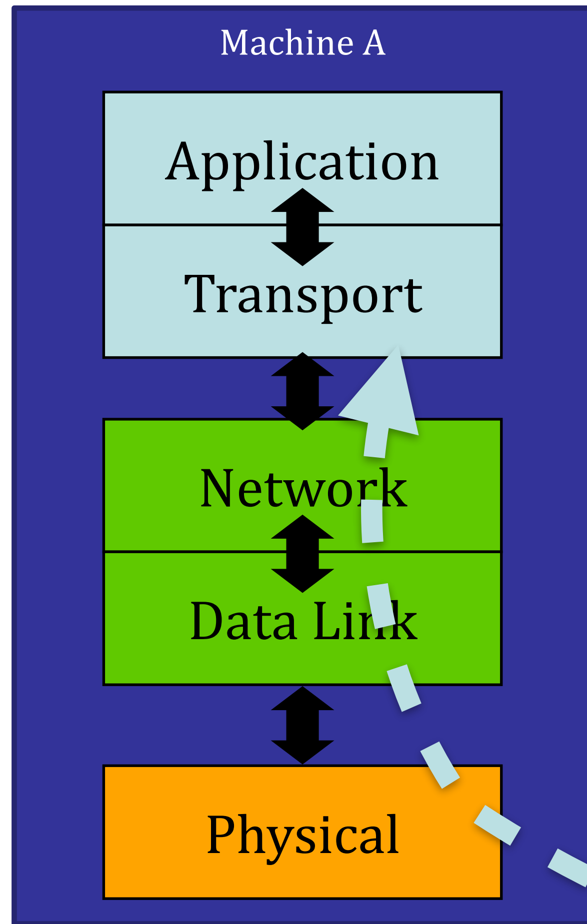
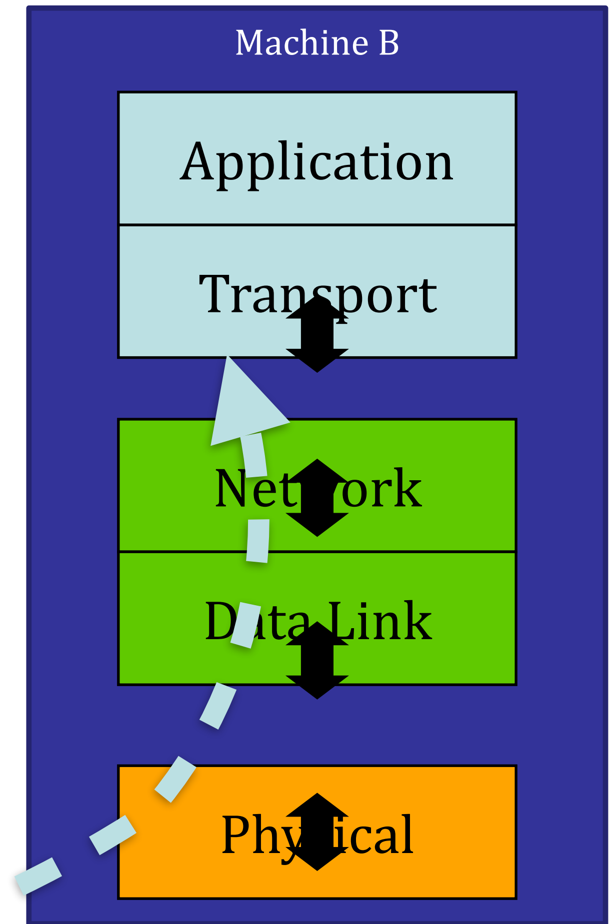


The Transport Layer:
TCP and UDP

Series of links, switches routers, LANs, …

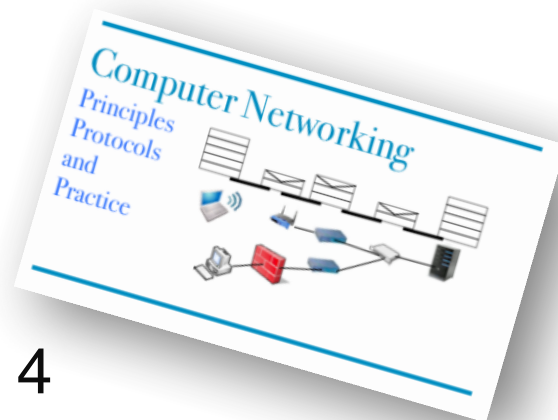


EPFL

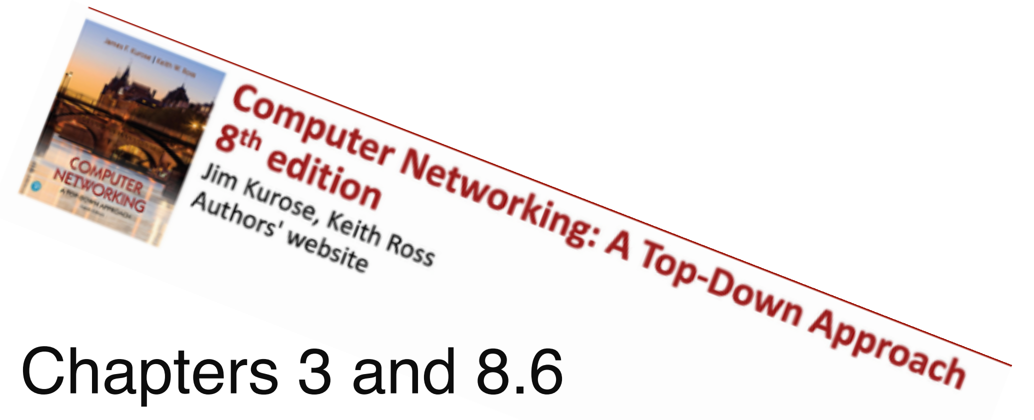
Contents

1. UDP
2. TCP Basics: Sliding Window and Flow Control
3. TCP Connections and Sockets
4. More TCP Bells and Whistles
5. Secure Transport
6. Where should packet losses be repaired ?

Textbook



Part 4



Chapters 3 and 8.6

Reminder from 1st lecture: Transport layer services

- Network + Data link + Physical layers carry packets end-to-end

- packets may be *lost* due to:
  - errors at the physical layer
  - buffer-overflow events at routers/switches
- or *reordered* due to:
  - following different paths to destination

- Transport layer
  - makes network services available to programs
  - is in end-systems only, *not* in routers' data plane (i.e. not at forwarding level)
  - may handle packet loss/reordering or not:
    - UDP (User Datagram Protocol): *not reliable* transfer, takes no action
    - TCP (Transmission Control Protocol): *reliable, in-order* transfer by using sophisticated mechanisms

What is the definition of a «server» in networks?

A. A machine that hosts resources to be used in the web
B. A computer with high CPU performance
C. A computer with large data storage
D. The role of a program that waits for requests to come
E. The role of a program that allows users to access large amounts of resources
F. None of the above
G. I don't know

Go to web.speakup.info or download speakup app

Join room 87072

Solution

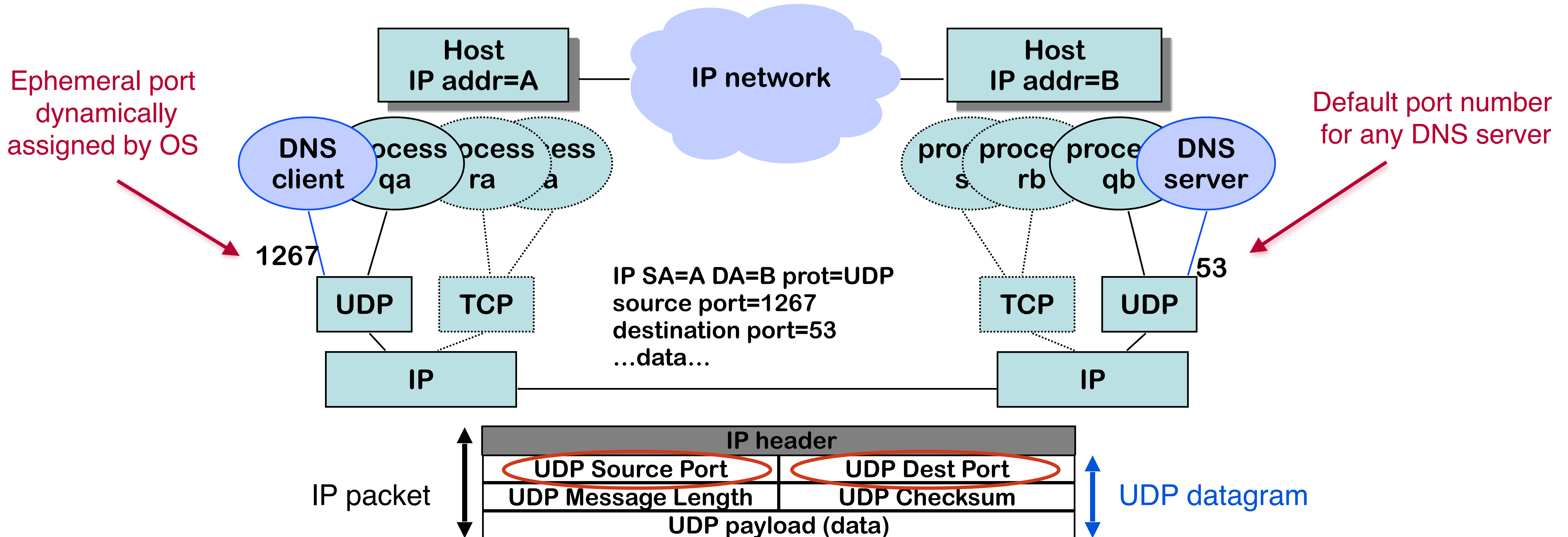
Answer D

Formally, a server is a role at the transport layer, where the program waits for requests to come.

In contrast, a client initiates communication to a server.

Reminder from 1st lecture: Port Numbers

- assigned by OS to identify processes within a host
- servers' port numbers must be *well-known* to clients (e.g. 53 for DNS, 80 for HTTP)
- src and dest port numbers are *inside transport-layer header*



The picture shows two processes (= network application programs) pa, and pb that are communicating. Each of them is associated locally with a port, as shown in the figure.

The example shows a packet sent by the name resolver process (DNS client) at host A, to the domain name server (DNS server) process at host B. The UDP header contains the source and destination ports.
The *destination port* number is used to contact the name server process at B;
the *source port* is not used directly; it will be used in the response from B to A.

The UDP header also contains a *checksum* of the UDP data plus the IP addresses and packet length. Checksum computation is not performed by all systems.

Ports are 16 bits unsigned integers. They are defined statically or dynamically. Typically, a server uses a port number defined statically.

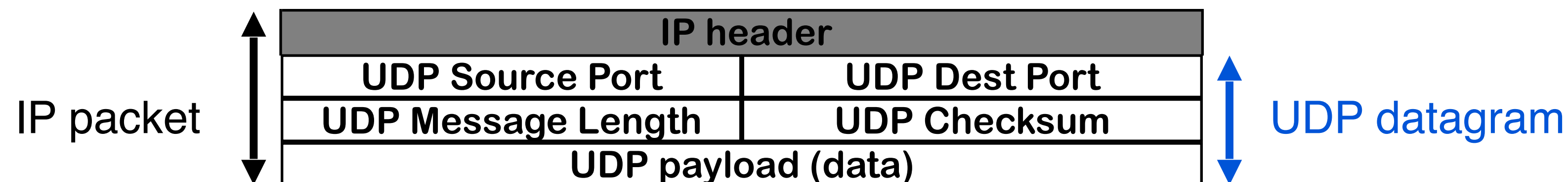
Standard services use well-known *default* port numbers; e.g., all DNS servers use port 53 (look at /etc/services).
Ports that are allocated dynamically are called *ephemeral*. They are usually above 1024. If you write your own client server application on a multiprogramming machine, you need to define your own server port number and code it into your application.

1. UDP: *message-oriented and unreliable*

- UDP delivers either the exact message (a.k.a. “datagram”) or nothing
- Each message ≤ 65535 bytes

Why?

- If a message is too large to fit into a single IP packet (i.e. larger than MTU), then IP layer *fragments* it:
  - not visible to the transport layer
  - at source’s IP layer, metadata about fragments is added to IP header
  - if a fragment/piece is lost, then the entire message is lost
- Application layer must handle *out of order* or *lost* messages, if needed
  - e.g. DNS client resends DNS query if no response is received

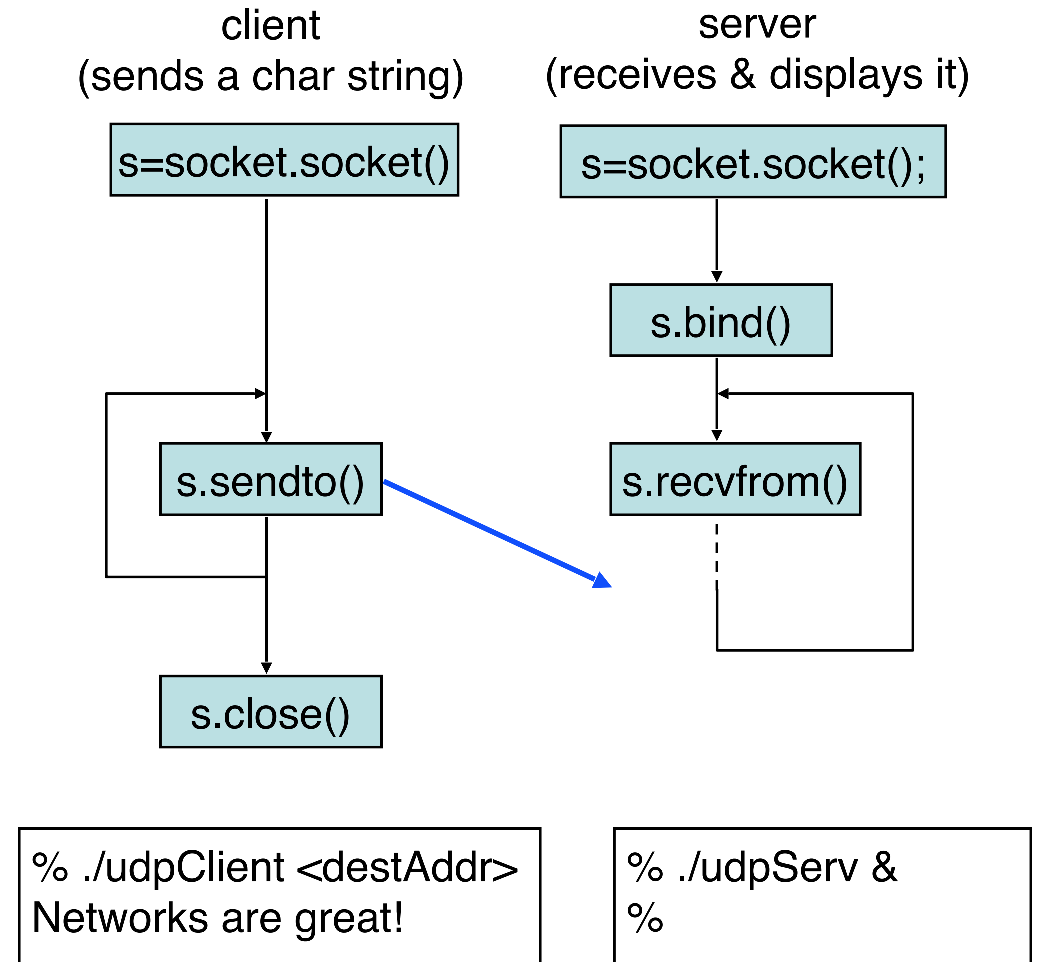


How is UDP implemented in practice?

Via *sockets* = programming interfaces

- (in Unix) for read/write: sockets ≈ files

- socket(socket.AF_INET,…), resp. socket(socket.AF_INET6,…) creates an IPv4 (resp. IPv6) socket, returns an ID (=file descriptor)
- bind() associates an IP address + port number with a socket, can be skipped for the client
  - *Port = 0* —> any available port,
  - *IP address = 0* (0.0.0.0 or ::) —> all host's IP addresses
- sendto() specifies: dest IP address,
  dest port number
  and the message to send
- recvfrom() blocks until a message is received for this port number; it returns: src IP address,
  src port number
  and the message

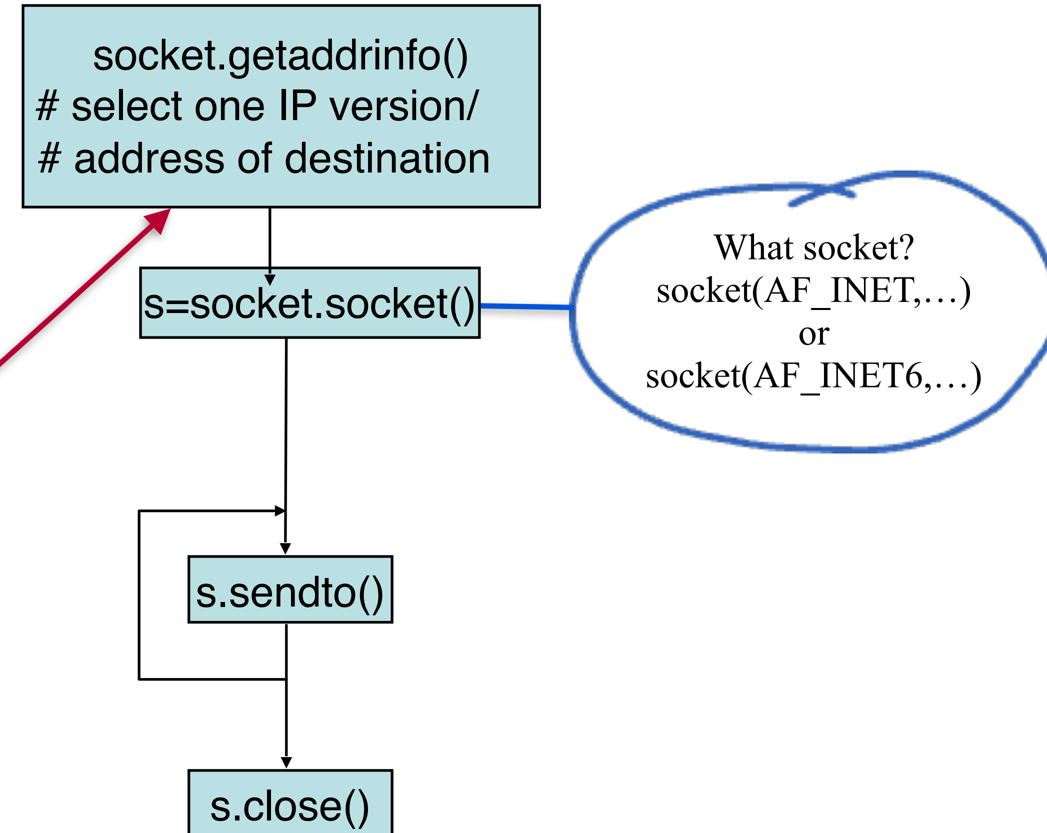


What socket to open? IPv4 or IPv6?

- Transport layer is not affected by the choice of IP (no UDPv6, nor TCPv6)
- But, an application must choose IPv4 or IPv6, or better support both

How? Use **socket.getaddrinfo()** to let the DNS give you whatever is available

```
> python
>>> import socket
>>> socket.getaddrinfo("nal.epfl.ch",None)
[(<AddressFamily.AF_INET6: 23>, 0, 0, '',
('2001:620:618:521:1:80b3:2127:1', 0, 0, 0)),
(<AddressFamily.AF_INET: 2>, 0, 0, '',
('128.179.33.39', 0))]
```



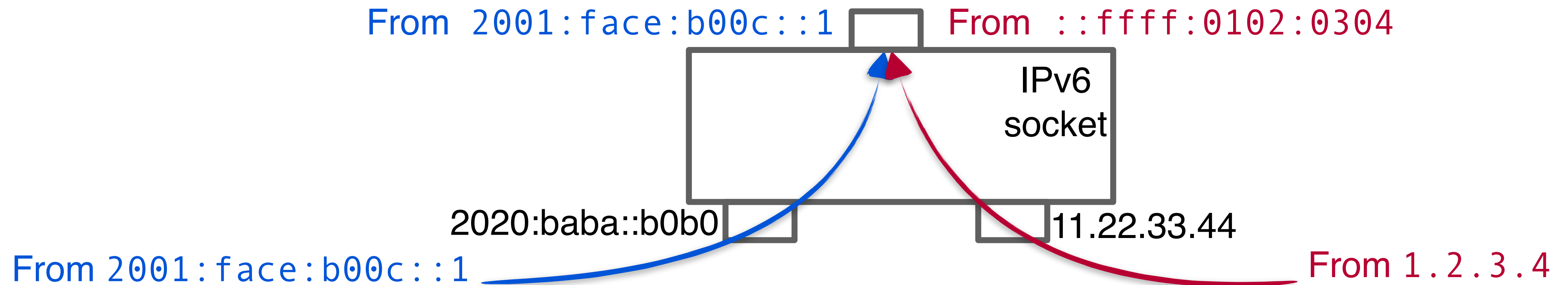
IPv6 sockets can be *dual-stack*

= bound to both IPv6 and IPv4 addresses of the local host

= receive packets from IPv6 and/or IPv4 correspondents

How? The correspondents' IPv4 addresses are mapped to IPv6 addresses

- using the *IPv4-mapped IPv6 address* format
- i.e., by appending the IPv4 address to prefix `::ffff:0:0/96`



- Default in Linux, must be enabled for every socket (with `setsockopt`) in Windows.
- An IPv4 socket *cannot* be dual-stack. Why?

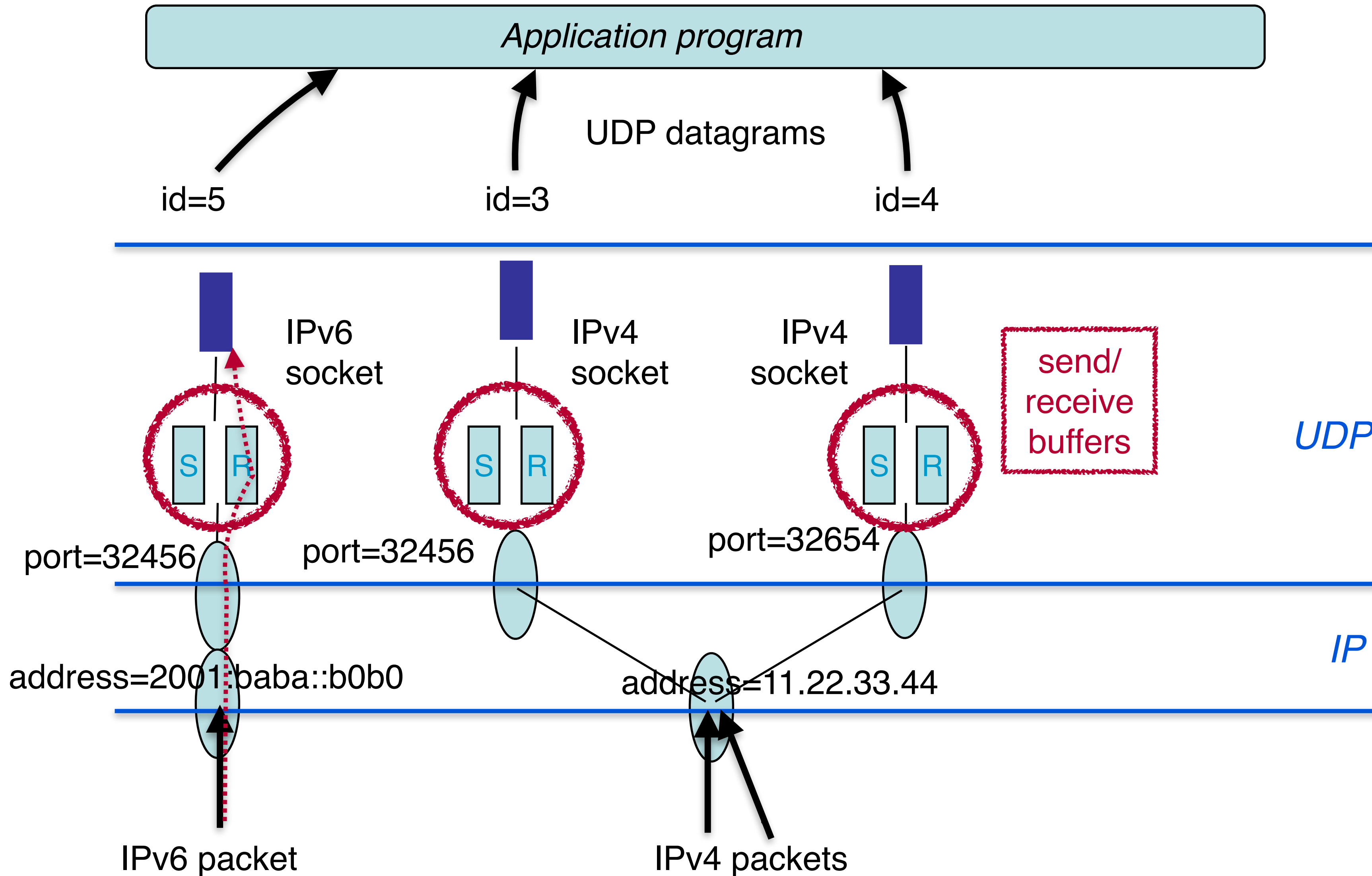
Solution

It is possible to map IPv4 addresses to a subset of the IPv6 space because IPv6 addresses are much longer in bits.

The converse is not possible: there are more IPv6 addresses than IPv4 addresses.

=> An IPv4 socket cannot receive data from an IPv6 source address.

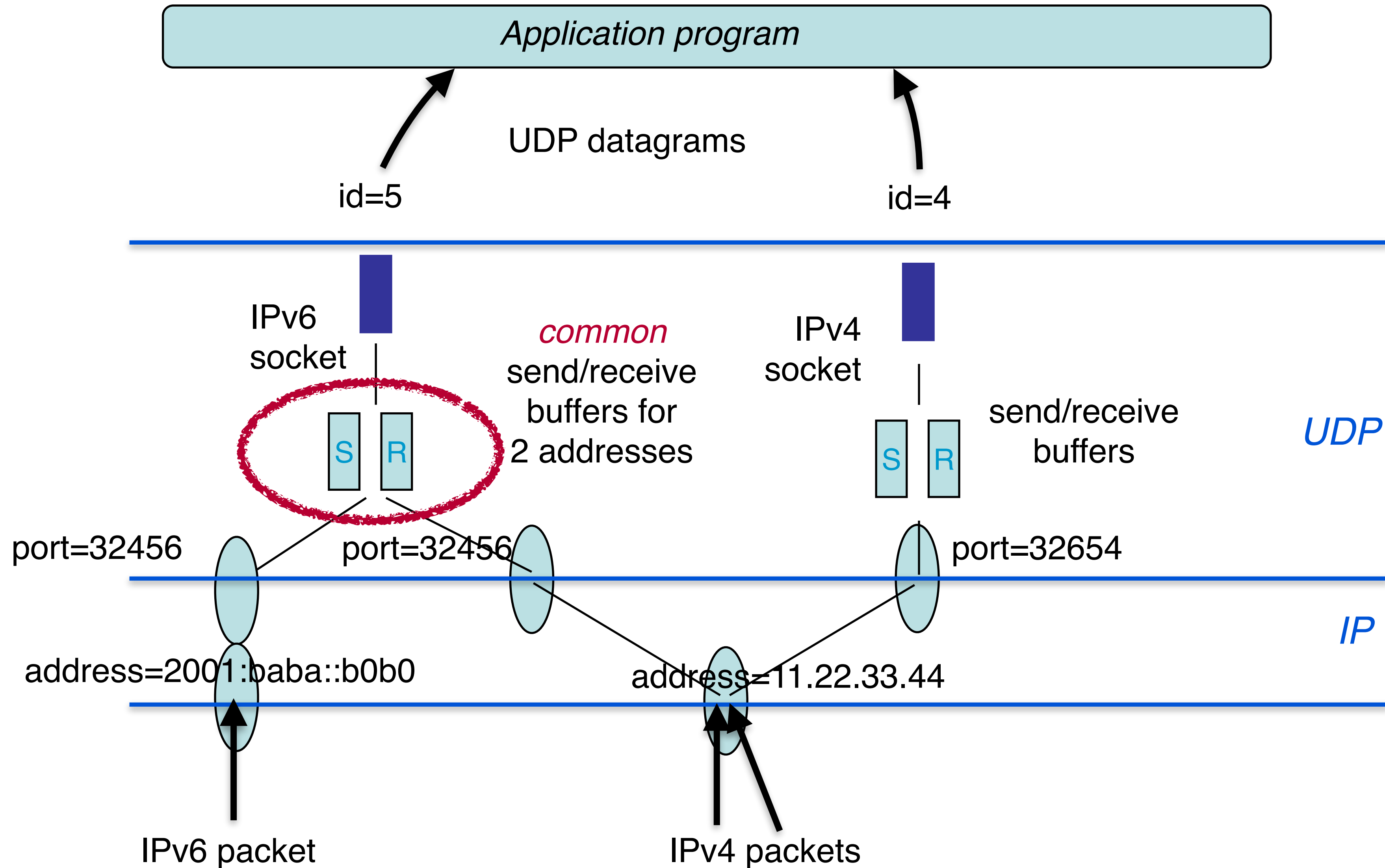
How does the Operating System view UDP?



UDP datagrams are delivered to sockets based on *dest IP address* and *port number*:

- Socket 5 is bound to local address 2001:baba::b0b0 and port 32456; receives all data to 2001:baba::b0b0 udp port 32456
- Socket 3 is bound to local address 11.22.33.44 and port 32456; receives all data to 11.22.33.44 udp port 32456
- Socket 4 is bound to local address 11.22.33.44 and port 32654; receives all data to 11.22.33.44 udp port 32654

With a dual-stack IPv6 socket?



Socket 5 is bound to any local address, which includes IPv6 and IPv4 addresses, and to port 32456; receives all packets to 2001:baba::b0b0, udp port 32456 and to 11.22.33.44 udp port 32456

Socket 4 is bound to IPv4 address 11.22.33.44 and port 32654; receives all packets to 11.22.33.44 udp port 32654

User's browser sends DNS query to DNS server, over UDP. What happens if query or answer is lost ?

A. Name resolver in browser waits for timeout, if no answer received before timeout, sends again
B. Messages cannot be lost because UDP assures message integrity
C. UDP detects the loss and retransmits
D. Je ne sais pas

Go to web.speakup.info or download speakup app

Join room 87072

Solution

Answer A

Recap - UDP

On the sending side:

OS sends the UDP message ASAP, but also uses a buffer to store data if interface is busy

OS may also fragment the message if needed.

On the receiving side:

OS *re-assembles* IP fragments of UDP message (if needed) and keeps the message in receive buffer ready to be read.
The message is

- "*consumed*" when application reads
- or "*dropped*" because of an overflow event

A socket is bound to a single port and one or multiple IP addresses of the local host

2. TCP offers reliable, in-order delivery

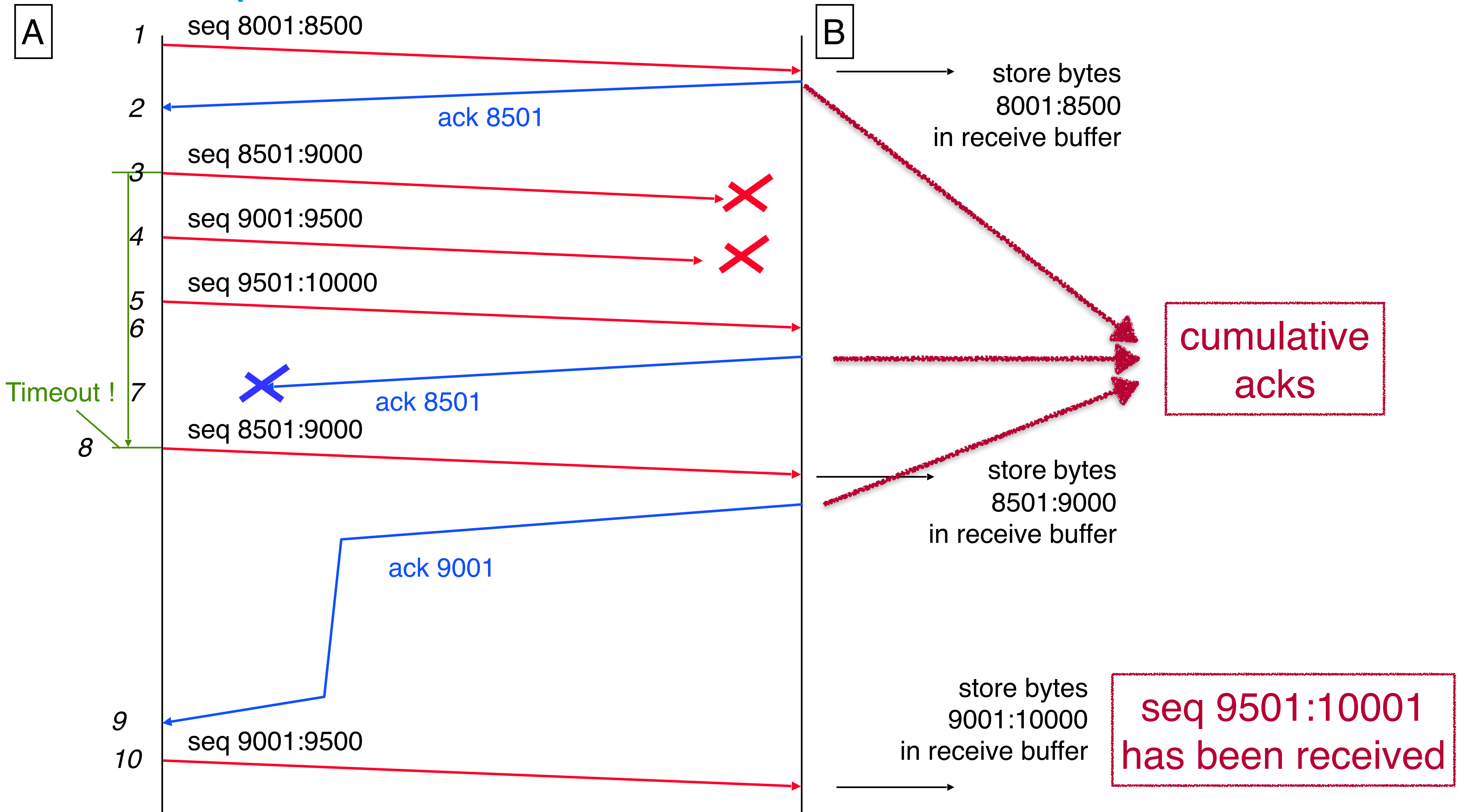
What?

TCP guarantees that all data is delivered *in order* and *without loss*, unless the connection is broken

How?

- *Detects* reordering and loss, via sophisticated mechanisms:
  - *per-byte* sequence numbers —> data is numbered
  - a connection-setup phase for the sender/receiver to *synchronize* their sequence nums
  - acknowledgements; if loss is detected, TCP retransmits
- further optimizations, e.g.:
  - *flow control* avoids buffer overflow at the receiver
  - TCP knows the allowable maximum segment size (MSS) and segments data accordingly —> avoids fragmentation at the IP layer

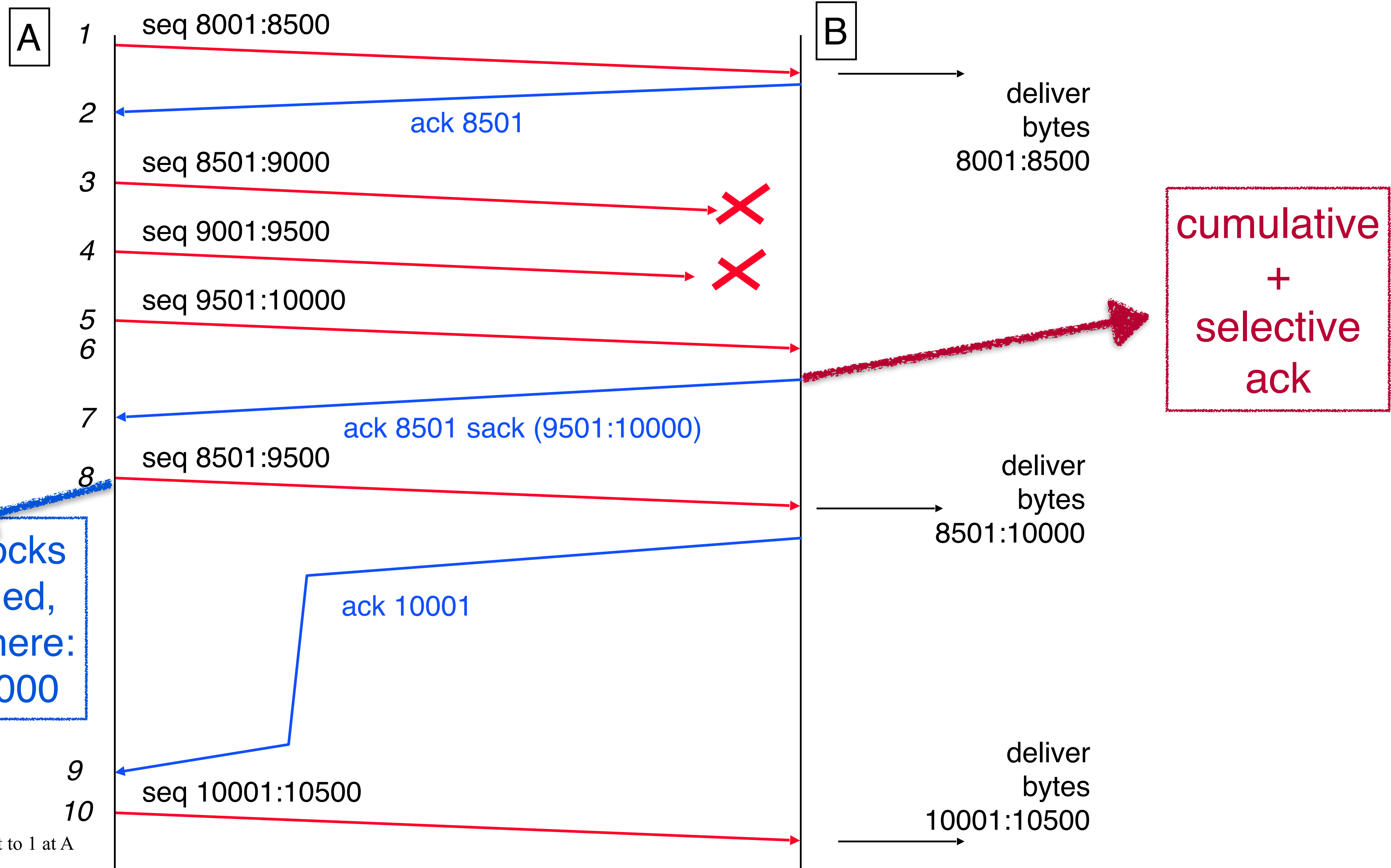
TCP Basic Operation 1: SEQ and ACK



The previous slide shows A in the role of sender and B of receiver.

- The application at A sends data in blocks of 500 bytes at a slow pace. So, TCP initially sends 500-byte segments.
- However, the maximum segment size in this example is 1000 bytes. So, TCP may also merge 2 blocks of data in one segment if this data happens to be available at the send buffer of the socket.
- Packets 3, 4 and 7 are lost.
- B returns an acknowledgement in the ACK field. The ACK field is *cumulative*, so ACK 8501 means: B is acknowledging all bytes up to (excluding) number 8501. I.e. the ACK field refers to the next byte expected from the other side.
- At line 8, the timer that was set at line 3 expires (A has not received any acknowledgement for the bytes in the packet sent at line 3 and experiences a *timeout*). A re-sends data that is detected as lost, i.e. bytes 8501:9001. When receiving packet 8, B delivers all bytes from 8501 to 9000 in order.
- When receiving packet 10, B can deliver bytes 9001:10000 because packet 5 was received and kept by B in the receive buffer.
- The out-of-order received data is kept in the buffer until all data is received correctly— this is not visible to the application.

TCP Basic Operation 2: SACK and optimized segmentation (if possible)



In addition to the ACK field, most TCP implementations also use the SACK field (*Selective Acknowledgement*).

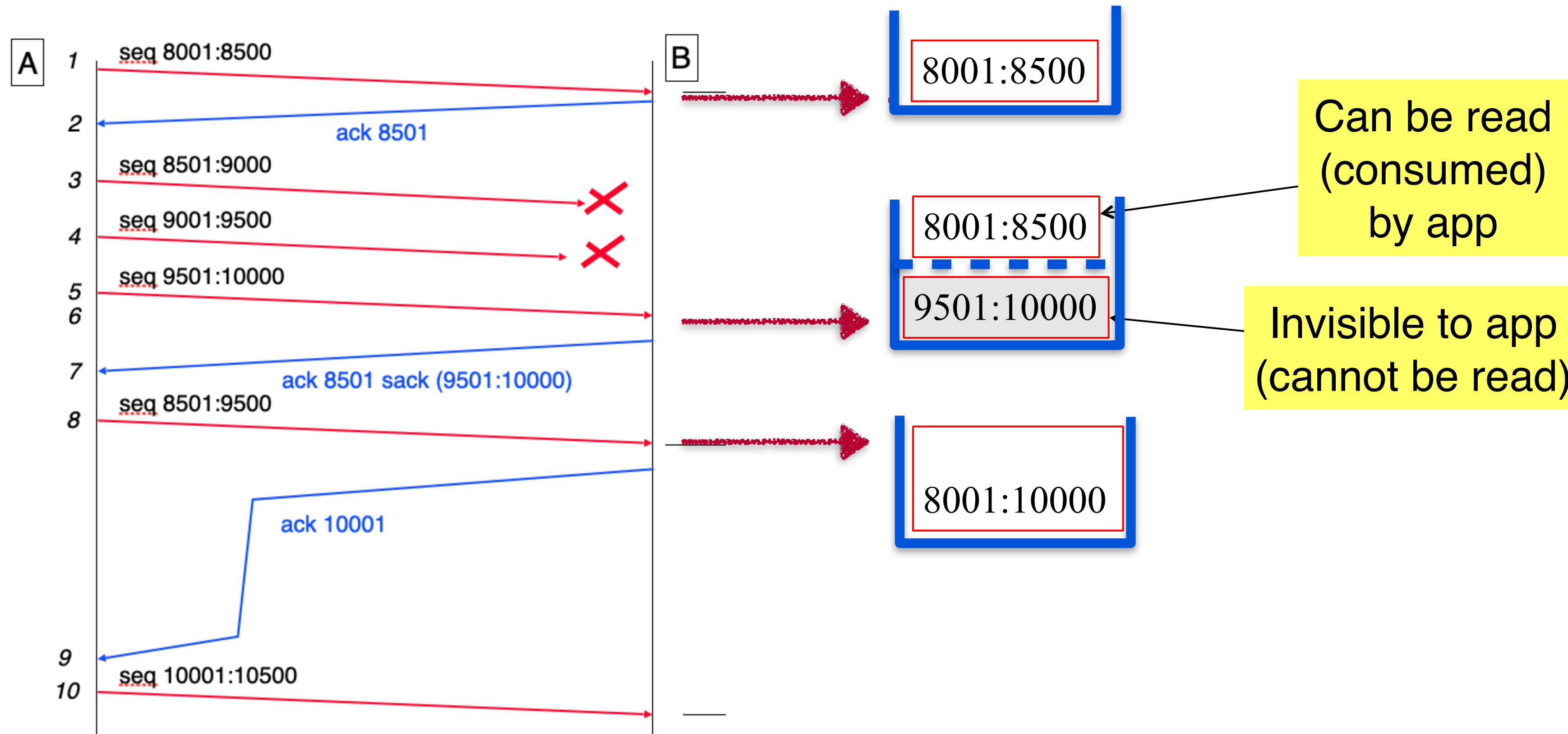
The previous slide shows the operation of TCP with SACK.

- The application at A sends data in blocks of 500 bytes. But, in this example, we assume that the maximum segment size is MSS=1000 bytes.
- Segments 3 and 4 are lost.
- At line 6, B acknowledges all bytes up to (excluding) number 8501.
- At line 7, B acknowledges all bytes up to 8501 and in the range 9501:10000. Since the set of acknowledged bytes is not contiguous, the SACK option is used. It contains up to 3 blocks that are acknowledged in addition to the range described by the ACK field.
- At line 8, A detects that the bytes 8501:9500 were lost and re-sends them ASAP without waiting for a timeout, because in this example host A uses TcpMaxDupACKs = 1 (we will discuss TcpMaxDupACKs later). What is important to notice is that at line 8, since the maximum segment size is 1000 bytes, only one packet is sent. This is what the slide's title means by “optimized segmentation”.
- When receiving packet 8, B can deliver bytes 9001:10000 because packet 5 was received and kept in the receive buffer.

TCP receiver uses a *receive buffer* = *re-sequencing buffer* to store incoming packets before delivering them to application

Why invented ?

- Application may not be ready to consume/read data
- Packets may need re-sequencing; out-of-order data is stored but is *not visible* to application



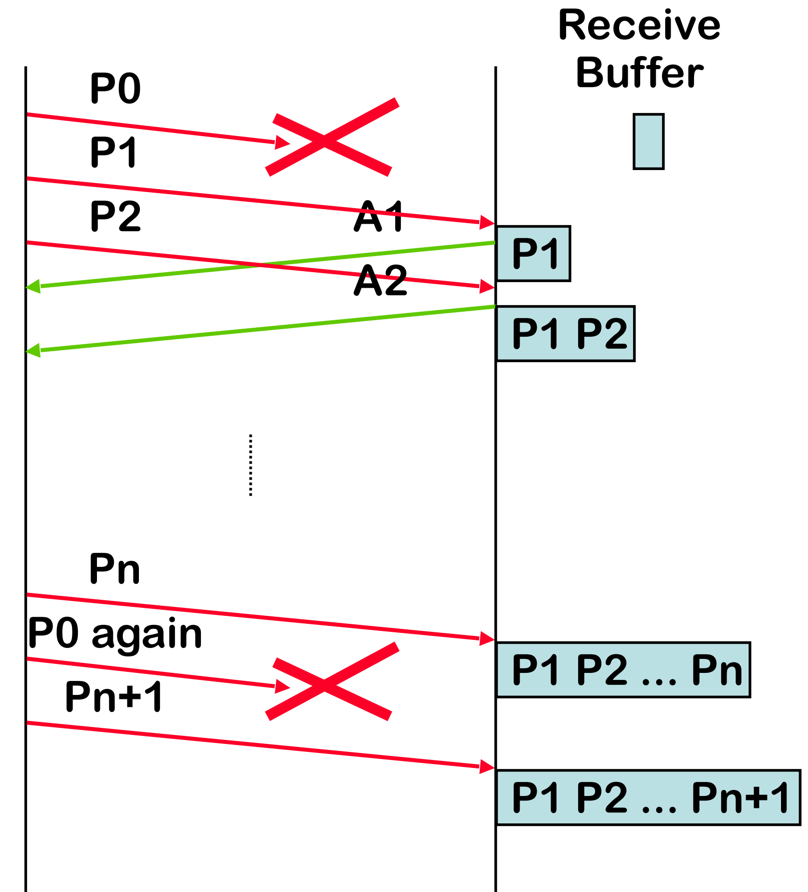
TCP uses a sliding window

Why?

Out-of-order packets may trigger buffer *overflows*

- e.g. what if a single segment “hangs”?

- The sliding window *limits* the number of data “on the fly” (= not yet acknowledged)



How does the sliding window work?

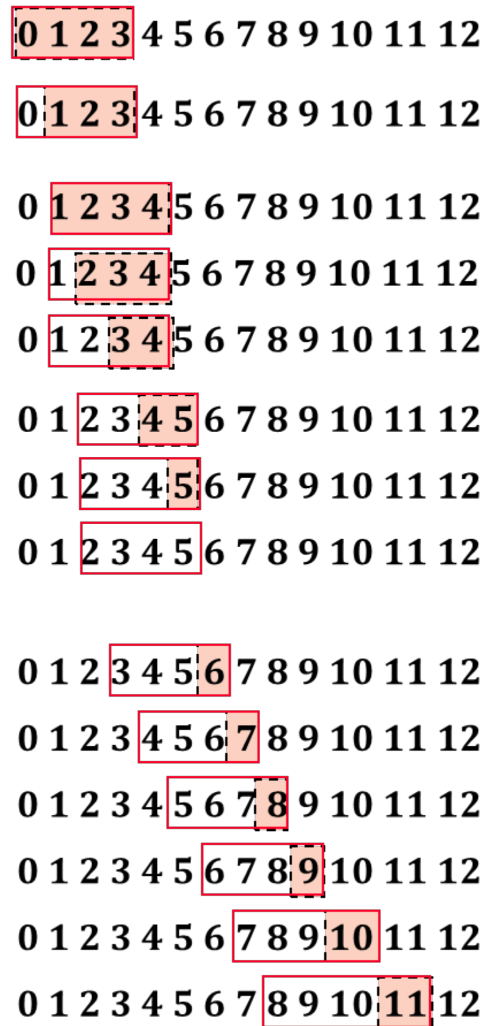
- Suppose:

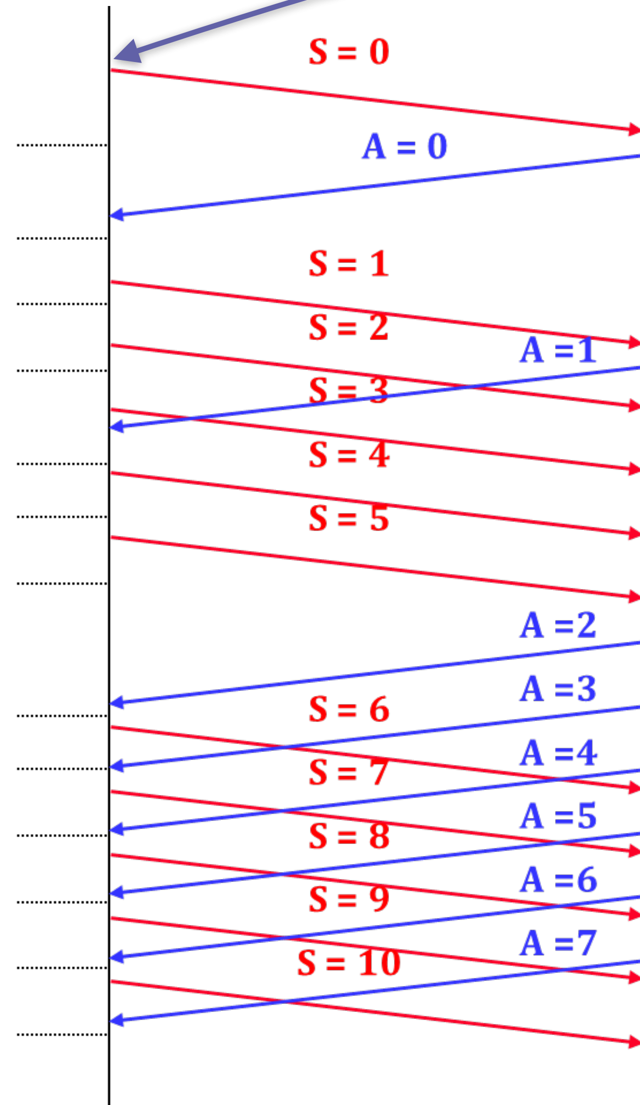
  Window size = 4000B;
  each segment =1000B

- Only seq numbers within the window can be sent (and are sent if enough data exist in the socket)

  lower window edge = smallest non-ack'ed sequence number

  upper window edge = lower_edge + window_size





Note: Assume that no more data exists in the socket; only segment 0 is sent, even though segments 0, 1, 2, 3 could be sent, too.

Window

usable part of the window, seq numbers that the sender can send