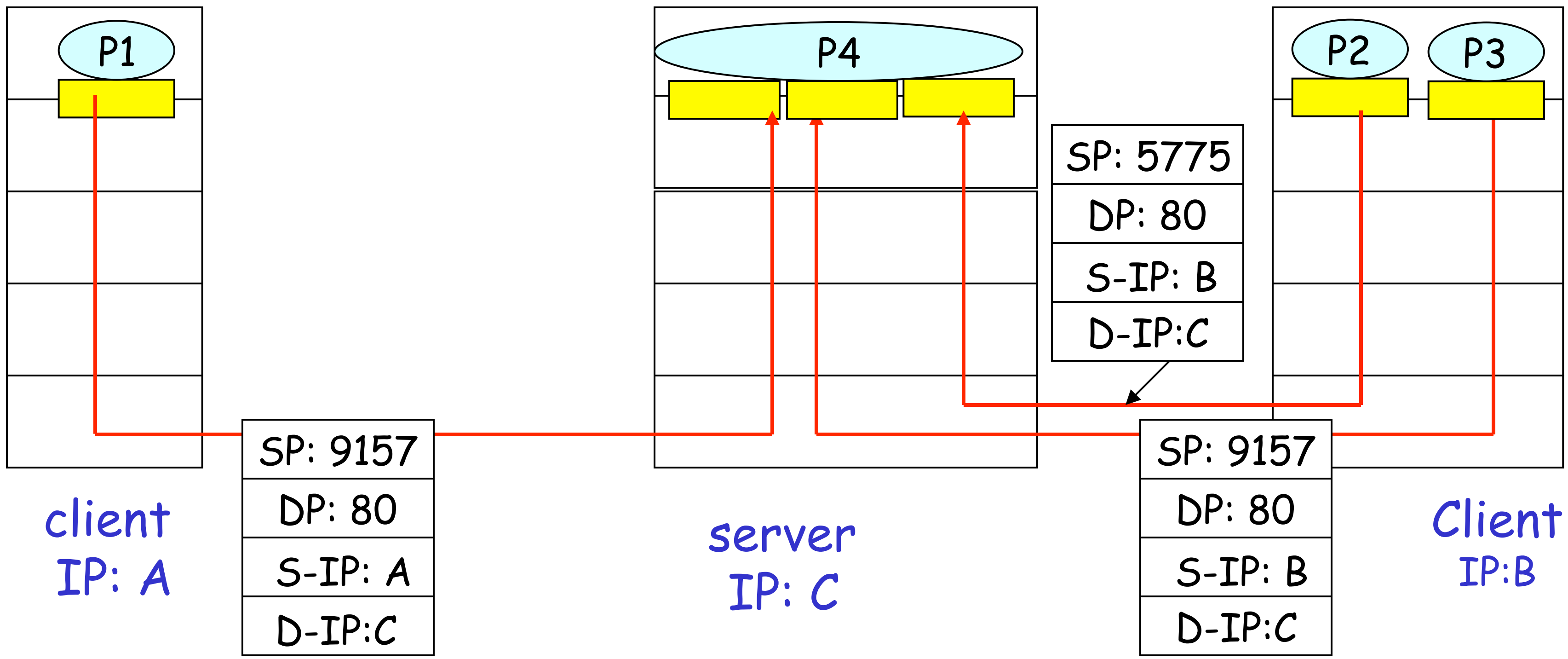
Connection-oriented demux

- TCP socket identified by 4-tuple:
  - source IP address
  - source port number
  - dest IP address
  - dest port number
- recv host uses all four values to direct segment to appropriate socket
- Server host may support many simultaneous TCP sockets:
  - each socket identified by its own 4-tuple
- Web servers have different sockets for each connecting client
  - non-persistent HTTP will have different socket for each request

Connection-oriented demux (cont)



Connection-oriented demux: Threaded Web Server



Chapter 3 outline

- 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

UDP: User Datagram Protocol [RFC 768]

- "no frills," "bare bones" Internet transport protocol
- "best effort" service, UDP segments may be:
  - lost
  - delivered out of order to app

reliable transfer over UDP: add reliability at application layer
  - application-specific error recovery!

- *connectionless:*
  - no handshaking between UDP sender, receiver
  - each UDP segment handled independently of others

Why is there a UDP?

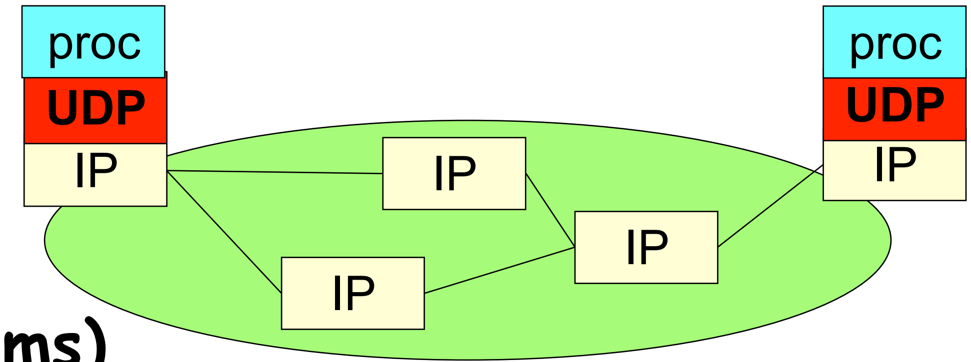
- no connection establishment (which can add delay)
- **simple: no connection state at sender, receiver**
- **small segment header (8 byte)**
- no congestion control: UDP can blast away as fast as desired

- often used for streaming multimedia apps
  - loss tolerant
  - rate sensitive

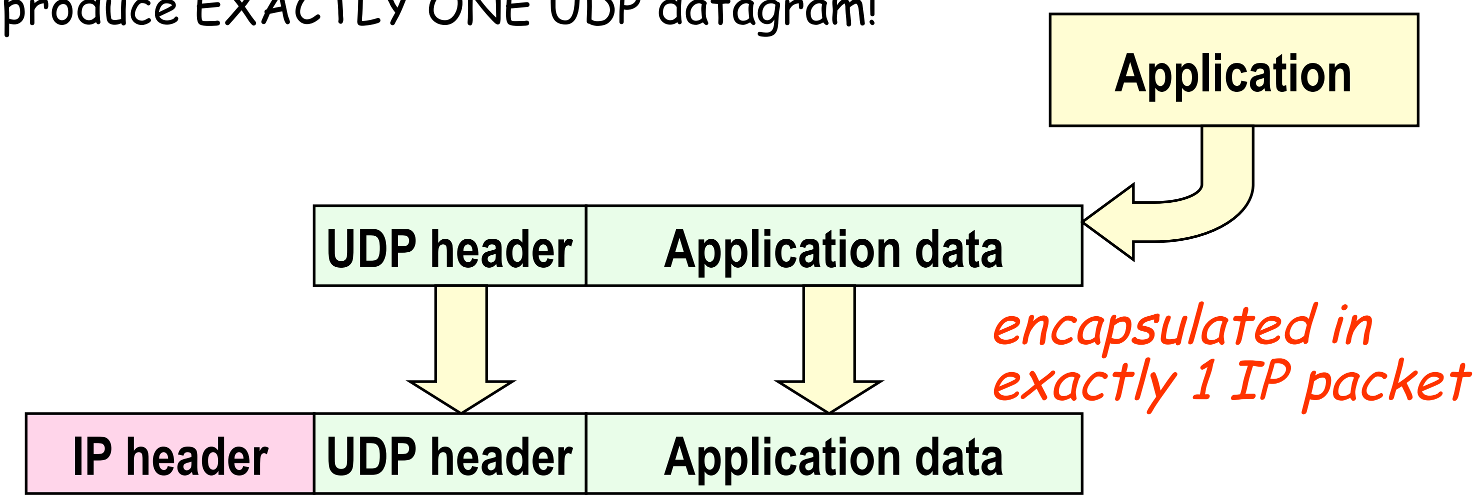
other UDP uses: DNS, SNMP..

UDP Packets

- **Connection-Less**
  - (no handshaking)



- **UDP packets (Datagrams)**
  - Each application interacts with UDP transport sw to produce EXACTLY ONE UDP datagram!



This is why, improperly, we use the term UDP packets

UDP datagram format

8 bytes header + variable payload

0 … 7 … 15	15 … 23 … 31
source port	destination port
length (bytes)	Checksum
Data	

- **UDP length field**
  - all UDP datagram
  - (header + payload)
- **payload sizes allowed:**
  - Empty
  - even size (bytes)

➔**UDP functions limited to:**

⇨**addressing**

→which is the only strictly necessary role of a transport protocol

⇨**Error checking**

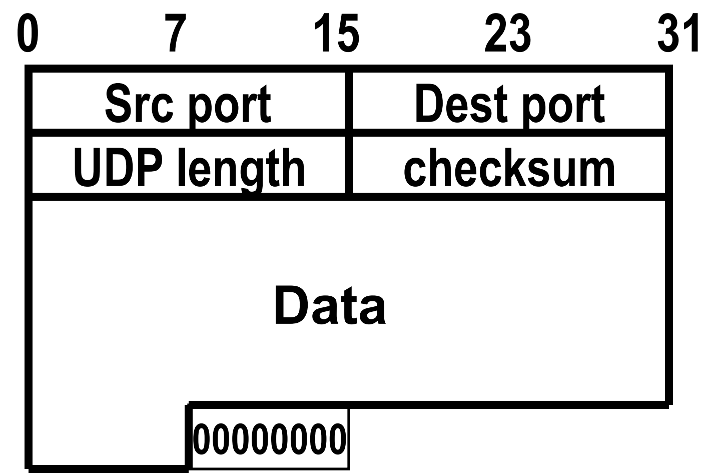
→which may even be **disabled** for performance

Maximum UDP datagram size

- 16 bit UDP length field:
  - Maximum up to 2^{16-1} = 65535 bytes
  - Includes 8 bytes UDP header (max data = 65527)
- But max IP packet size is also 65535
  - Minus 20 bytes IP header, minus 8 bytes UDP header
  - Max UDP_data = **<u>65507</u>** bytes!
- Moreover, most OS impose further limitations!
  - most systems provide 8192 bytes maximum (max size in NFS)
  - some OS had (still have?) internal implementation features (bugs?) that limit IP packet size
    - SunOS 4.1.3 had 32767 for max tolerable IP packet transmittable (but 32786 in reception…) – bug fixed only in Solaris 2.2
- Finally, subnet Maximum Transfer Unit (MTU) limits may fragment datagram – annoying for reliability!
  - E.g. ethernet = 1500 bytes; PPP on your modem = 576

Error checksum

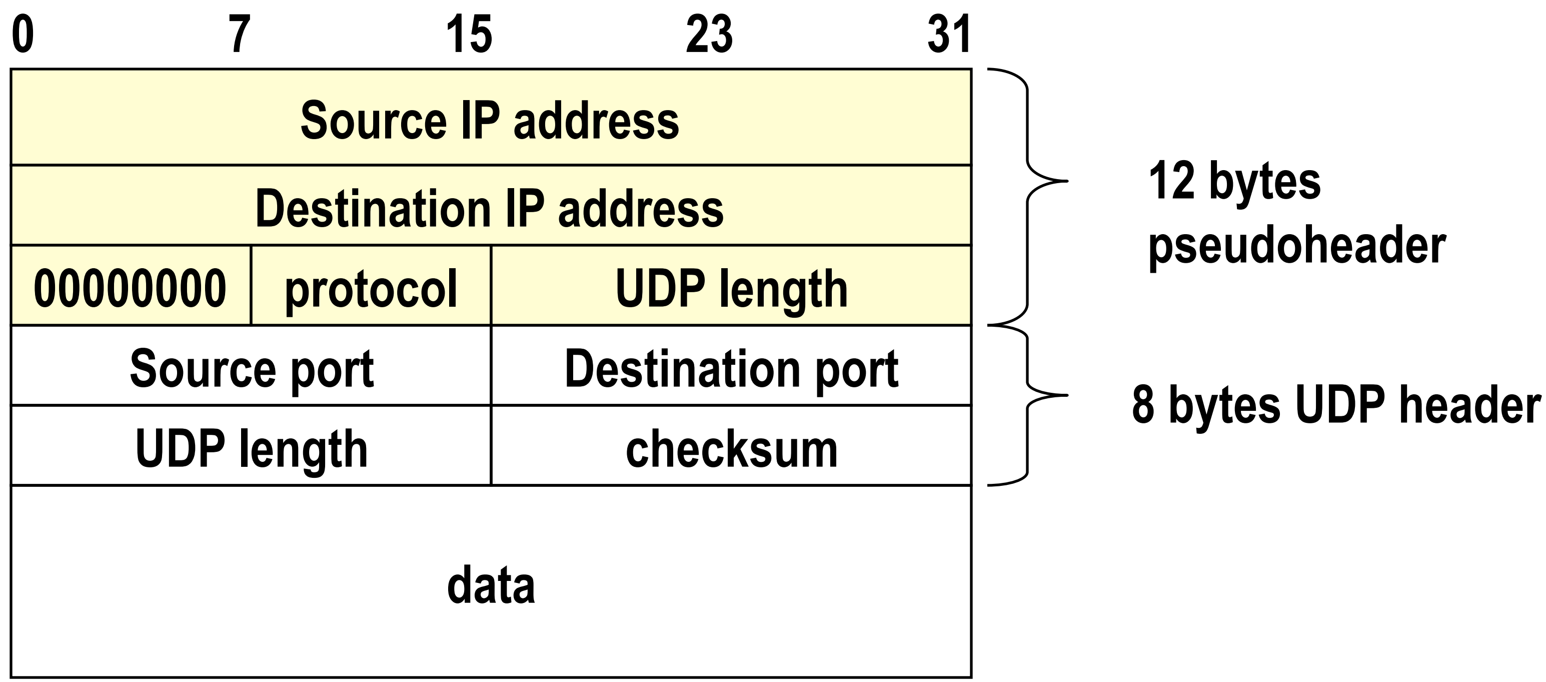
- 16 bit checksum field, obtained by:
  - summing up all 16 bit words in header data and **pseudoheader**, in 1's complement (checksum fields filled with 0s initially)
  - take 1's complement of result
  - if result is 0, set it to 111111…11 (65535==0 in 1's complement)
  - Sender puts checksum value into UDP checksum field
- at destination:
  - 1's complement sum should return 0, otherwise error detected
  - upon error, no action (just packet discard)
- efficient implementation RFC 1071



- Zero padding
  - To multiple of 16 bits
- checksum disabled
  - by source, by setting 0 in the checksum field

Pseudo header

- Is not transmitted!
  - But it is information available at transmitter and at receiver
  - intention: double check that packet has arrived at correct destination



Protocol field (TCP=6,UDP=17) necessary, as same checksum calculation used in TCP. UDP length duplicated.

disabling checksum

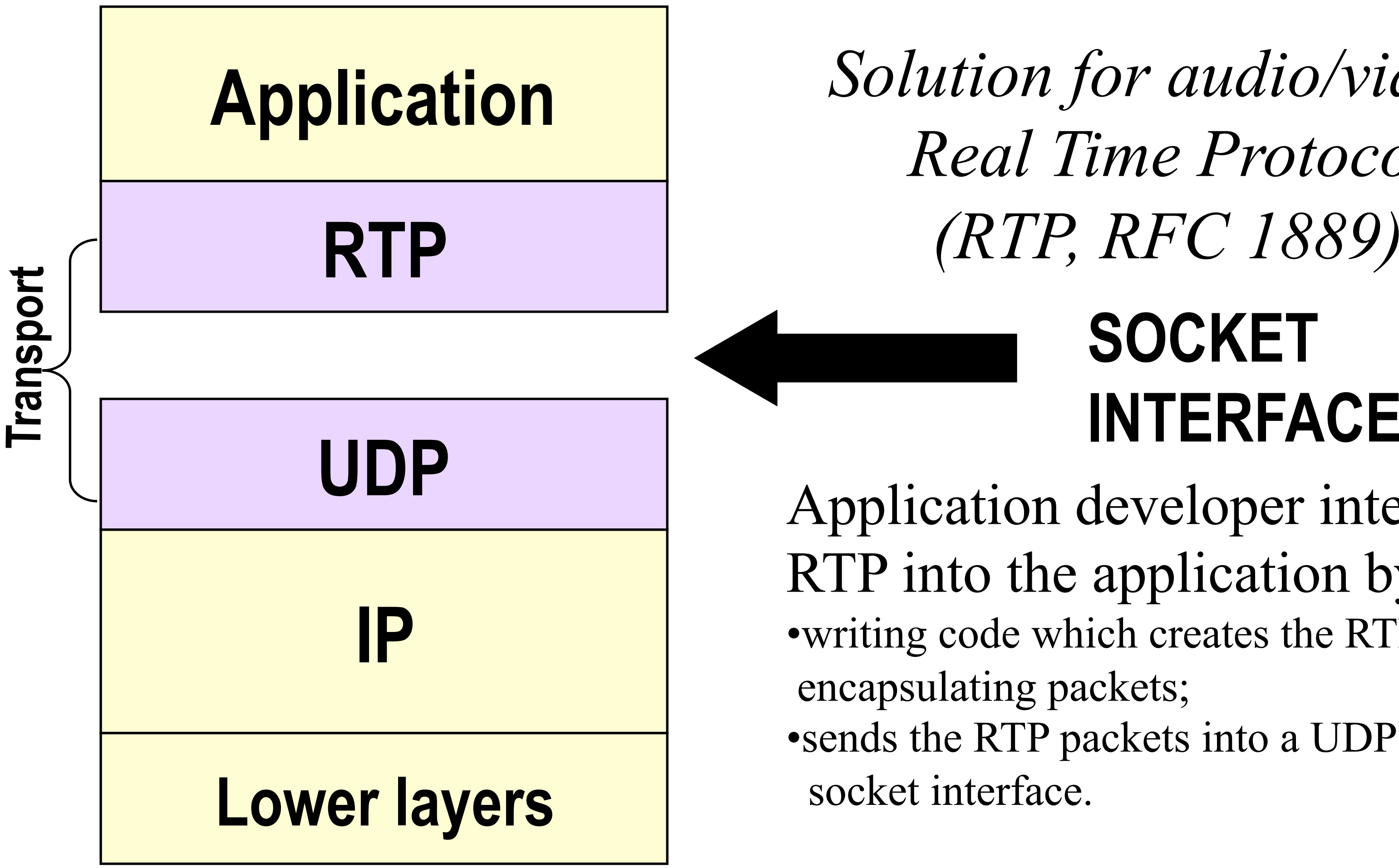
- In principle never!
  - Remember that IP packet checksum DOES NOT include packet payload.
- In practice, often done in NFS
  - sun was the first, to speed up implementation
- may be tolerable in LANs under one's control.
- Definitely dangerous in the wide internet
  - Exist layer 2 protocols without error checking

UDP: a lightweight protocol

- No connection establishment
  - no initial overhead due to handshaking
- No connection state
  - greater number of supported connections by a server!
- Small packet header overhead
  - 8 bytes only vs 20 in TCP
- originally intended for simple applications, oriented to short information exchange
  - DNS
  - management (e.g. SNMP)
  - etc
- No rate limitations
  - No throttling due to congestion & flow control mechanisms
  - No retransmission (for certain application loss tolerable)
- extremely important features for today multimedia applications! Expecially for real time applications which can tolerate some packet loss but require a minimum send rate.

RTP as seen from Application

Be careful: UDP ok for multimedia because it does not provide anything at all (no features = no limits!). Application developers have to provide supplementary transport capabilities at the application layer!



Solution for audio/video: Real Time Protocol (RTP, RFC 1889)

Application developer integrates RTP into the application by:

- writing code which creates the RTP encapsulating packets;
- sends the RTP packets into a UDP socket interface.

Details of RTP in subsequent courses – unless we are ahead of schedule

Chapter 3 outline

- 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

A MUCH more complex transport for three main reasons

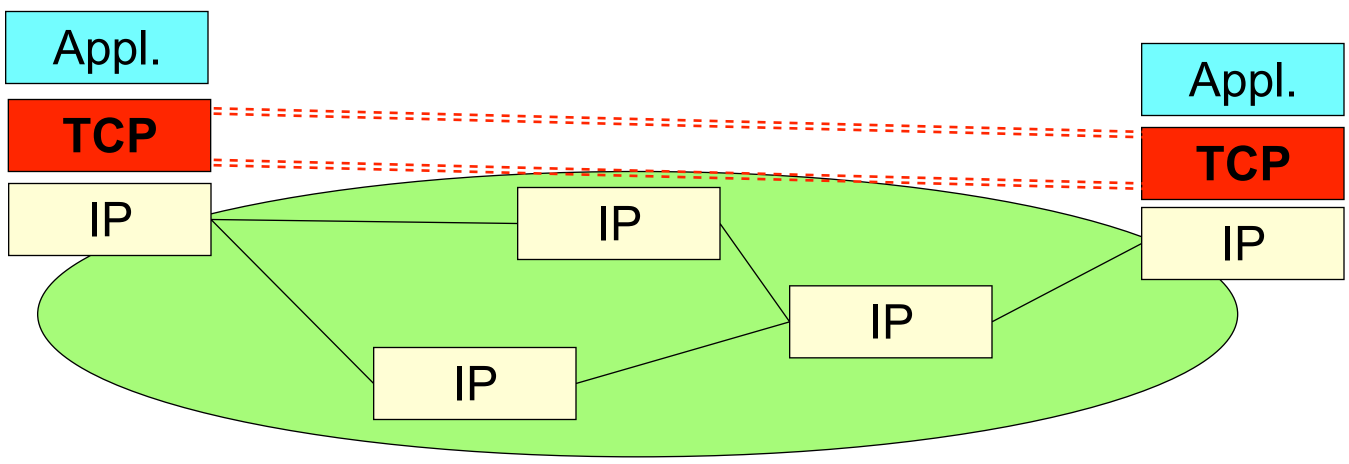
- Connection oriented
  - implements mechanisms to setup and tear down a full duplex connection between end points
- Reliable
  - implements mechanisms to guarantee error free and ordered delivery of information
- Flow & Congestion controlled
  - implements mechanisms to control traffic

TCP services

- connection oriented
  - TCP connections
- *reliable* transfer service
  - all bytes sent are received

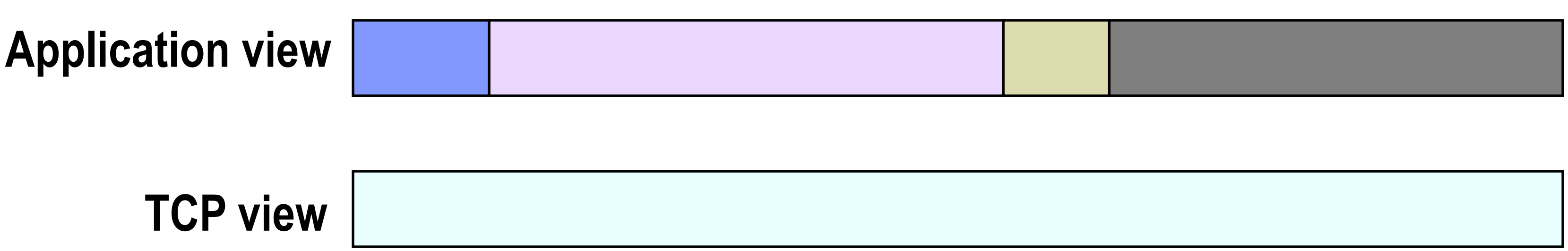
→ TCP functions

- → application addressing (ports)
- → error recovery (acks and retransmission)
- → reordering (sequence numbers)
- → flow control
- → congestion control



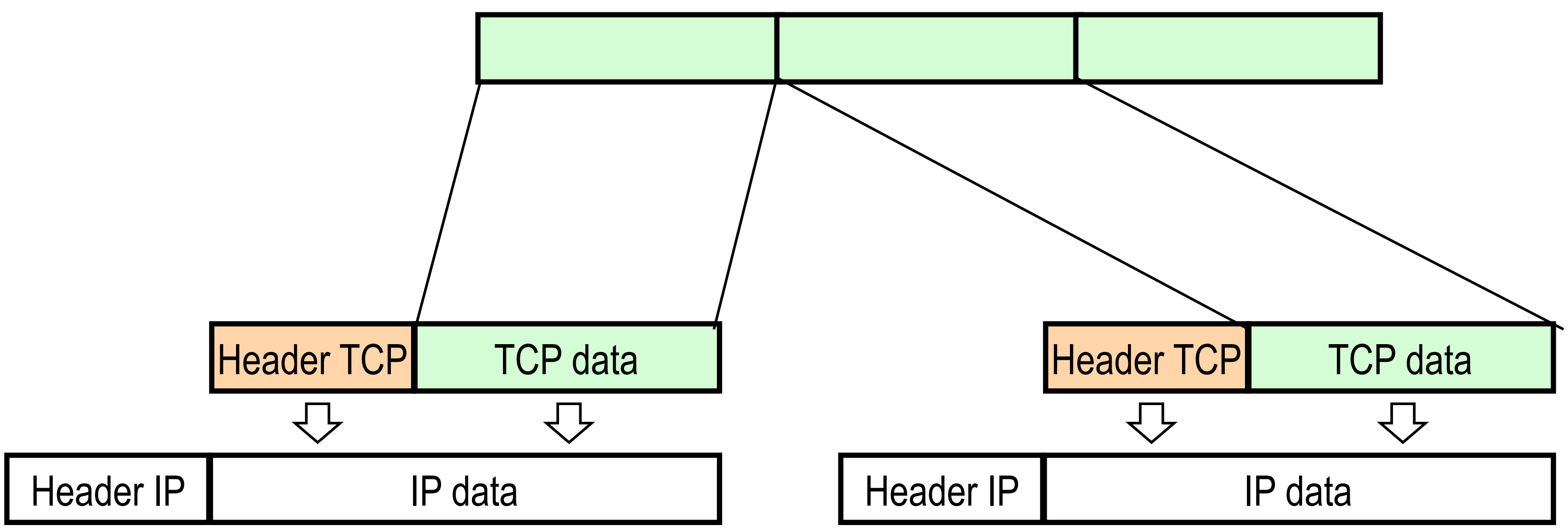
Byte stream service

- TCP exchange data between applications as a stream of bytes.
- It does not introduce any data delimiter (an application duty)
  - source application may enter 10 bytes followed by 1 and 40 (grouped with some semantics)
  - data is buffered at source, and transmitted
  - at receiver, may be read in the sequence 25 bytes, 22 bytes and 4 bytes...



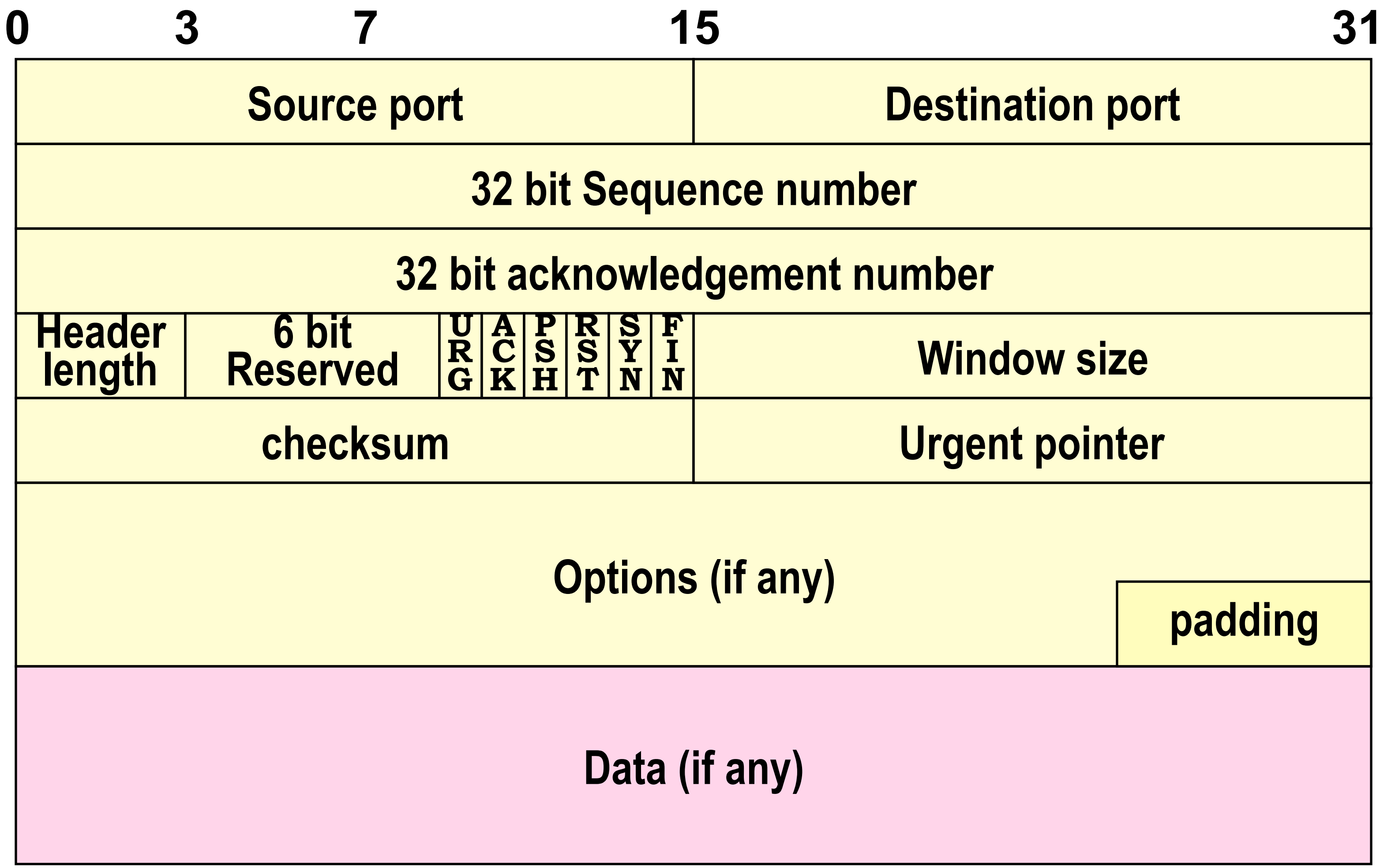
TCP segments

- Application data broken into segments for transmission
- segmentation totally up to TCP, according to what TCP considers being the best strategy
- each segment placed into an IP packet
- very different from UDP!!



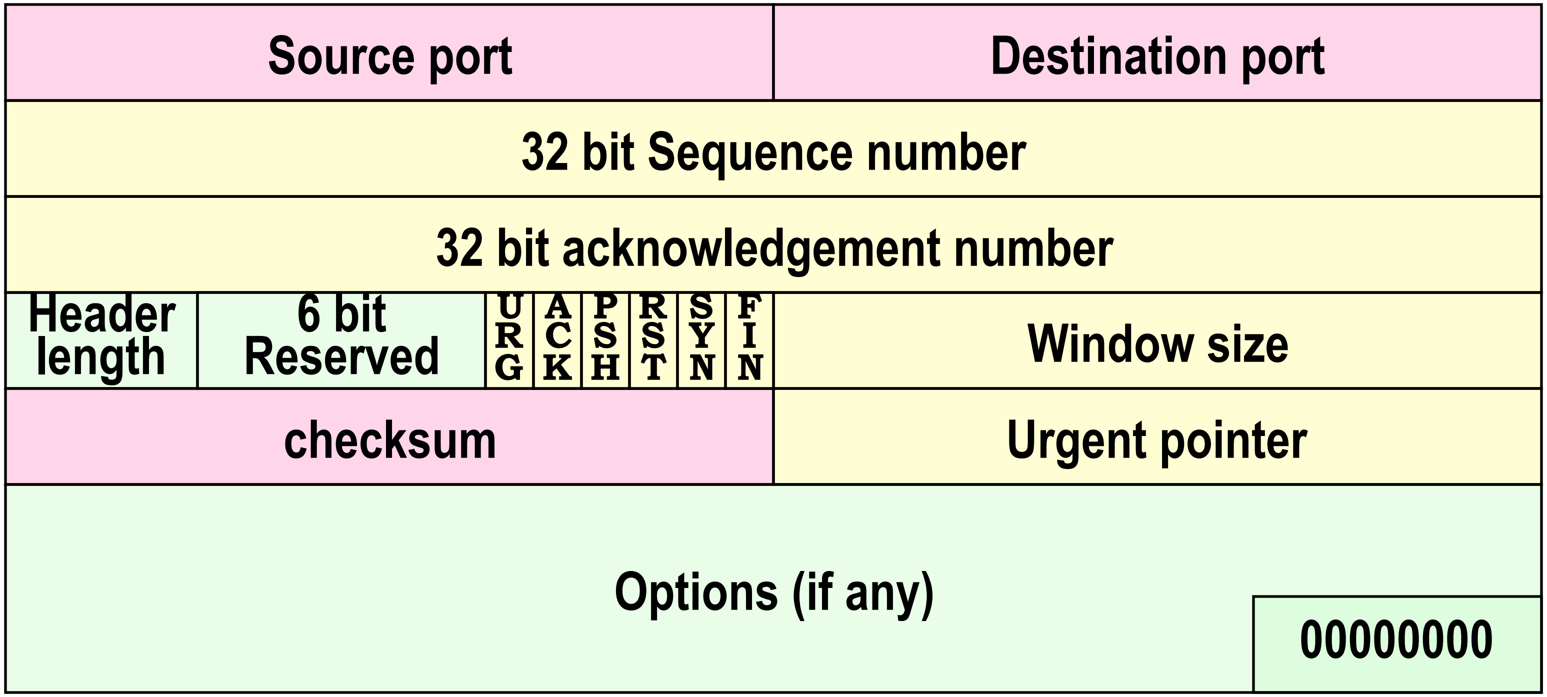
TCP segment format

20 bytes header (minimum)



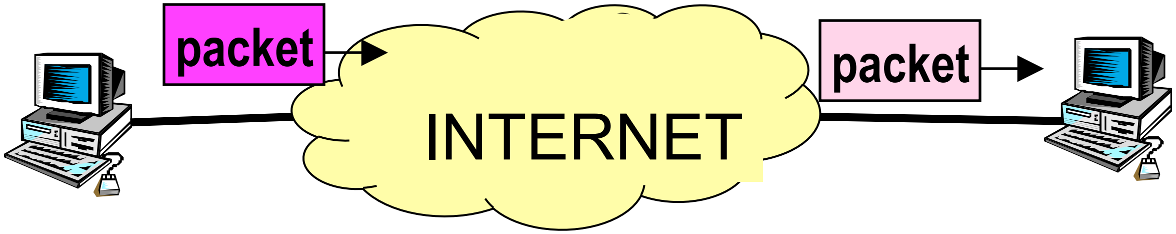
Source port								Destination port
32 bit Sequence number								
32 bit acknowledgement number								
Header length	6 bit Reserved	URG	ACK	PSH	RST	SYN	FIN	Window size
checksum								Urgent pointer

- Source & destination port + source and destination IP addresses
  - univocally determine TCP connection
- checksum as in UDP
  - same calculation including same pseudoheader
- no explicit segment length specification



- Header length: 4 bits
    - specifies the header size (n*4byte words) for options
    - maximum header size: 60 (15*4)
    - option field size must be multiple of 32bits: zero padding when not.
- Reserved: 000000 (still today!)

Reliable data transfer: issues



PROBLEMS:

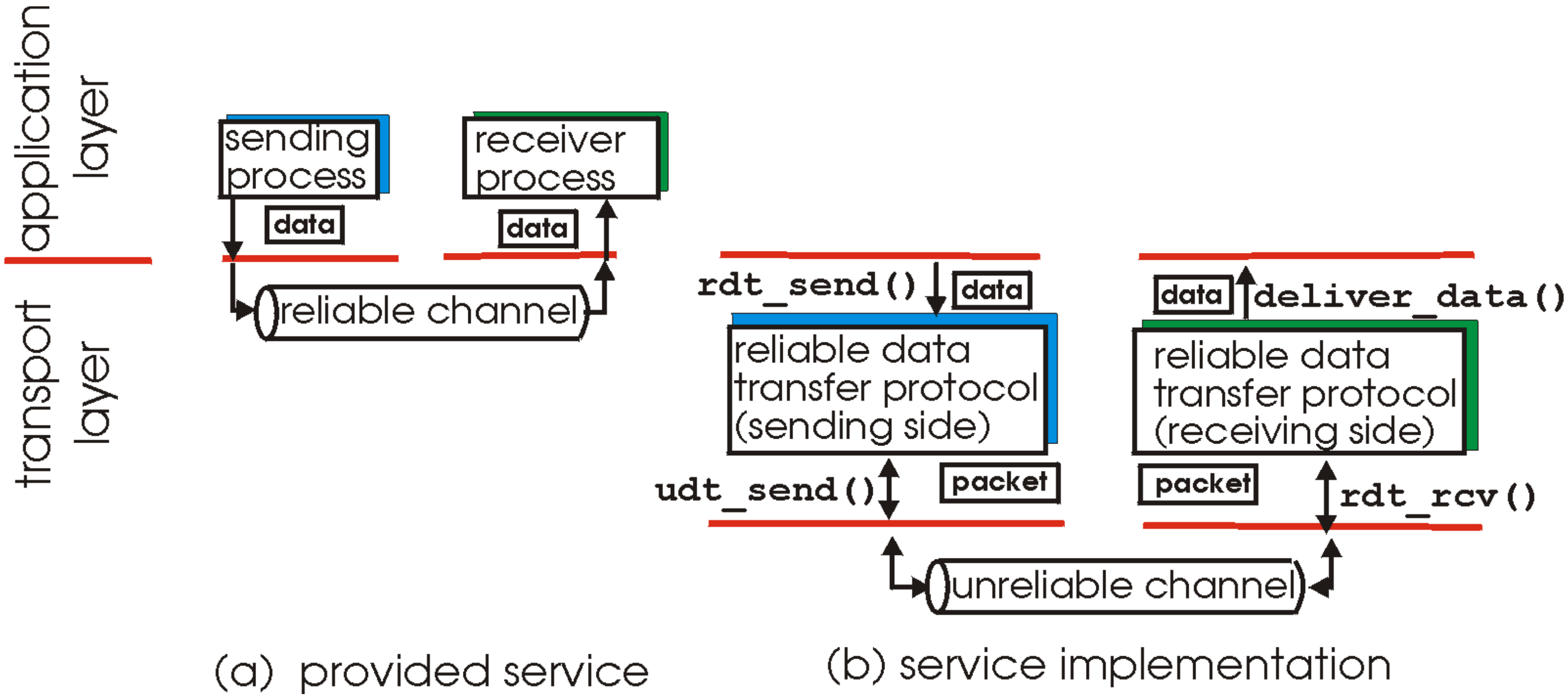
1) Packet received with errors
2) Packet not received at all

Same problem considered at DATA LINK LAYER
(although it is less likely that a whole packet is lost at data link)

- mechanisms to guarantee correct reception:
  - Forward Error Correction (FEC) coding schemes
    - Powerful to correct bits affected by error, not effective in case of packet loss
    - Mostly used at link layer
  - Error detection (e.g. checksum used in UDP)
  - Retransmission – issues:
    - ACK
    - NACK
    - TIMEOUT

Principles of Reliable data transfer

- important in app., transport, link layers
- top-10 list of important networking topics!

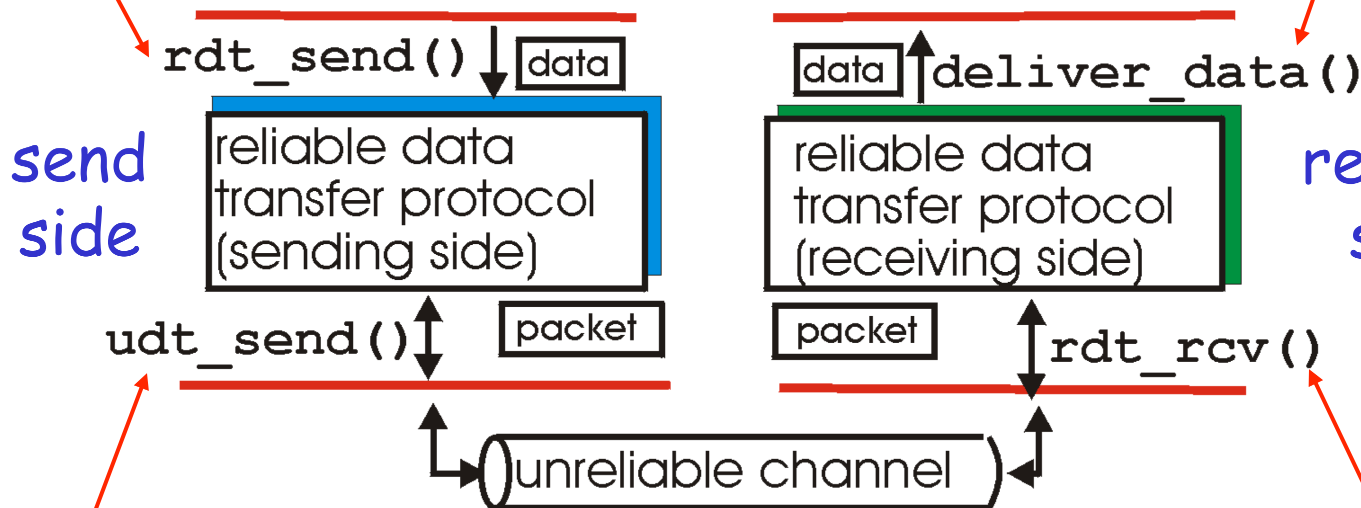


- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

Reliable data transfer: getting started

rdt_send(): called from above, (e.g., by app.). Passed data to deliver to receiver upper layer

deliver_data(): called by **rdt** to deliver data to upper



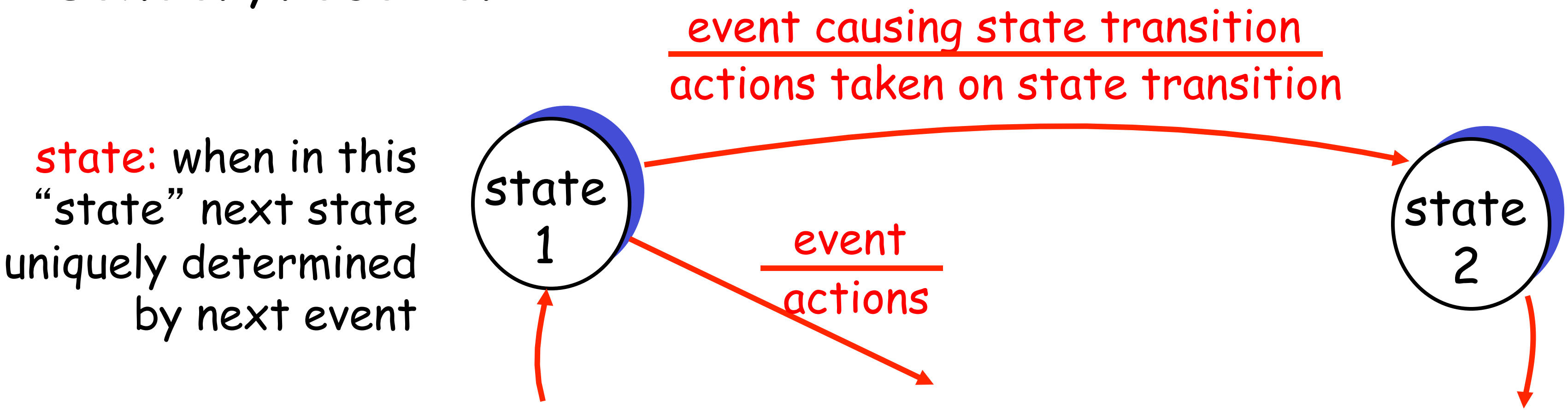
udt_send(): called by rdt, to transfer packet over unreliable channel to receiver

rdt_rcv(): called when packet arrives on rcv-side of channel

Reliable data transfer: getting started

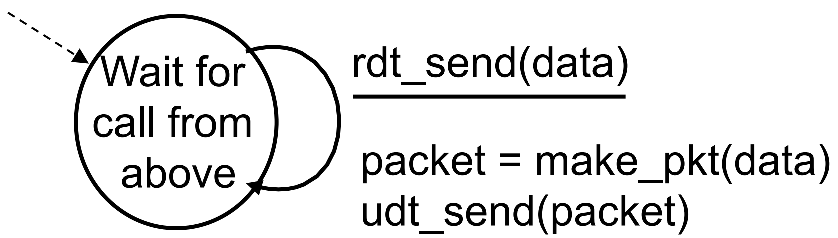
We'll:

- incrementally develop sender, receiver sides of reliable data transfer protocol (rdt)
- consider only unidirectional data transfer
  - but control info will flow on both directions!
- use finite state machines (FSM) to specify sender, receiver

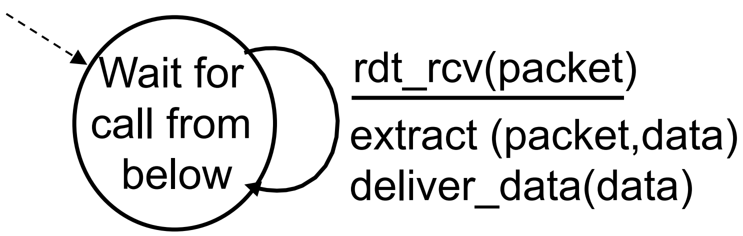


Rdt1.0: reliable transfer over a reliable channel

- underlying channel perfectly reliable
  - no bit errors
  - no loss of packets (→no congestion, no buffer overflows)
- separate FSMs for sender, receiver:
  - sender sends data into underlying channel
  - receiver read data from underlying channel



sender

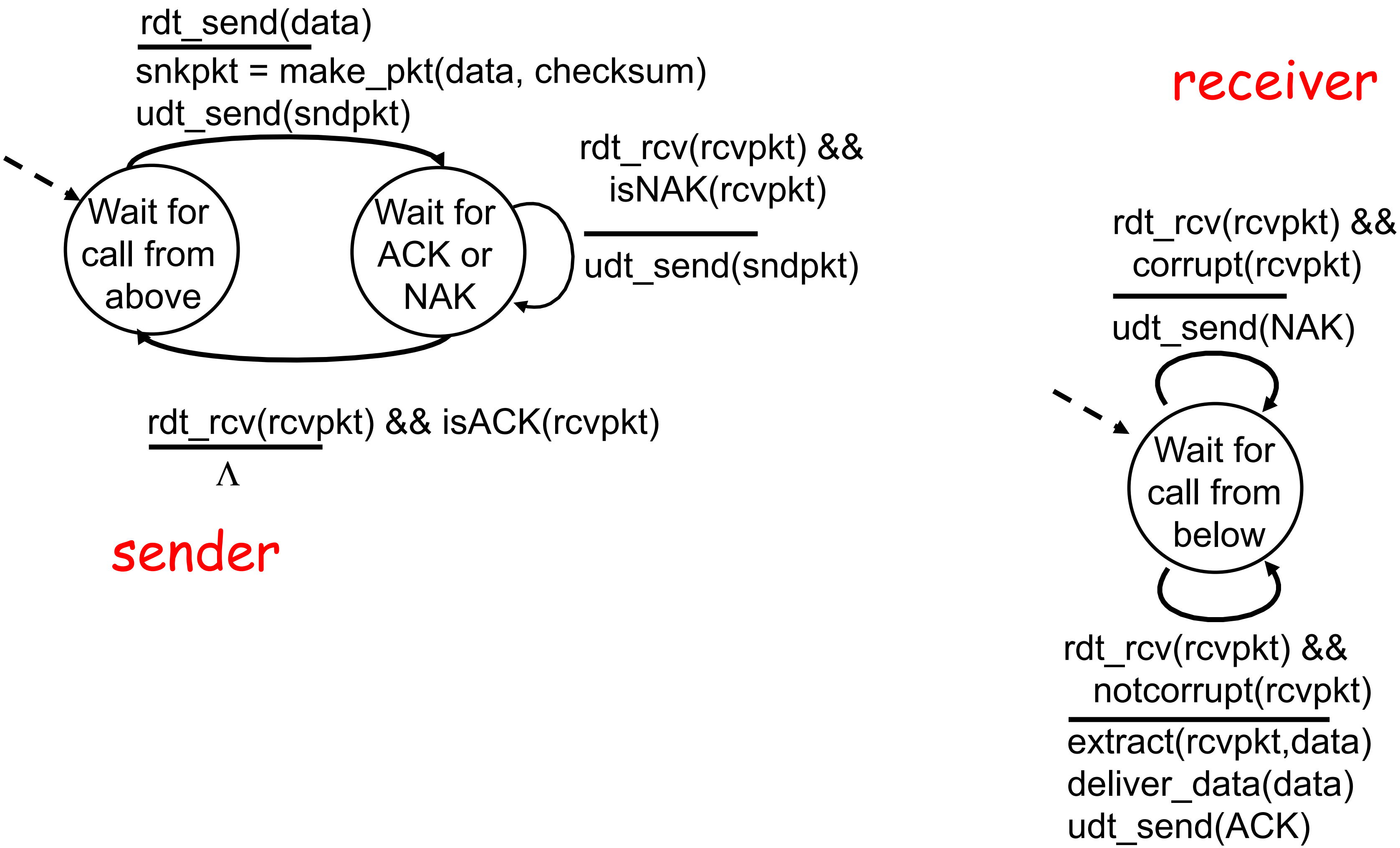


receiver

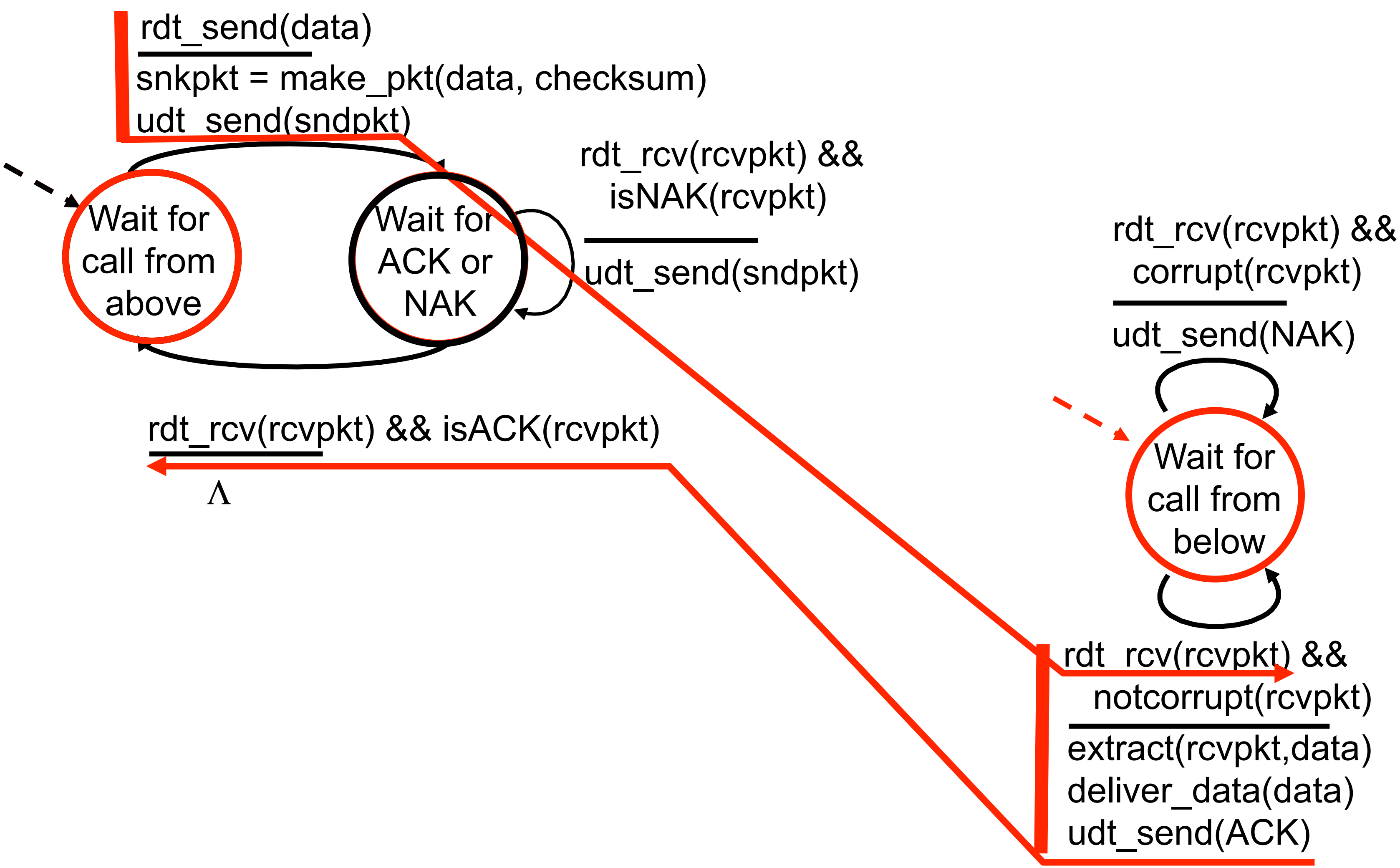
Rdt2.0: channel with bit errors

- underlying channel may flip bits in packet
  - recall: UDP checksum to detect bit errors
- **Still no loss!!**
- *the* question: how to recover from errors:
  - *acknowledgements (ACKs):* receiver explicitly tells sender that pkt received OK
  - *negative acknowledgements (NAKs):* receiver explicitly tells sender that pkt had errors
  - sender retransmits pkt on receipt of NAK
  - human scenarios using ACKs, NAKs?
- new mechanisms in `rdt2.0` (beyond `rdt1.0`):
  - error detection
  - receiver feedback: control msgs (ACK,NAK) rcvr->sender

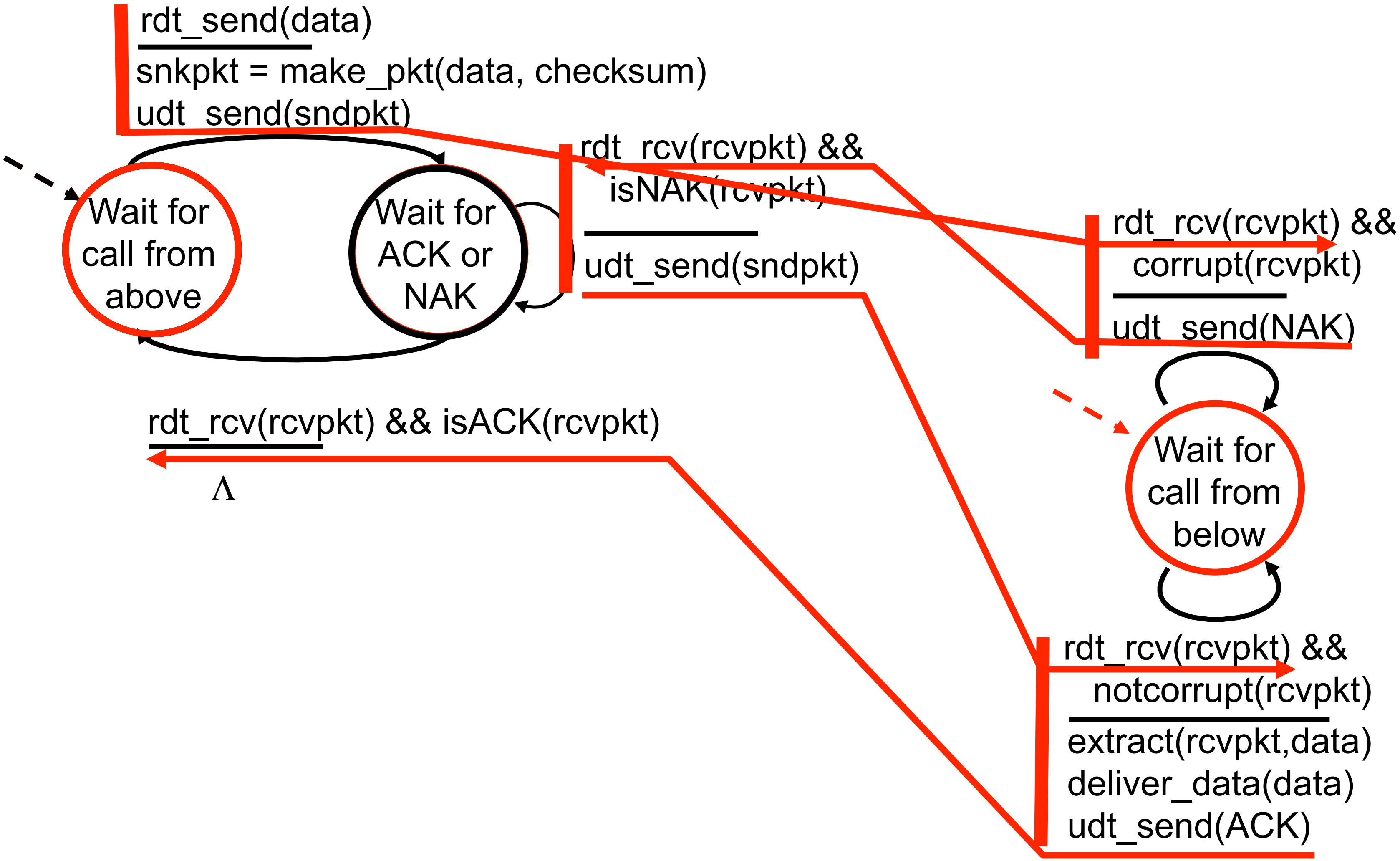
rdt2.0: FSM specification



rdt2.0: operation with no errors



rdt2.0: error scenario



rdt2.0 has a fatal flaw!

What happens if ACK/NAK corrupted?

- sender doesn't know what happened at receiver!
- can't just retransmit: possible duplicate

What to do?

- sender ACKs/NAKs receiver's ACK/NAK? What if sender ACK/NAK lost?
- retransmit, but this might cause retransmission of correctly received pkt!

Handling duplicates:

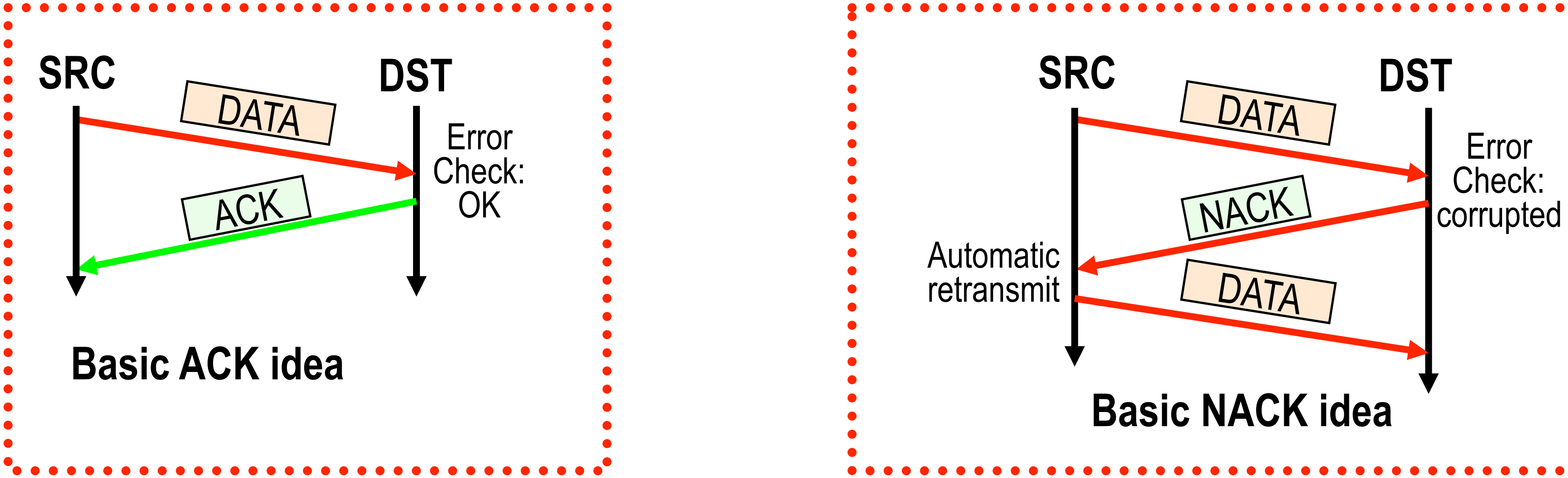
- sender adds *sequence number* to each pkt
- sender retransmits current pkt if ACK/NAK garbled
- receiver discards (doesn't deliver up) duplicate pkt

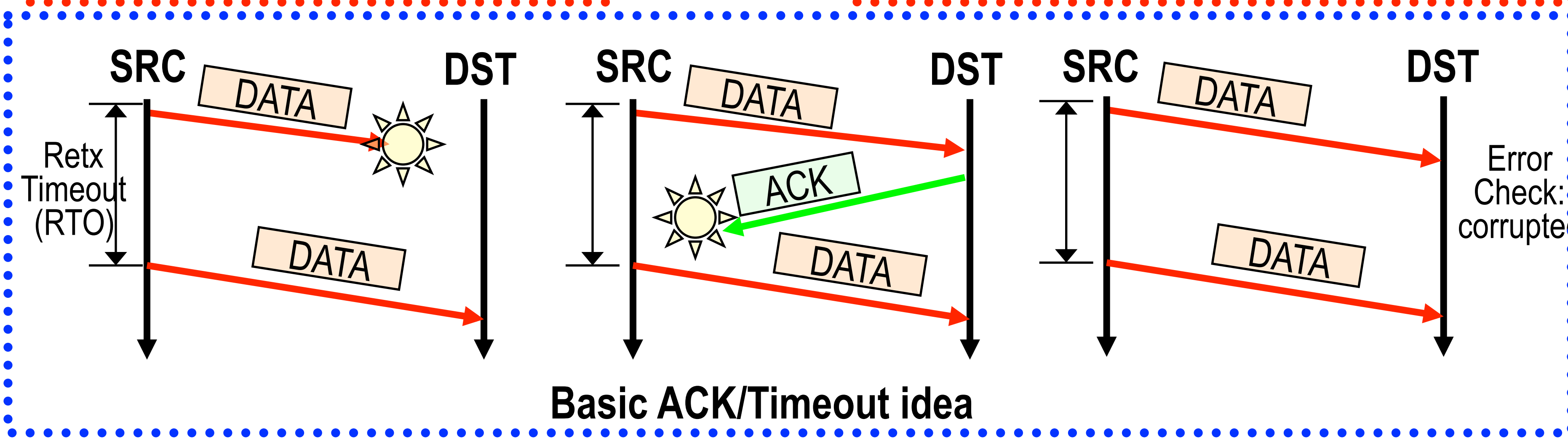
stop and wait
Sender sends one packet, then waits for receiver response

Retransmission scenarios

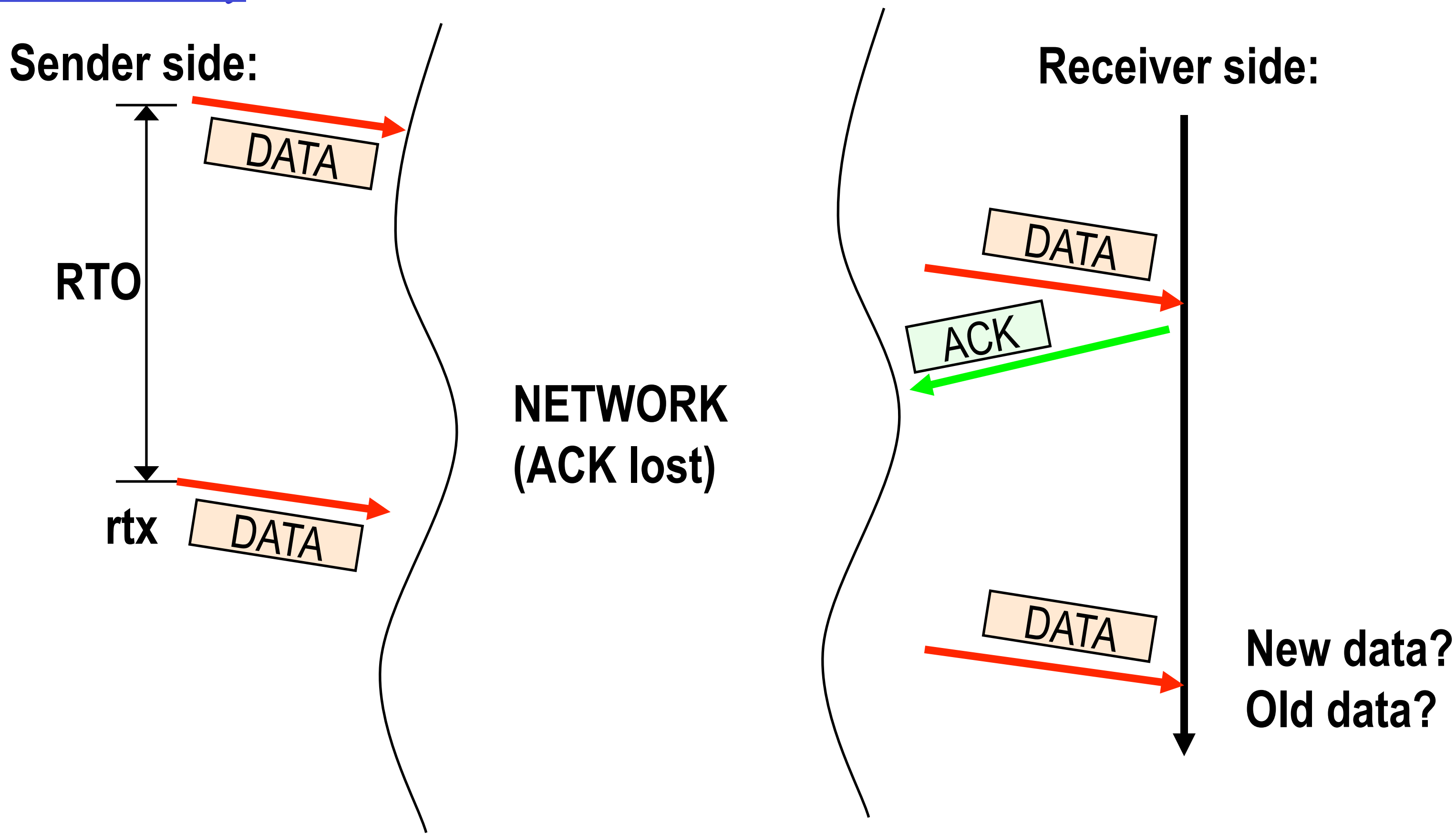
referred to as ARQ schemes (Automatic Retransmission reQuest)

COMPONENTS: a) error checking at receiver; b) feedback to sender; c) retx



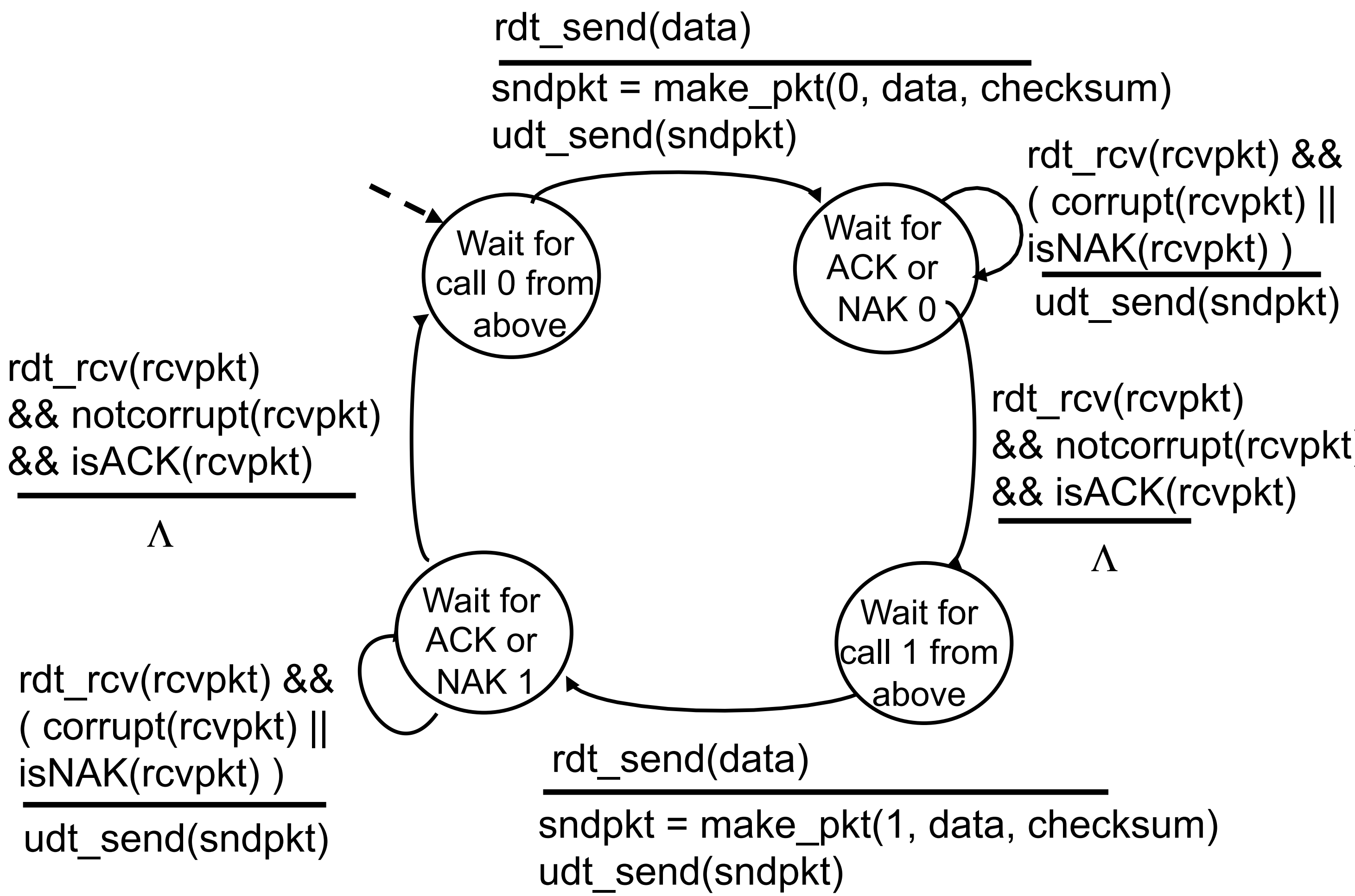


Why sequence numbers?
(on data)



Need to univocally "label" all packets circulating in the network between two end points. 1 bit (0-1) enough for Stop-and-wait

rdt2.1: sender, handles garbled ACK/NAKs



rdt2.1: receiver, handles garbled ACK/NAKs