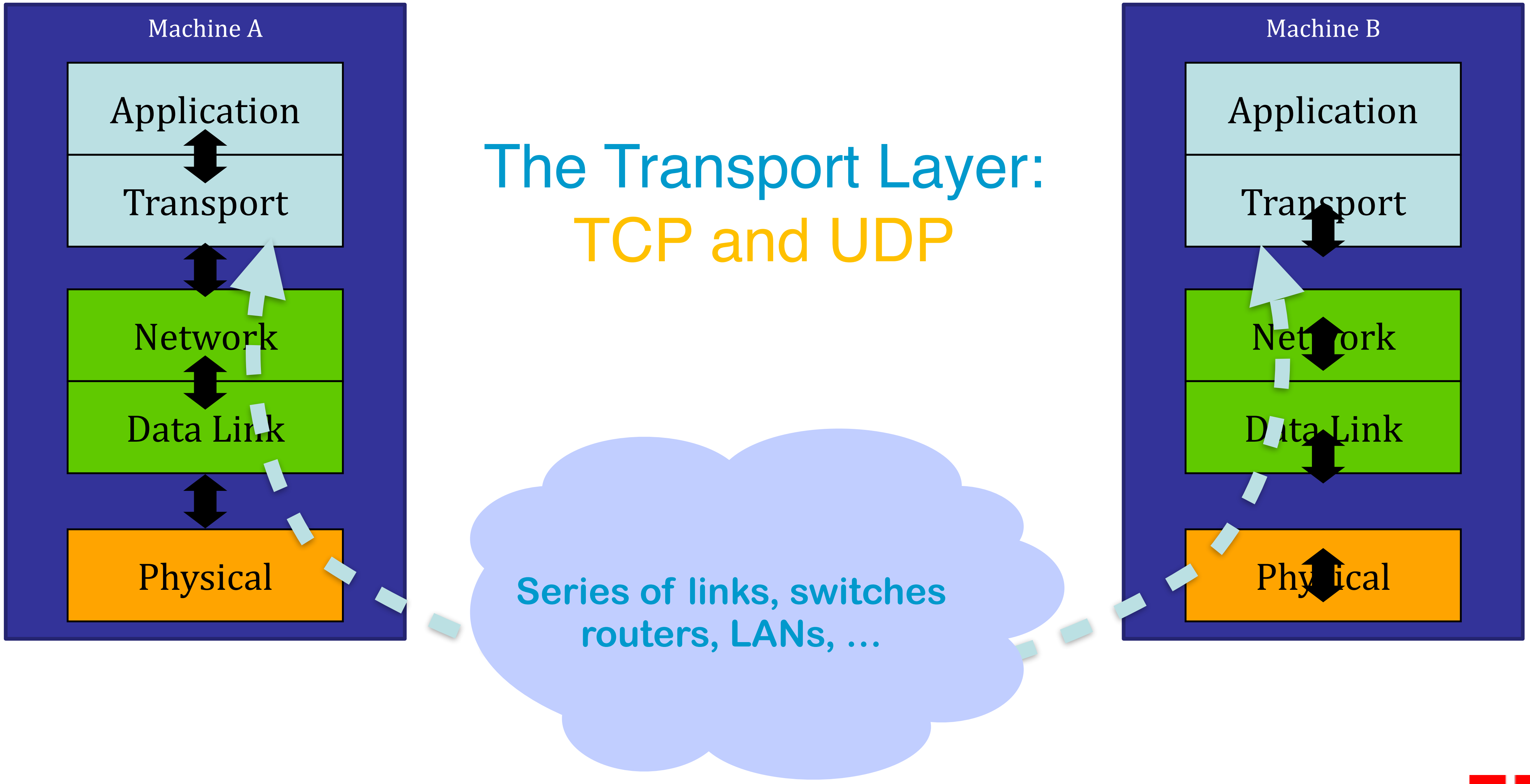
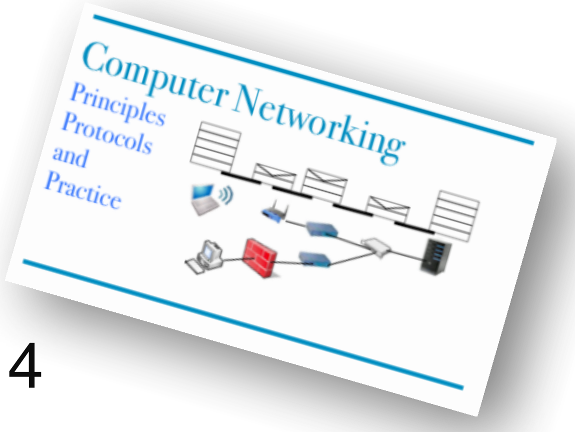
The Transport Layer: TCP and UDP

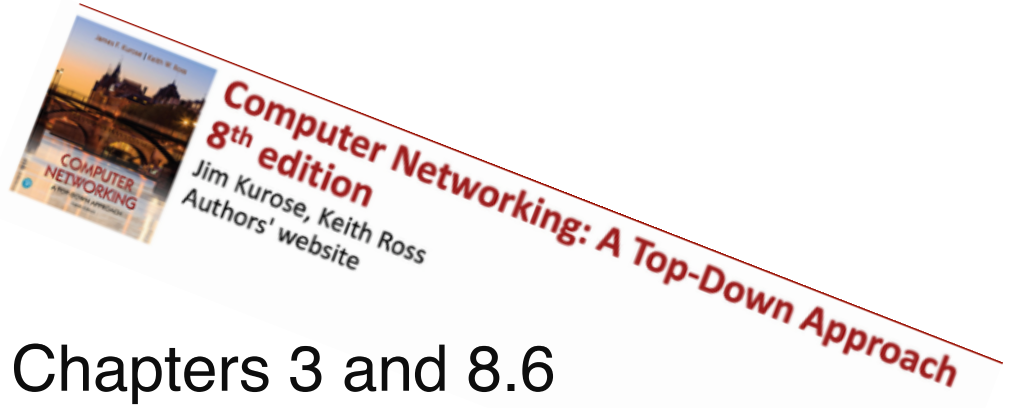


Contents

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* because of:
  - errors at the physical layer
  - buffer overflow events at routers or switches
- or *reordered* because they may follow 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 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

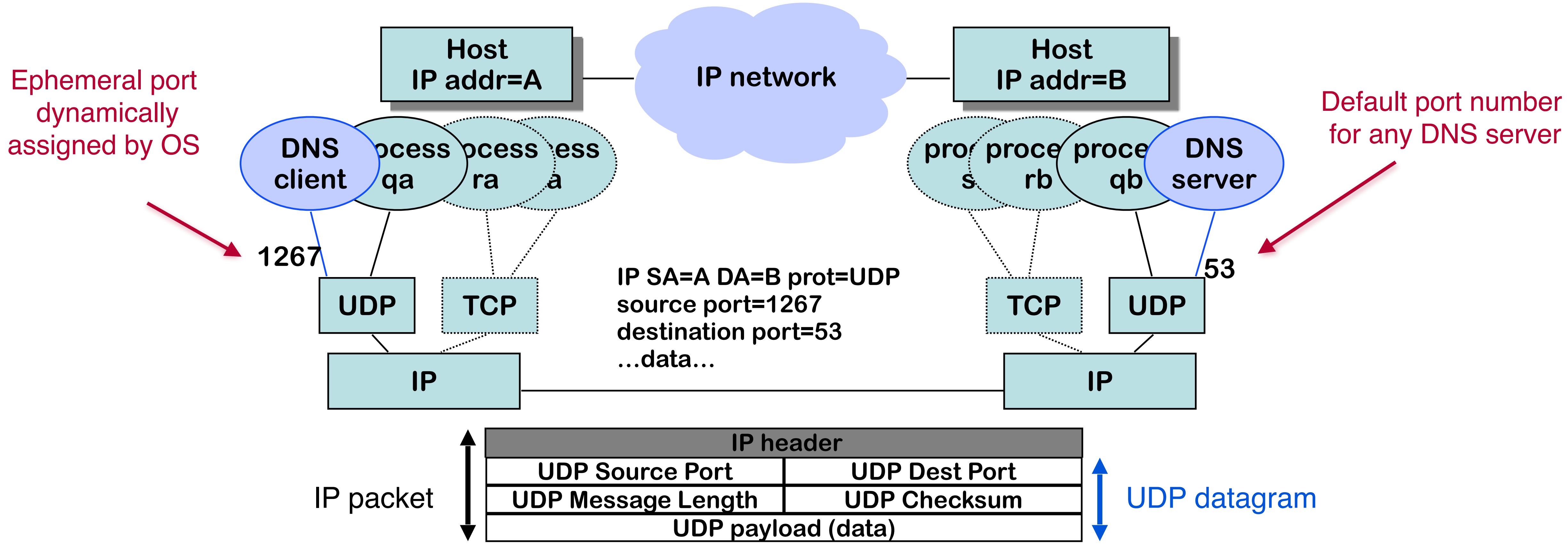
Join room 46045

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)
- 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 at host A, to the domain name server (DNS) 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 is *message-oriented,* and *unreliable*

- UDP delivers the exact message (a.k.a. “datagram”) or nothing
- Consecutive messages may arrive *out of order*
- Messages may be *lost*

application layer should handle these, if necessary

- One message, up to 65,535 bytes
- If UDP message is too large to fit into a single IP packet (i.e. larger than MTU), then IP layer *fragments* it
  - at the IP layer of the source, info about fragments added inside IP header
  - not visible to the transport layer
  - if a fragment/piece is lost then the entire message is considered lost

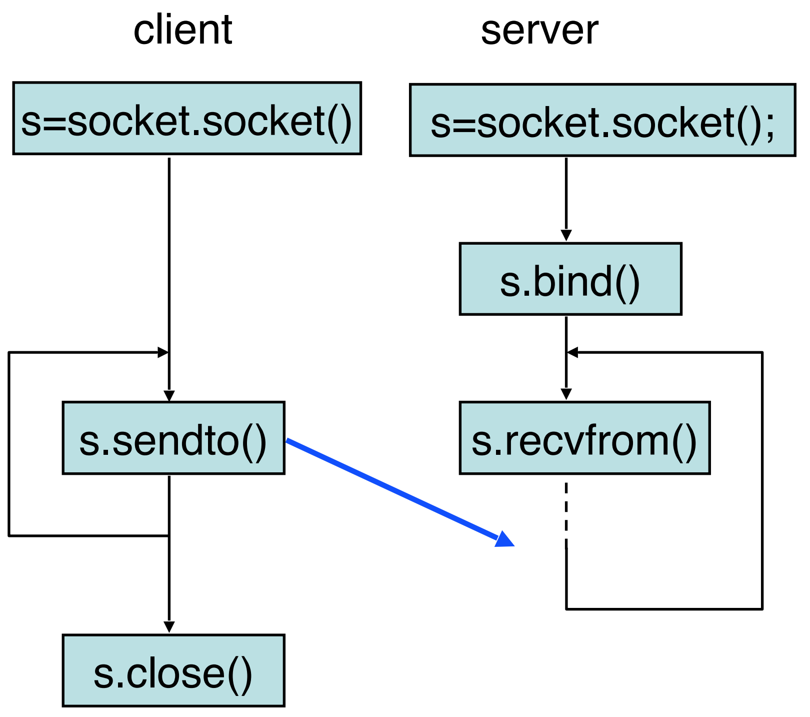
How is UDP implemented in practice?

Via a *socket library* = programming interface

- sockets in Unix are similar to files for read/write

Figure shows a client-server app using UDP:
client sends a char string to server,
which receives (and displays) it

- socket(socket.AF_INET,…) creates an IPv4 socket and returns a number (=file descriptor) if successful
- socket(socket.AF_INET6,…) creates an IPv6 socket
- bind() associates an IP address and port number with the socket—can be skipped for a client socket.
  Port = 0 means any available port,
  IP address = 0 (0.0.0.0 or ::) means all addresses of host
- sendto() specifies destination IP address, destination port number and the message to send
- recvfrom() blocks until a message is received for this port number; it returns the source IP address and port number and the message.



```
% ./udpClient <destAddr> bonjour les amis
%
```

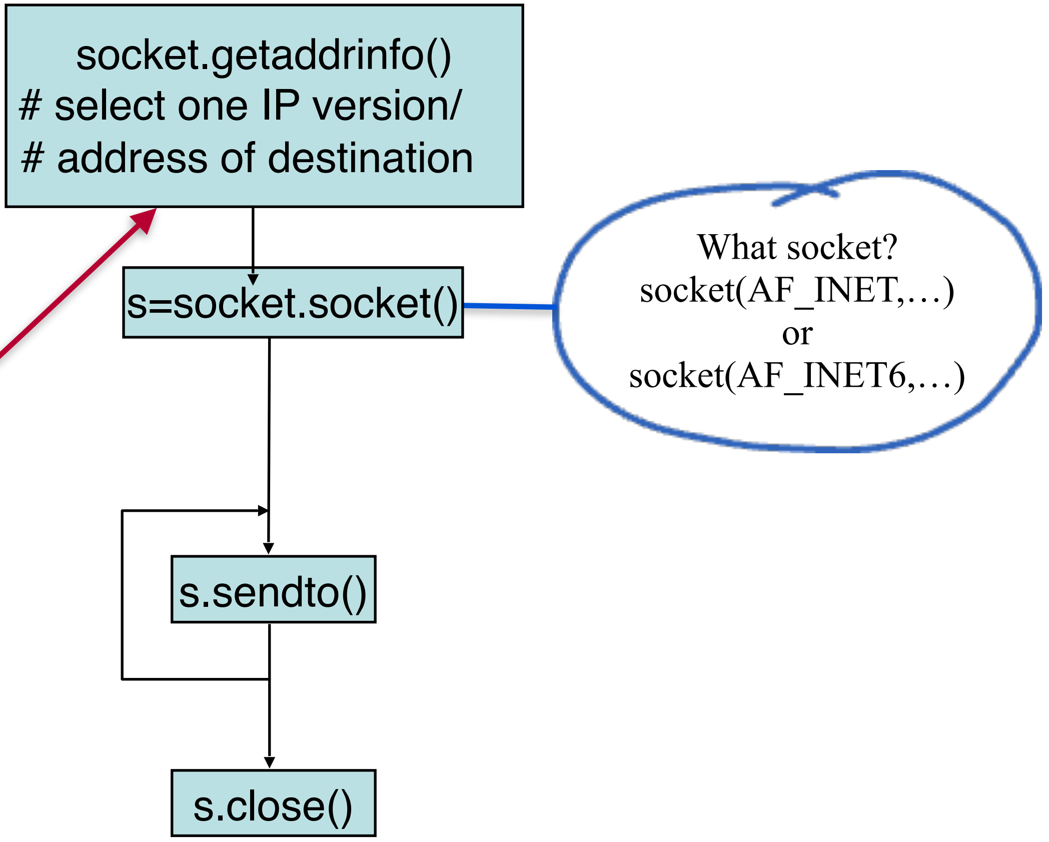
```
% ./udpServ &
%
```

What socket to open? IPv4 or IPv6?

- Transport layer is not affected by the choice of IP (no UDPv6, nor TCPv6)
- But, there are IPv4 and IPv6 sockets
- An application program has to choose IPv4 or IPv6, or better support both

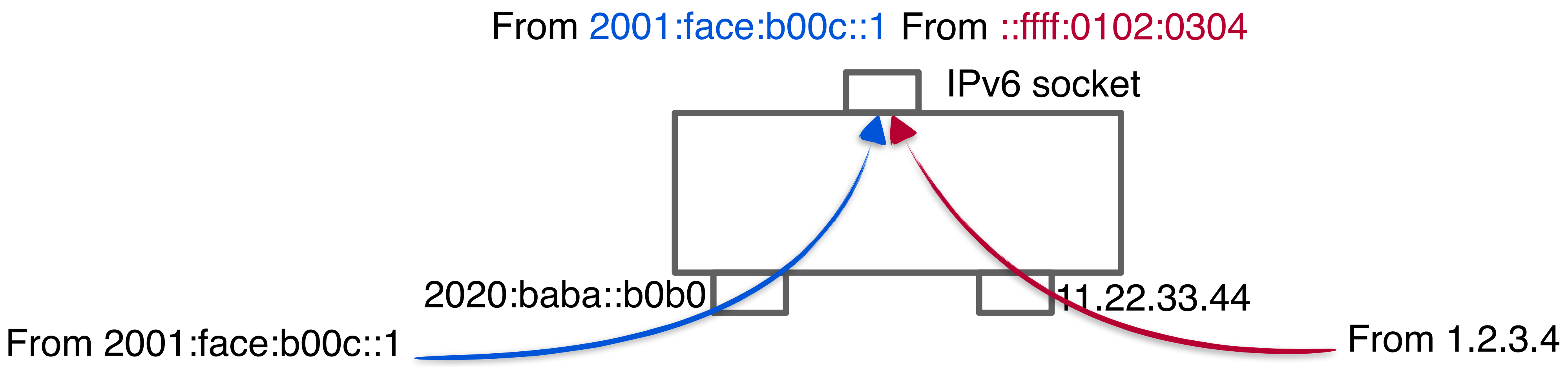
How? Use **socket.getaddrinfo()** to let the DNS give you whatever is available

```
> python
>>> import socket
>>> socket.getaddrinfo("lca.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))]
```



An IPv6 socket can be *dual-stack*

- In some machines, IPv6 sockets can be bound to both IPv6 and IPv4 addresses of the local host
- 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
- Such sockets can receive packets from IPv6 and from IPv4 correspondents.



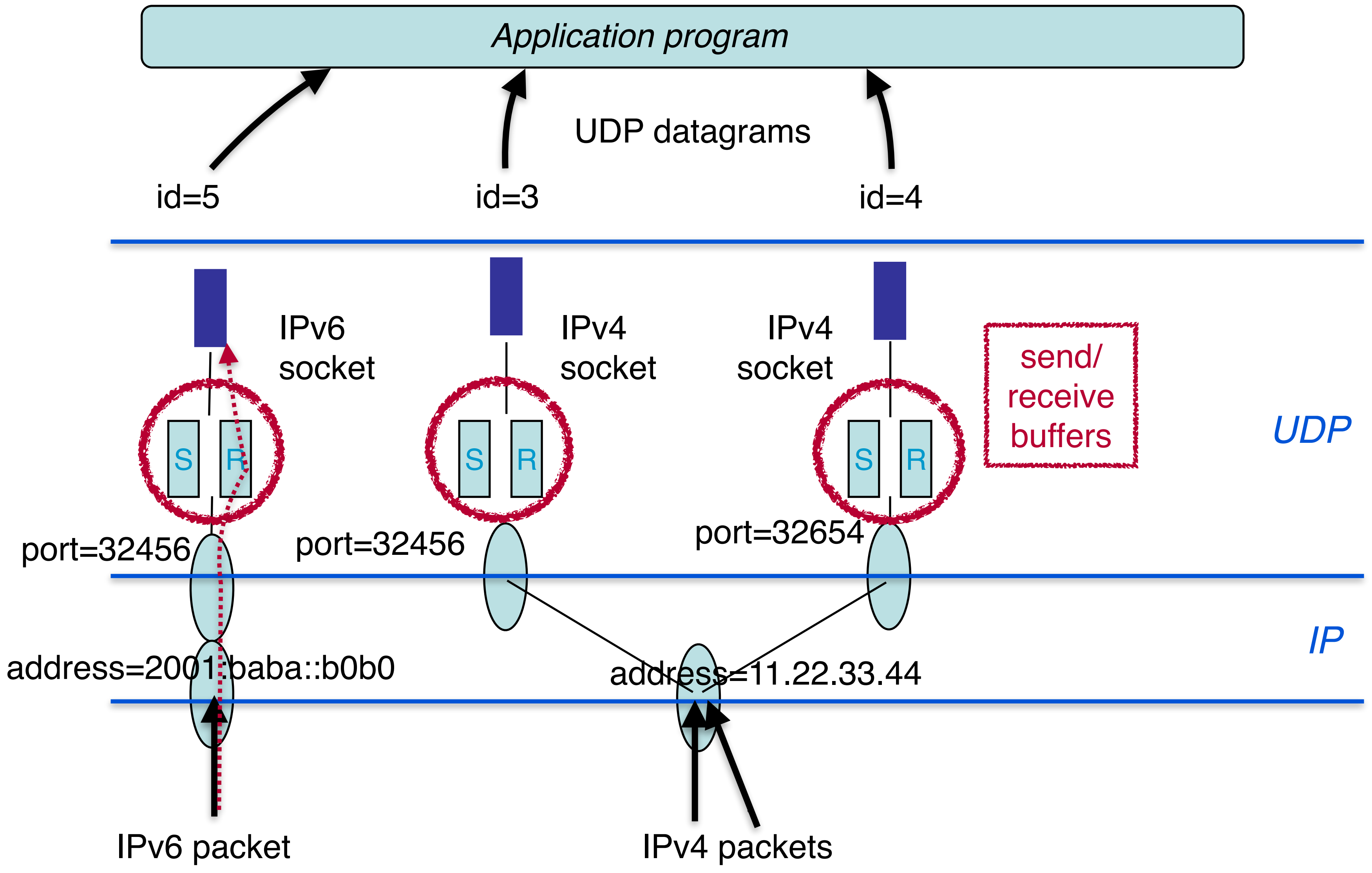
- Default in Linux, must be enabled for every socket (with `setsockopt`) in Windows.
- An IPv4 socket *cannot* be dual-stack. Why?

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.

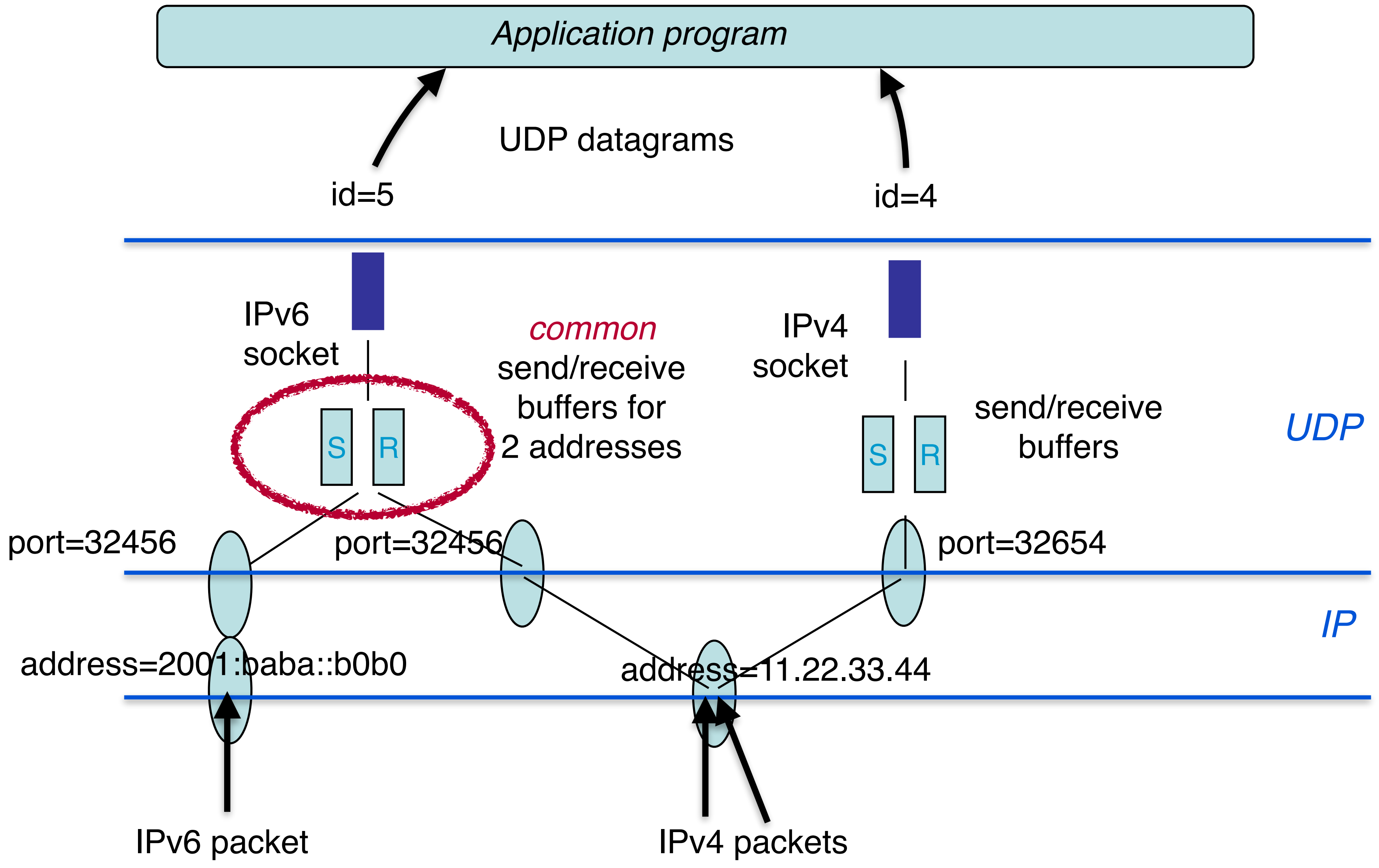
How does the Operating System view UDP?



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

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.

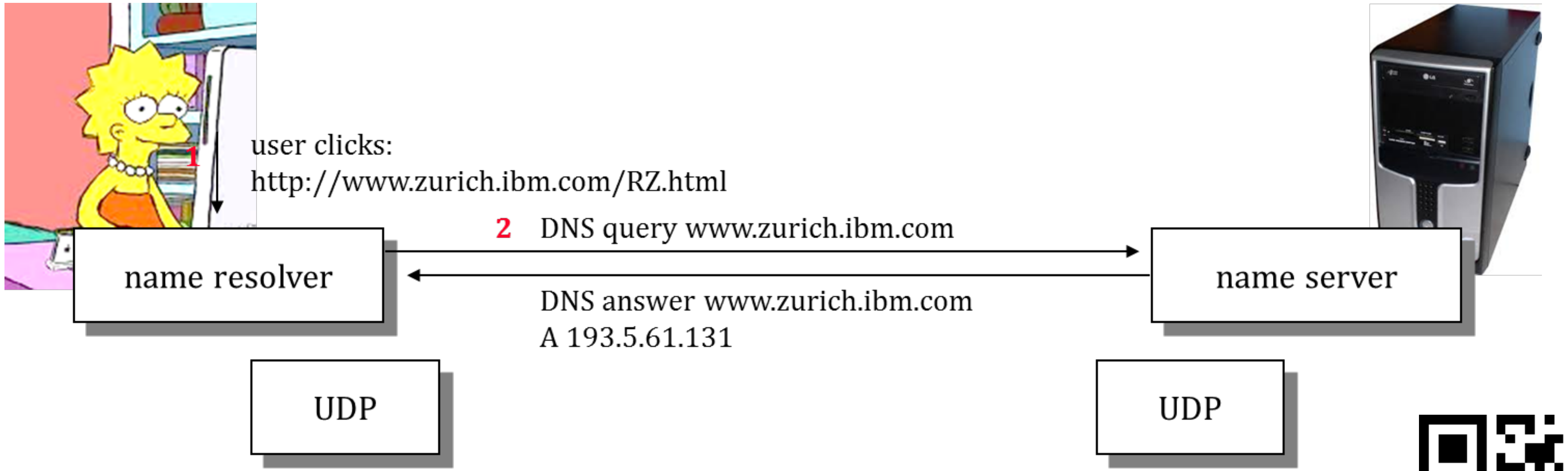
On the receiving side:

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

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


1
user clicks:
http://www.zurich.ibm.com/RZ.html
2 DNS query www.zurich.ibm.com
name resolver
name server
DNS answer www.zurich.ibm.com
A 193.5.61.131
UDP
UDP

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 46045

Solution

Answer A

2. TCP offers reliable, in-order delivery

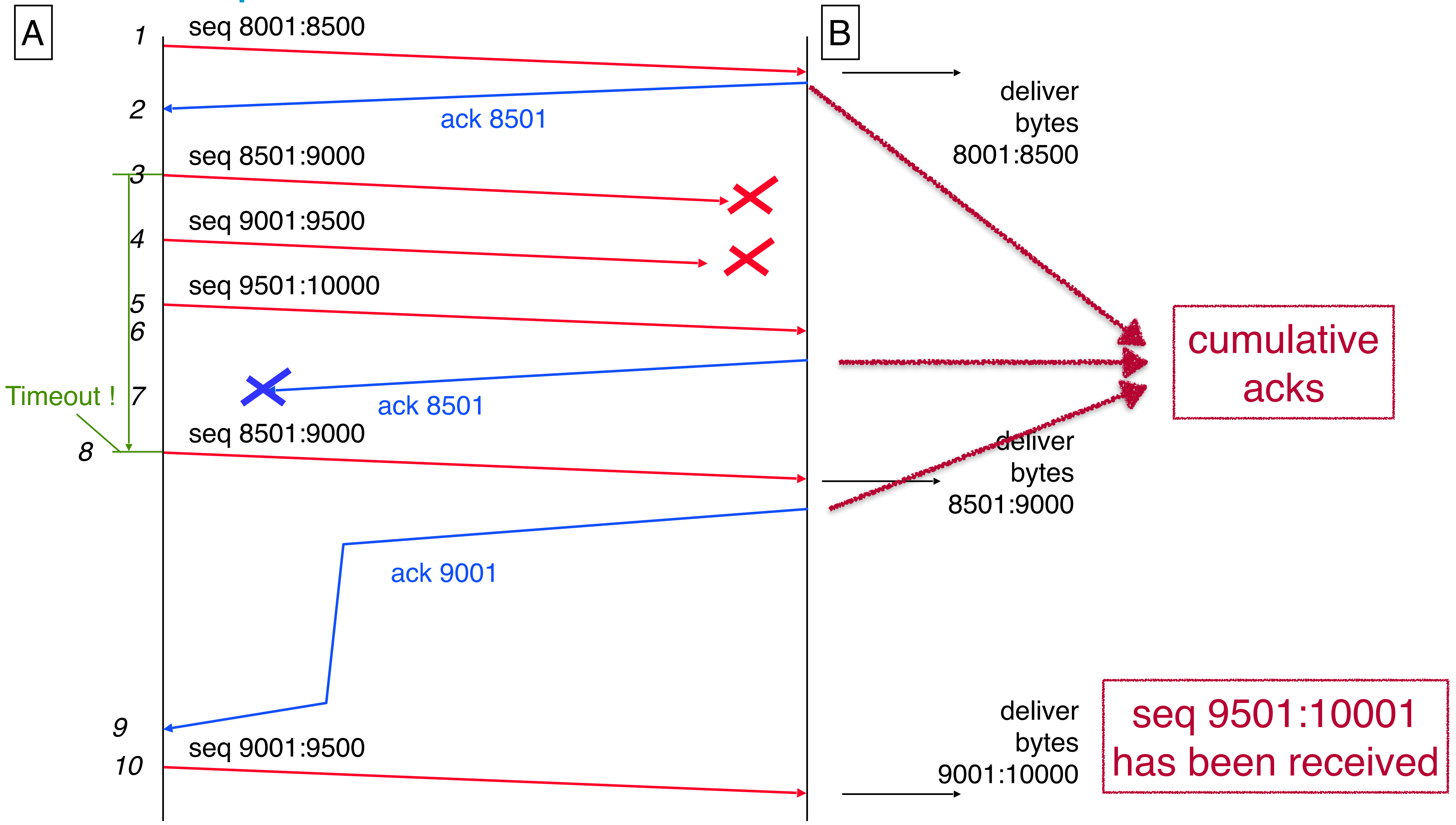
What does this mean?

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

How does TCP achieve this?

- Uses sophisticated mechanisms to *detect* reordering and loss:
  - *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 re-transmits
- 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

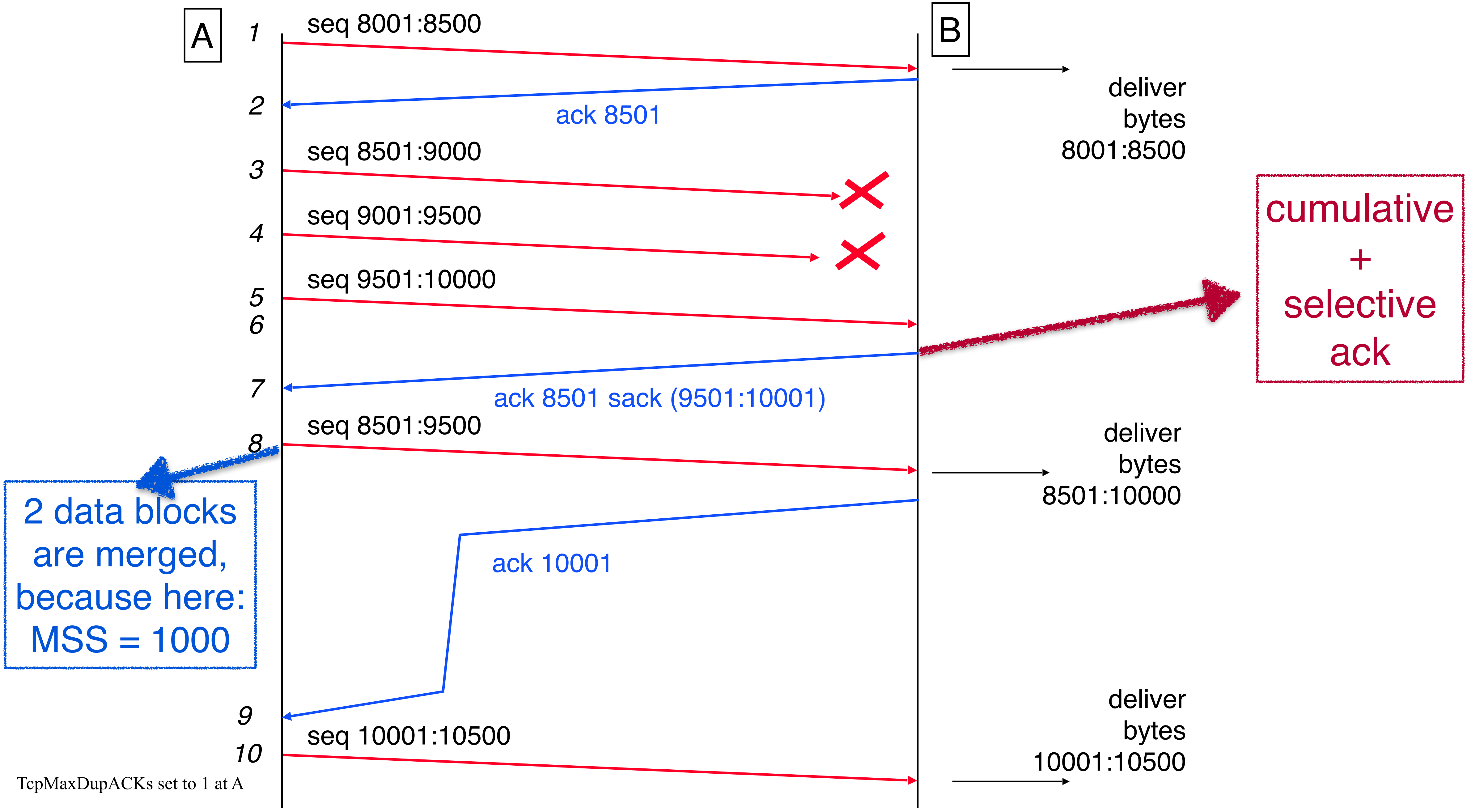
TCP Basic Operation 1: SEQ and ACK



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.

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



TcpMaxDupACKs set to 1 at A

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

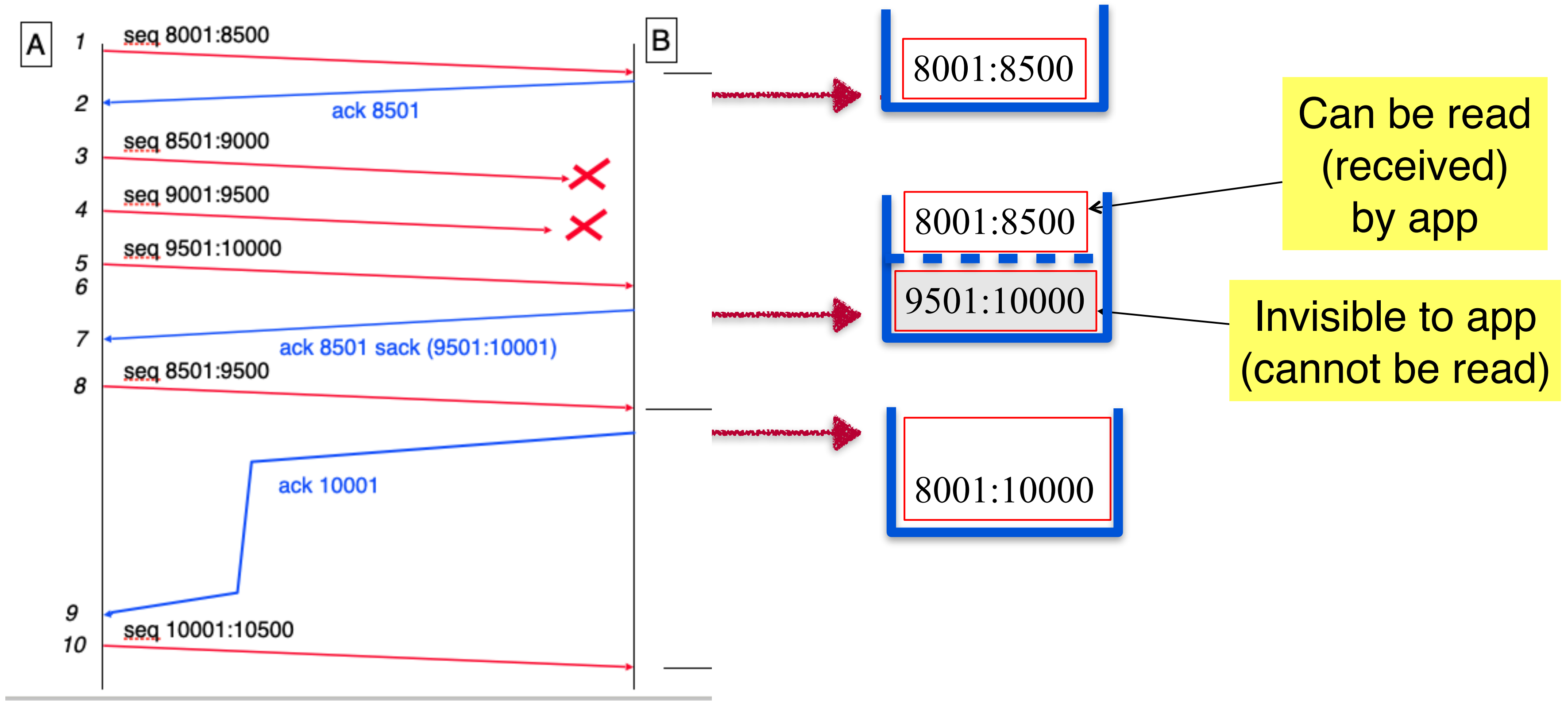
The previous slide shows the operation of TCP with SACK.

- 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 9501:10001. 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 8501:9501 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 9001:10001 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

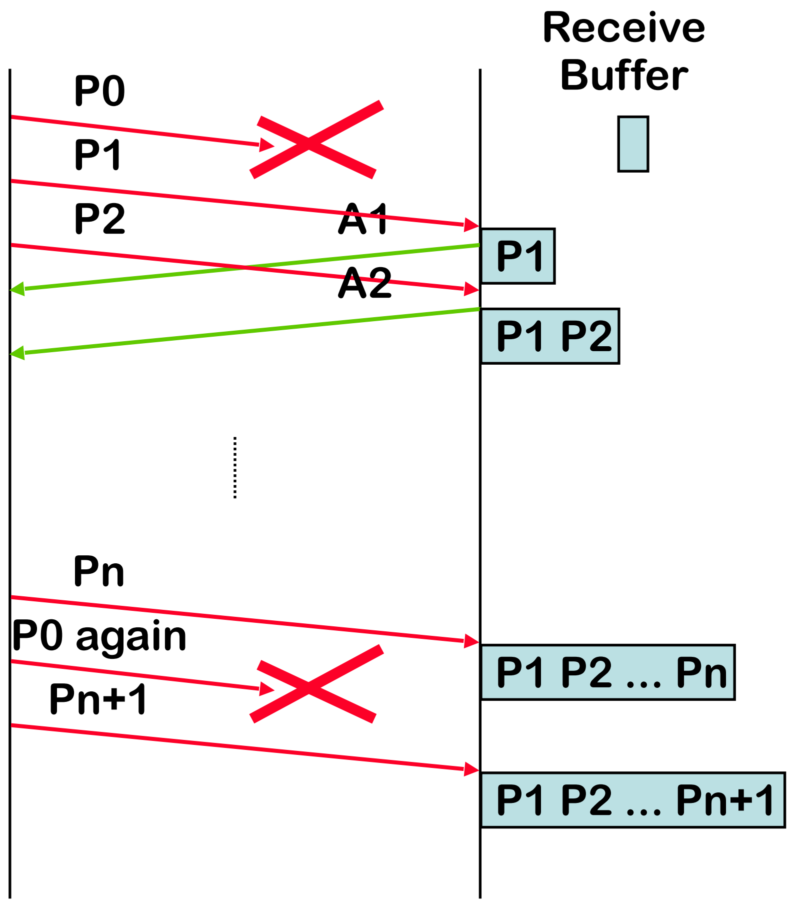


TCP uses a sliding window

Why?

- The receive buffer may overflow if one piece of data “hangs”
  - multiple losses affect the same packet,
  - so, multiple out-of-order packets fill the buffer

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



How does the sliding window work?

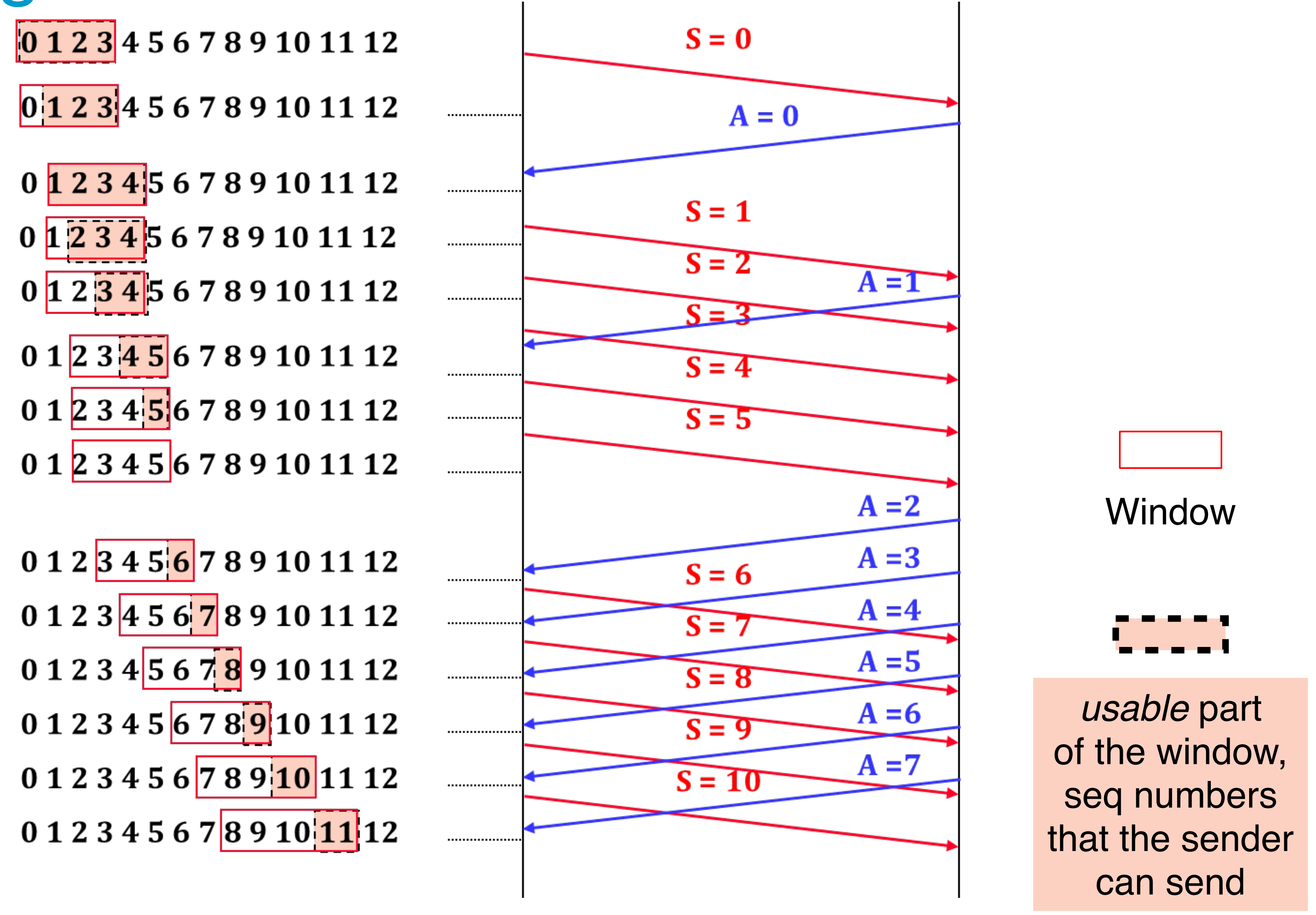
Suppose:

Window size = 4000B;
each segment =1000B

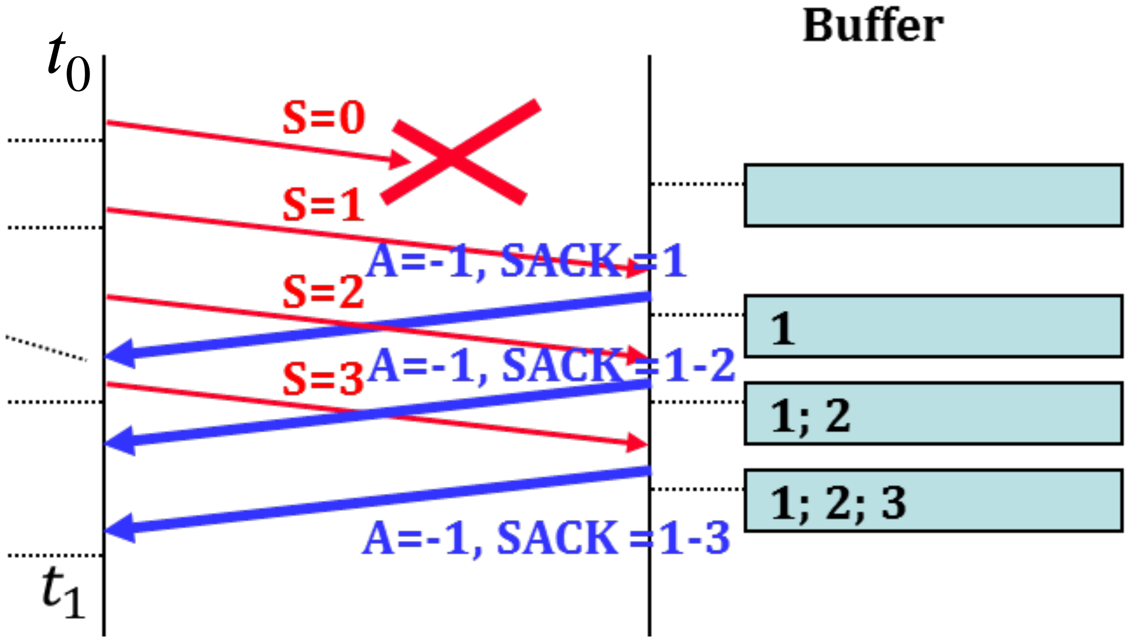
Only seq numbers that are in the window can be sent
(if of course this data is in the socket waiting to be sent)

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

upper window edge =
lower_edge +
window_size
```



Window size = 4’000 bytes, one packet = 1’000 bytes
At time t_0, the usable part of the sliding window is 4000, and there is a lot of data in the socket to be sent.
So, the sender sends 4 segments.
At time t_1, sender…

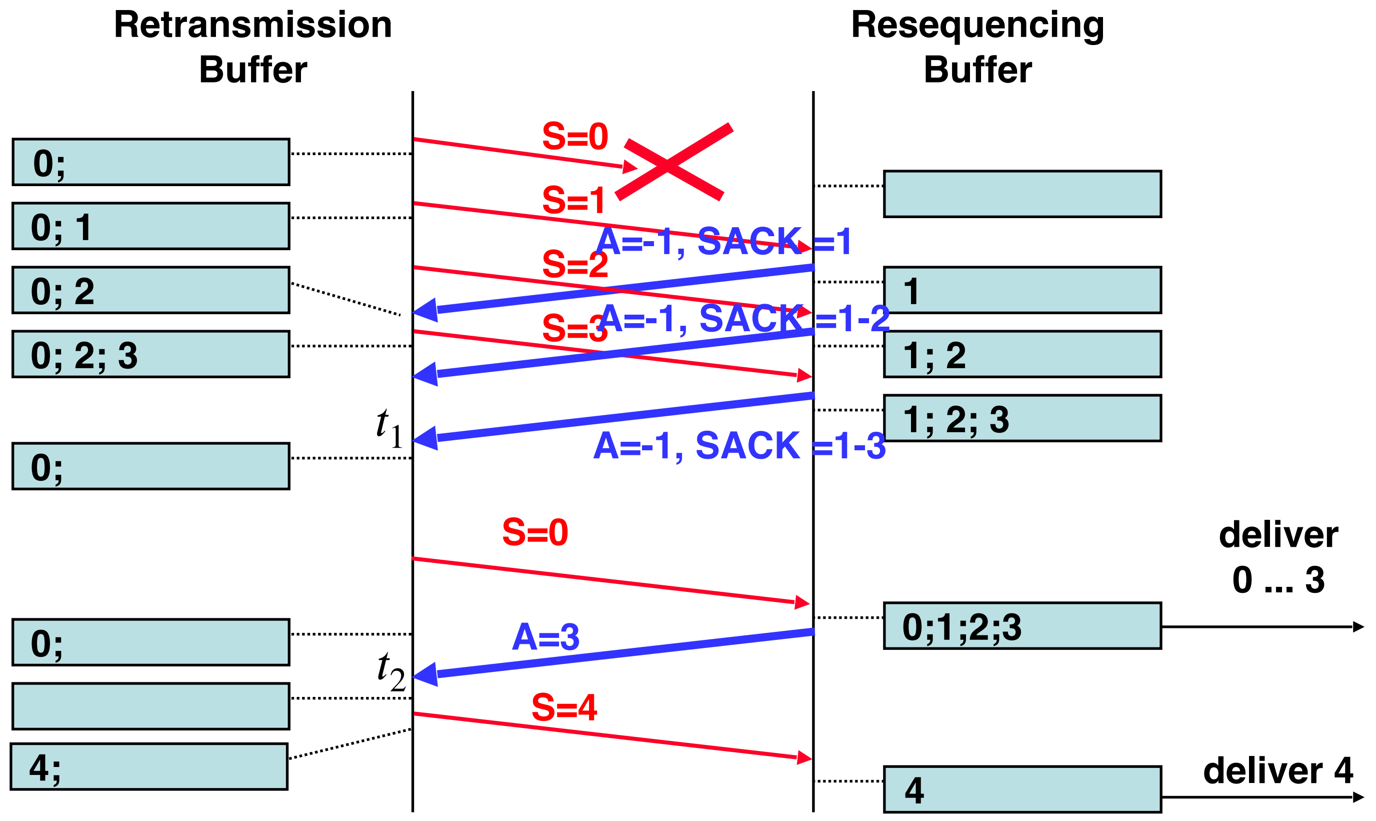


A. … can send segment S=4
B. … cannot send segment S=4
C. It depends on whether data was consumed by application
D. I do not know

Go to web.speakup.info or download speakup app

Join room 46045

Solution



Answer B.

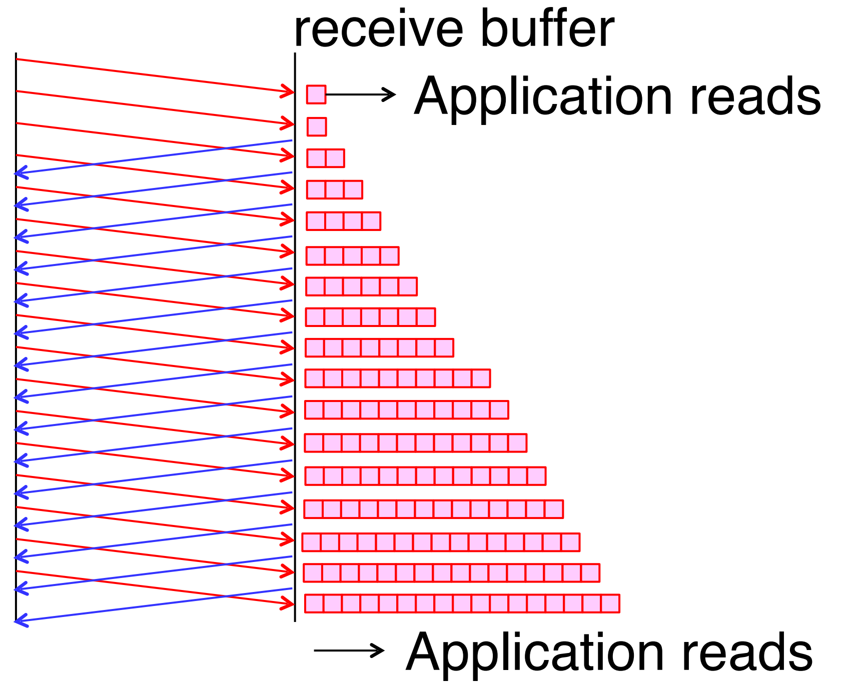
The window size is 4’000 B, namely here 4 packets.
At time t_1 packets -1, 1, 2 and 3 are acked. The window is packets $[0 ; 3]$. Packet 4 is outside the window and cannot be sent. It has to wait until the loss of packet 0 is repaired (at time t_2)

Sender also needs a buffer (“retransmission buffer”); its size is the window size.
Segments are removed from the resequencing/receive buffer when they are finally in-order and application reads them.

A fixed-size window cannot prevent receive-buffer *overflow*

- In-order data still remains in receive buffer, until it is consumed by application (typically using a socket "read" or "receive")

- *Slowly reading* receiver app could cause buffer overflow



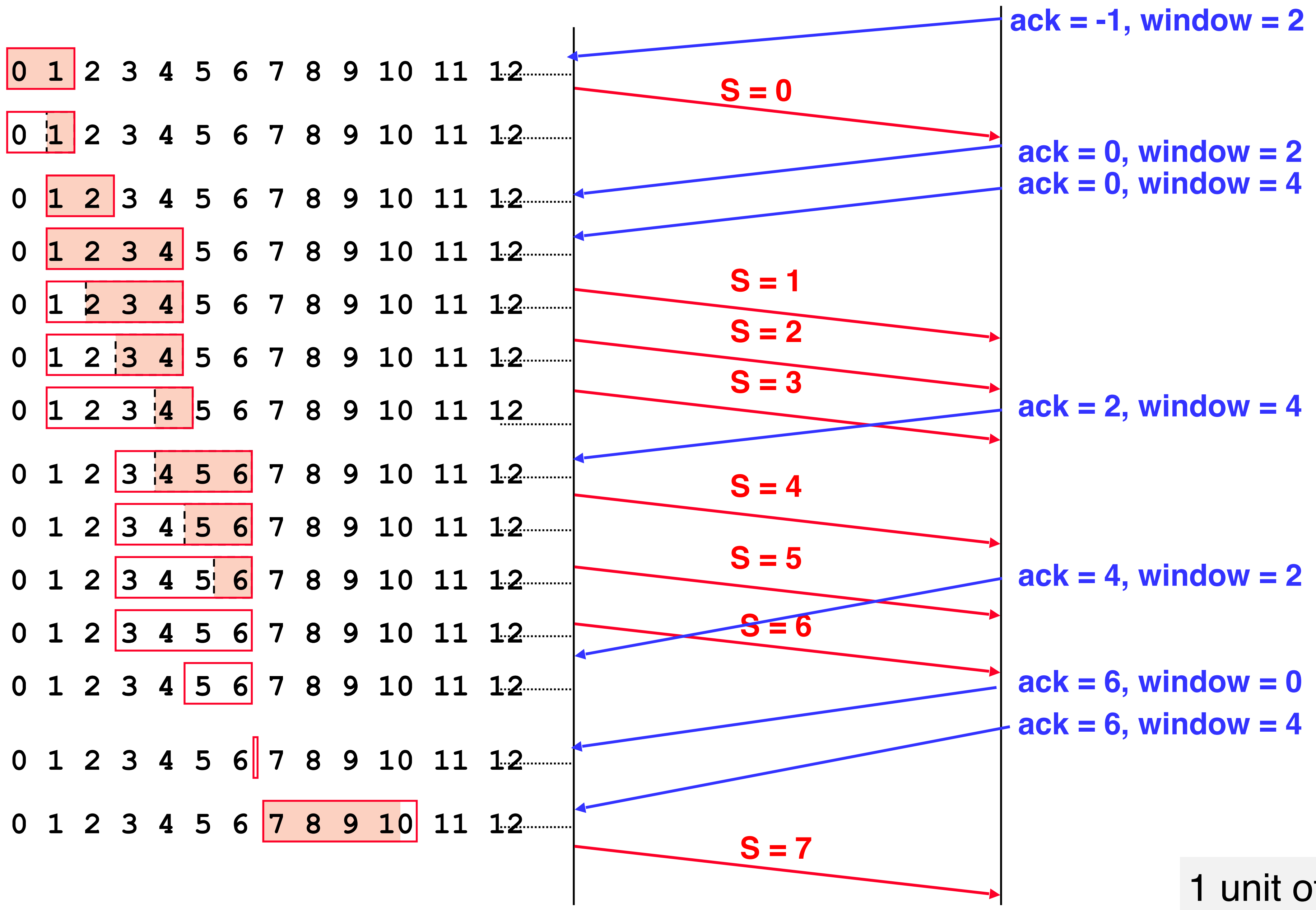
Window size = 4000 bytes
One packet = ▨ = 1000 bytes

Flow control: an *adaptive* window size

- TCP flow control constantly adapts the size of the window by sending *window advertisements* back to the source.
  - advertized window size is equal to available buffer size
  - if no space in buffer, window size is set to 0

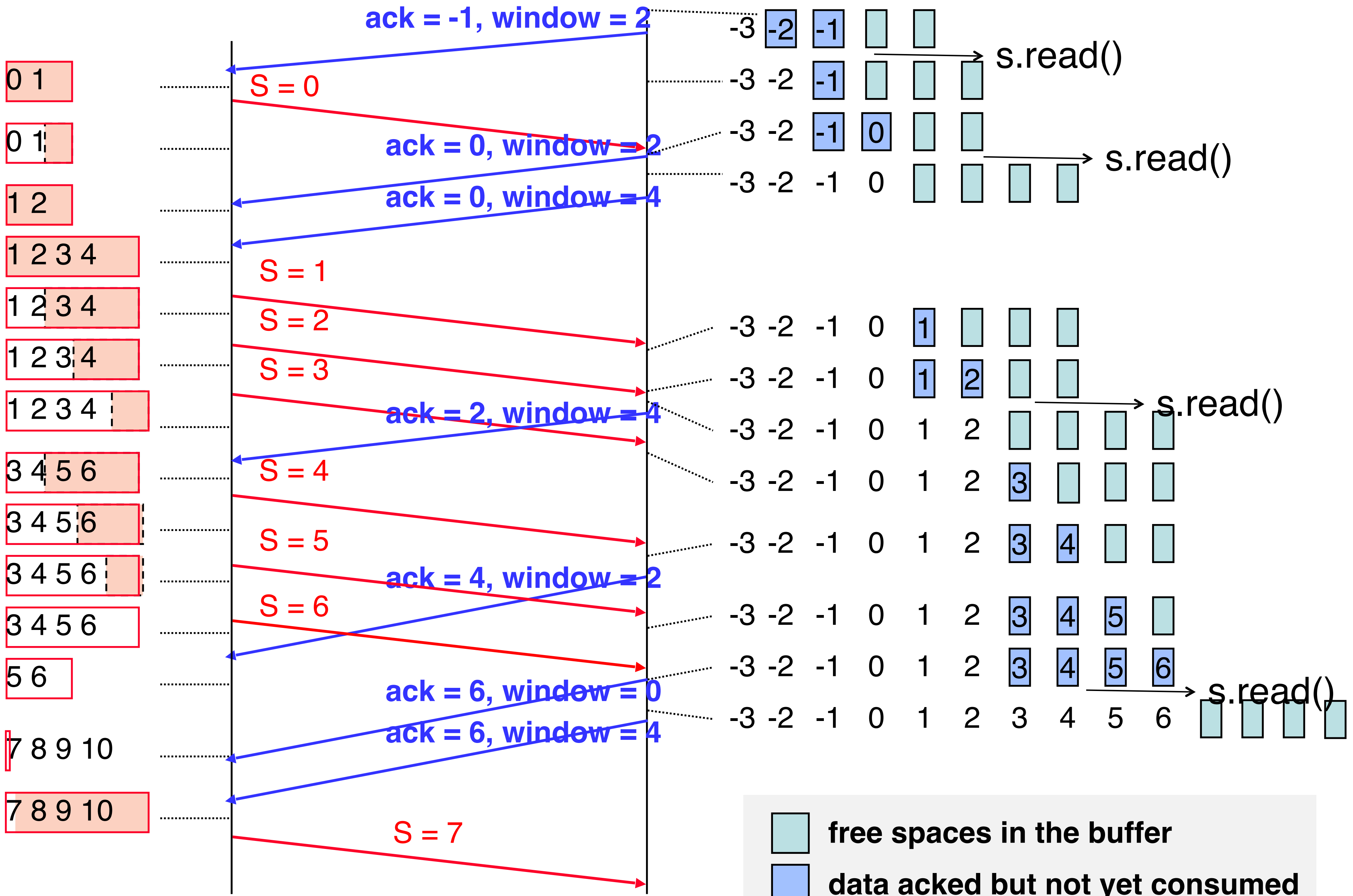
- thus, TCP adapts source's sending rate to receiver's consuming speed

$\neq$ Congestion Control, which adapts source's sending rate to network conditions
[we will see this in an oncoming later lecture]



1 unit of data = 1000 bytes

1 packet = 1000 bytes



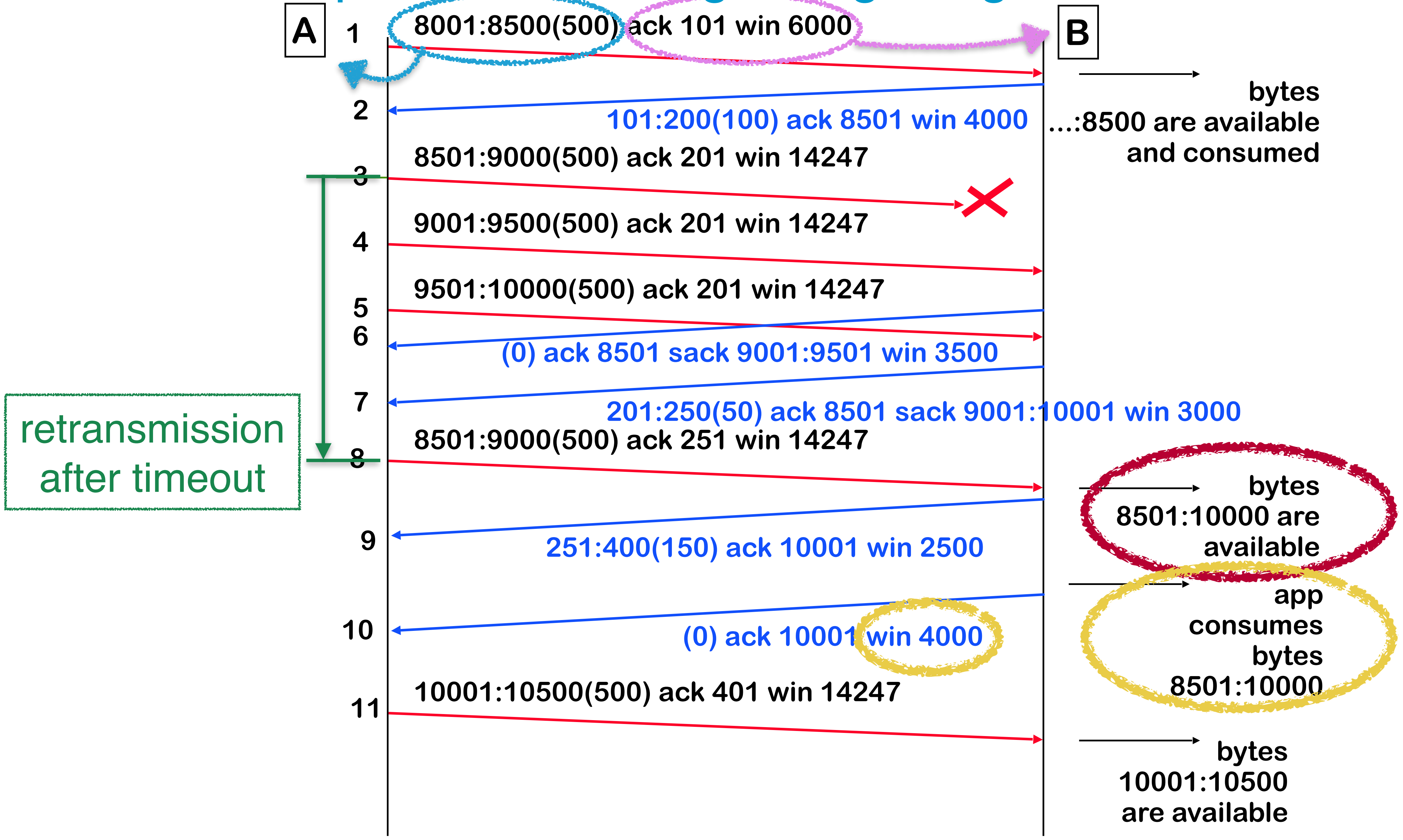
🟦 free spaces in the buffer

🟦 data acked but not yet consumed

receive buffer size = 4000 bytes

1 unit of data, 1 packet = 1000 bytes

TCP Basic Operation, Putting Things Together


A
B
1
8001:8500(500) ack 101 win 6000
2
101:200(100) ack 8501 win 4000
bytes
...:8500 are available
and consumed
3
8501:9000(500) ack 201 win 14247
4
9001:9500(500) ack 201 win 14247
5
9501:10000(500) ack 201 win 14247
6
(0) ack 8501 sack 9001:9501 win 3500
7
201:250(50) ack 8501 sack 9001:10001 win 3000
retransmission
after timeout
8
8501:9000(500) ack 251 win 14247
bytes
8501:10000 are
available
9
251:400(150) ack 10001 win 2500
app
consumes
bytes
8501:10000
10
(0) ack 10001 win 4000
11
10001:10500(500) ack 401 win 14247
bytes
10001:10500
are available

The picture shows a sample exchange of messages. Every packet carries the sequence number for the bytes in the packet; in the reverse direction, packets contain the acknowledgements for the bytes already received in sequence. The connection is bidirectional, with acknowledgements and sequence numbers for each direction. So here A and B are both senders and receivers.

Acknowledgements are not sent in separate packets ("piggybacking"), but are in the TCP header. Every segment thus contains a sequence number (for itself), plus an ack number (for the reverse direction). The following notation is used:

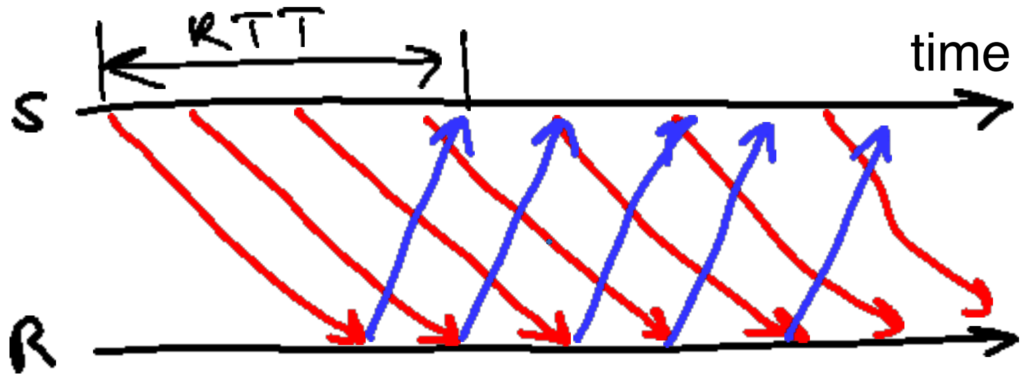
- firstByte":"lastByte+1 "("segmentDataLength") ack" ackNumber+1 "win" offeredWindowSise. Note the +1 with ack and lastByte numbers.

At line 8, A retransmits the lost data. When packet 8 is received, the application is not yet ready to read the data.

Later, the application reads (and consumes) the data 8501:10001. This frees some buffer space on the receiving side of B; the window can now be increased to 4000. At line 10, B sends an empty TCP segment with the new value of the window.

Note that numbers on the figure are rounded for simplicity. In real examples we are more likely to see non-round numbers (between 0 and 232 -1). The initial sequence number is not 0, but is chosen at random.

If there's no loss or reordering, and on a link with capacity c bytes/second, the min window size required for sending at the capacity is…



A. $W_{min} = RTT \times c$

B. $W_{min} = \frac{c}{RTT}$

C. $W_{min} = \frac{RTT}{c}$

D. None of the above

E. I do not know

Go to web.speakup.info or download speakup app

Join room 46045

Solution

Answer A

If the window size is large enough, the window is never fully used and the sender can send at rate c.

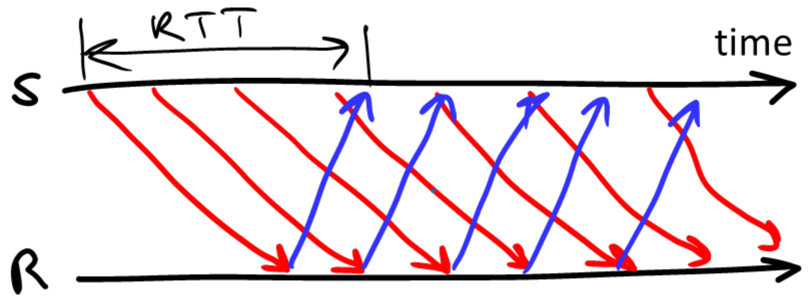
This case occurs when the total amount of data in flight, $c \times RTT$, is not larger than W, i.e. when $W \geq c \times RTT$

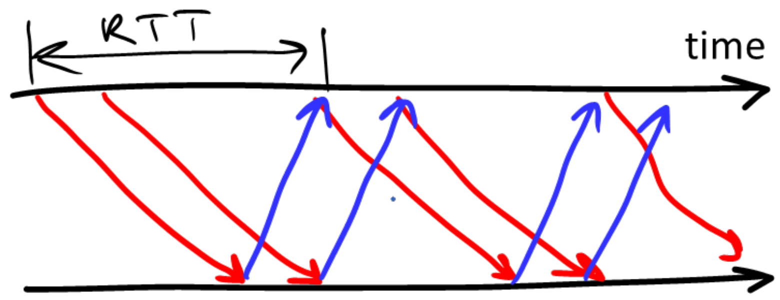
(i.e. window ≥ bandwidth-delay product)

If the window size is small, the sender is blocked after sending a full window.

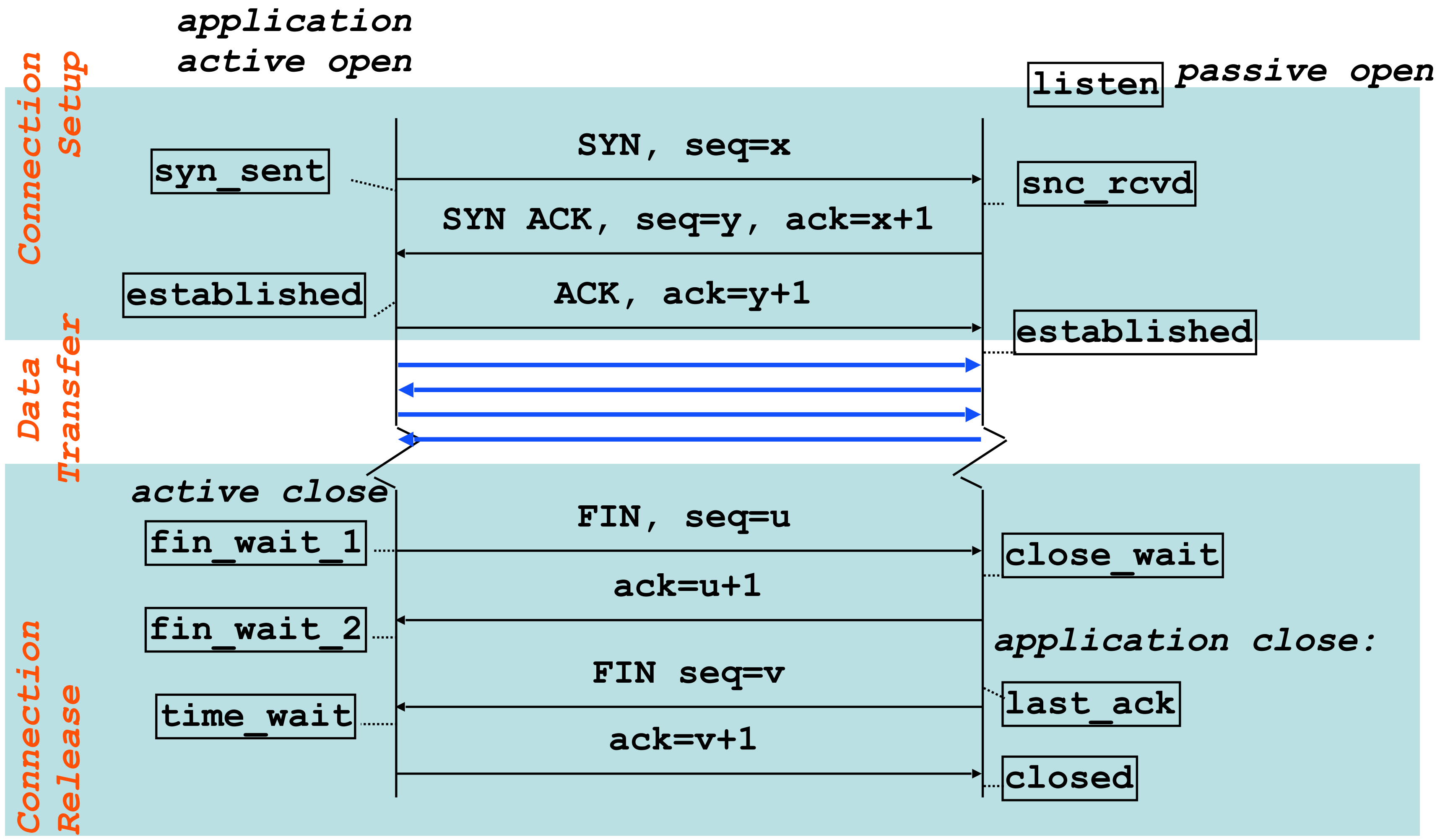
The sending rate in this case is $\frac{W}{RTT}$.

This case occurs when $W < c \times RTT$





3. TCP Connection Phases



Before transmitting useful data, TCP requires a *connection setup* phase:

- used to agree on seq numbers and make sure buffers and window are initially empty

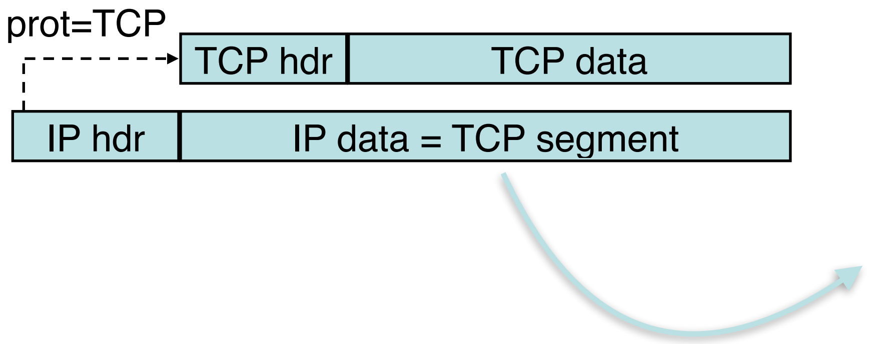
The previous slide shows all phases of a TCP connection:

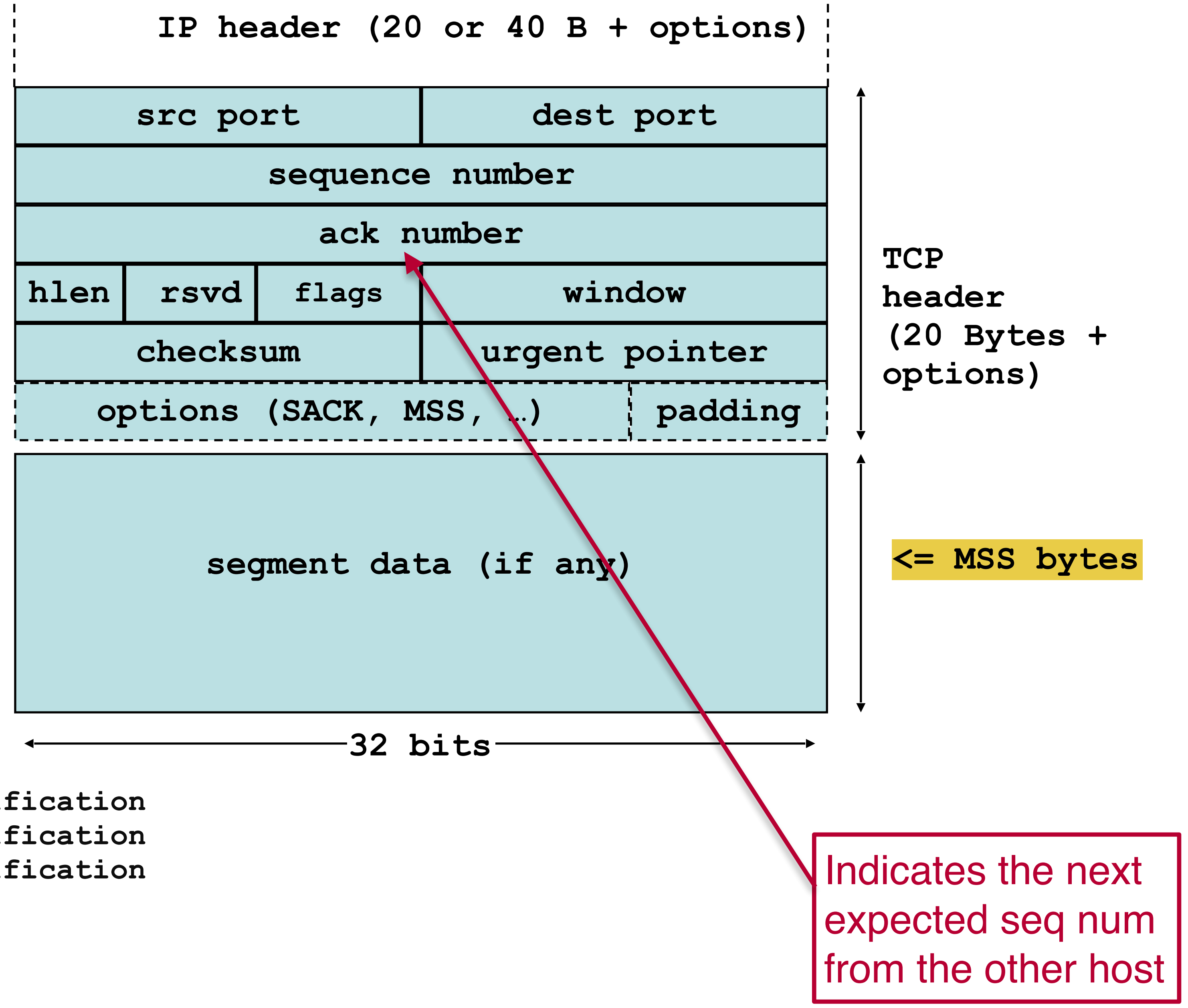
- Before data transfer takes place, the TCP connection is opened using SYN packets. The effect is to synchronize the counters on both sides.
- The initial sequence number is a random number.
- Then the data transfer begins and works as described earlier.
- Finally the connection closes. This can be done in a number of ways. The picture shows a graceful release where both sides of the connection are closed in turn.

There are many more subtleties (e.g. how to handle connection termination, lost or duplicated packets during connection setup, etc [see Textbook sections 4.3.1 and 4.3.2]

Recall: TCP connections involve only two hosts; routers in between are not involved.

TCP Segment Format





flags	meaning
NS	used for explicit congestion notification
CWR	used for explicit congestion notification
ECN	used for explicit congestion notification
urg	urgent ptr is valid
ack	ack field is valid
psh	this seg requests a push
rst	reset the connection
syn	connection setup
fin	sender has reached end of byte stream

The previous slide shows the TCP segment format.

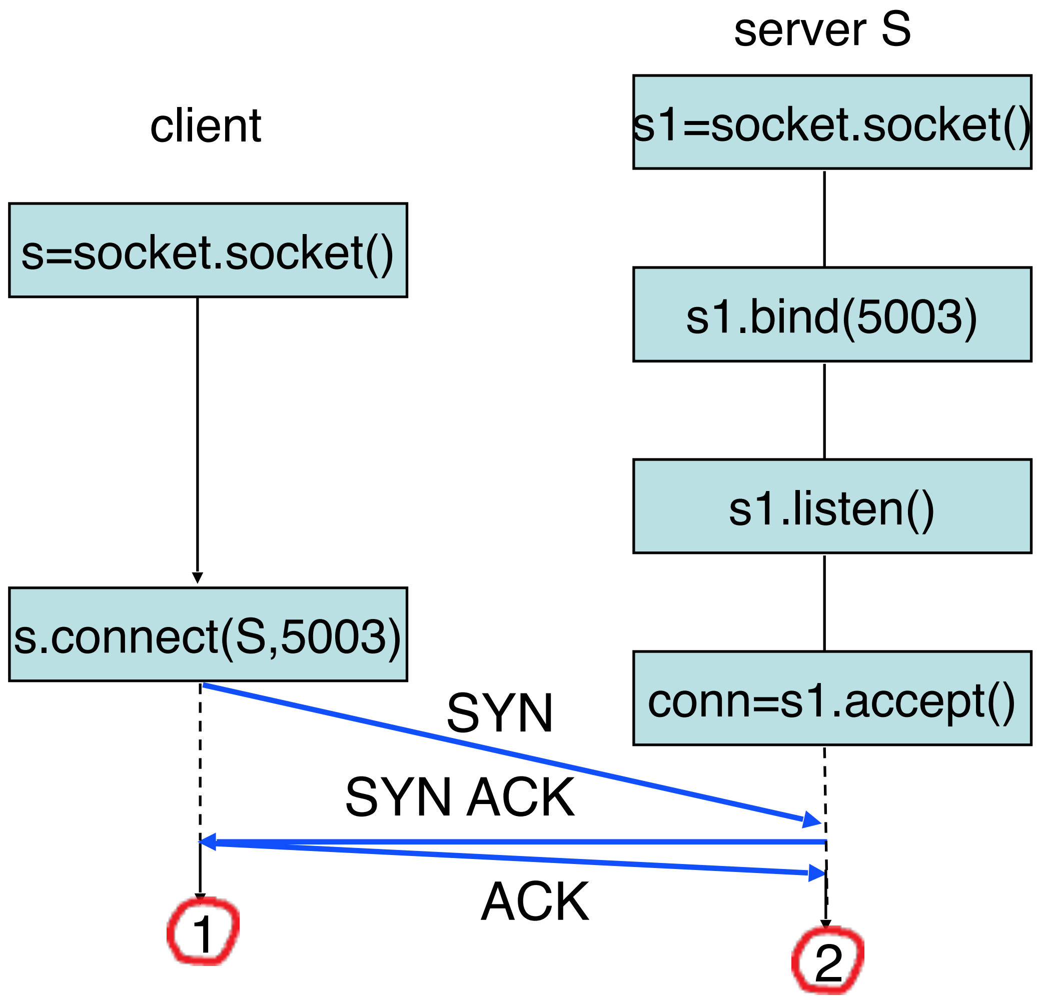
- *SYN* and *FIN* are used to indicate connection setup and close. Each one uses one sequence number.
- The *sequence* number is that of the *first byte* in the data.
- The *ack* number is the *next expected sequence number*.
- Options may include the Selective ack (*SACK*) field, or the Maximum Segment Size (*MSS*), which is negotiated during SYN-SYNACK phase—the negotiation of the maximum size for the connection results in the smallest value to be selected.
- The checksum is mandatory.
- The NS, CRW and ECN bits are used for congestion control [see lecture on congestion control].
- The push bit can be used by the upper layer using TCP; it forces TCP on the sending side to create a segment immediately. If it is not set, TCP may pack together several SDUs (=data passed to TCP by the upper layer) into one PDU (= segment). On the receiving side, the push bit forces TCP to deliver the data immediately. If it is not set, TCP may pack together several PDUs into one SDU. This is because of the stream orientation of TCP. TCP accepts and delivers contiguous sets of bytes, without any structure visible to TCP. The push bit is used by Telnet after every end of line.
- The urgent bit indicates that there is urgent data, pointed to by the urgent pointer (the urgent data need not be in the segment). The receiving TCP must inform the application that there is urgent data. Otherwise, the segments do not receive any special treatment. This is used by Telnet to send interrupt type commands.
- RST is used to indicate a RESET command. Its reception causes the connection to be aborted.

TCP Sockets

More *complicated* than UDP because of the need to open/close a connection

Opening a TCP connection requires one side to *listen* (this side is called "server") and one side to *connect* (called "client")

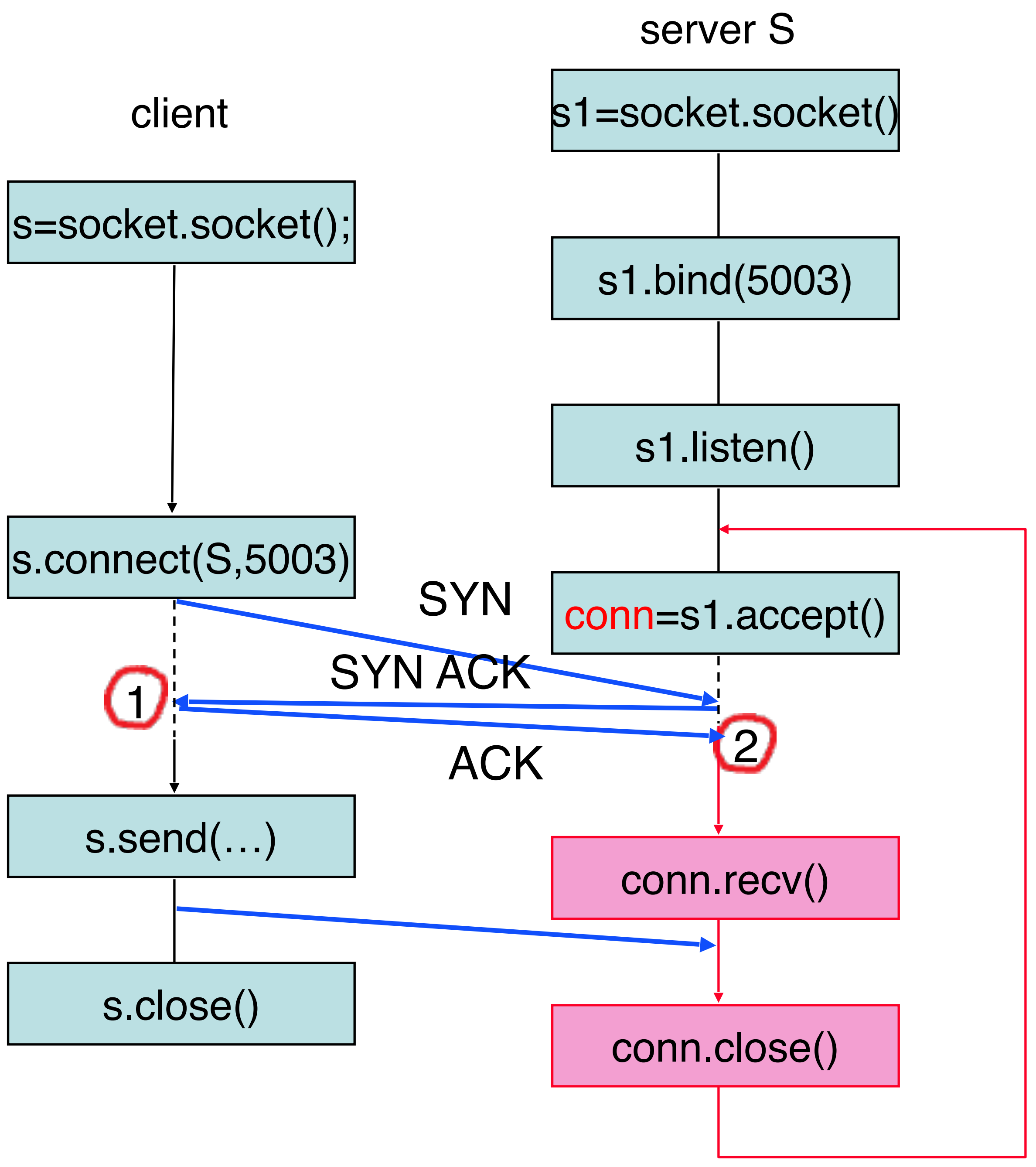
At t=1, client can use the connection to send or receive data on this socket



A New Socket is Created by accept()

At t=2, on server side, a new socket (conn) is created – will be used by server to send or receive data.

This example is simplistic: client sends one message to server and quits;
server handles one client at a time.



The figure of the previous 2 slides shows toy client and servers. The client sends a string of chars to the server which reads and displays it.

- socket(AF_INET,…) creates an IPv4 socket and returns a socket object if succesful socket(AF_INET6,…) creates an IPv6 socket
- bind(5003) associates the local port number 5003 with the socket; the server must bind, the client need not bind, a temporary port number is allocated by the OS
- connect(S,5003) associates the remote IP address of S and its port number with the socket and sends a SYN packet
- send() sends a block of data to the remote destination
- listen() declares the size of the buffer used for storing incoming SYN packets;
- accept() blocks until a SYN packet is received for this local port number. It creates a new socket (in pink) and returns the file descriptor to be used to interact with this new socket
- recv() blocks until one block of data is ready to be consumed on this port number. You must tell in the argument how many bytes at most you want to read. It returns a block of bytes or raises an exception when the connection was closed by the other end.

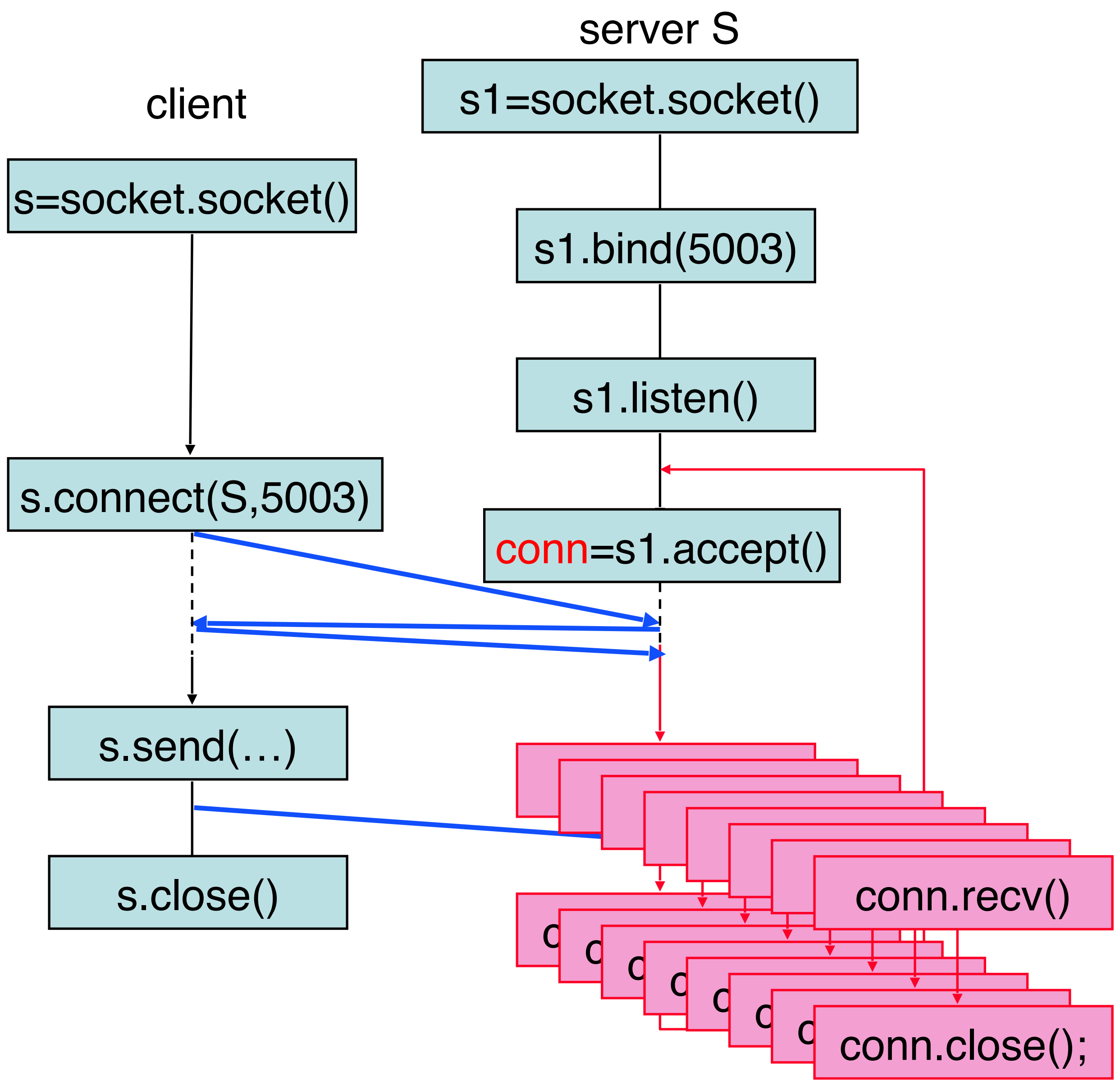
A more practical server

TCP Server uses parallel execution threads to handle several TCP connections + to listen to incoming connections

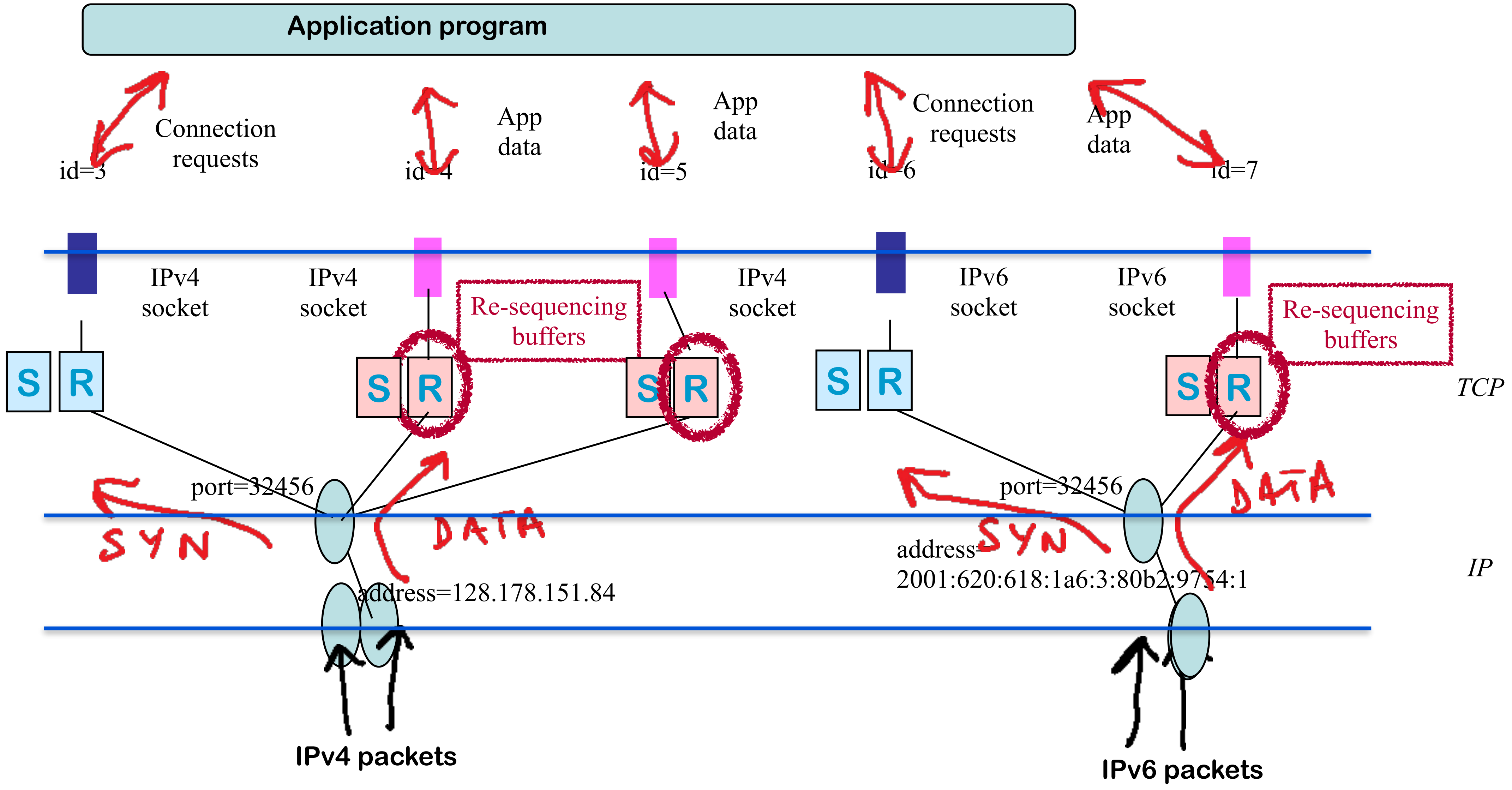
“conn”-type are *connected* sockets (*pink*),
“s1”-type is a *non-connected* socket (*blue*)

TCP uses default port number for listening; e.g. TCP port 80 is used for web servers

A TCP connection is identified by:
src IP addr, src port, dest IP addr, dest port



How the Operating System views TCP Sockets



MSS and segmentation

TCP, not the application, chooses how to segment data
TCP segments should not be fragmented at source

TCP segments have a maximum size (called MSS):

- *default* values are:

  536 bytes for IPv4 operation (576 bytes IPv4 packet),

  1220 bytes for IPv6 operation (1280 bytes IPv6 packets)
- otherwise *negotiated* in Options header field during connection setup = hosts set it to the smallest value that both declare

Modern OSs use TCP Segmentation Offloading (TSO): Segmentation is performed at the network interface card NIC with *hardware* assistance (reduces CPU consumption of TCP)

Recap: TCP offers a streaming service

Sender side:

- data accumulates in send buffer until TCP decides to create a segment

Receiver side:

- data accumulates in receive buffer until put in order and application reads it

No boundaries between bytes: several small messages written by A's app may be received by B as a single segment—
and conversely, a single message written by A's app may be received by B as multiple segments;

➡ so, apps need to group bytes to messages (if needed)

A side effect is *head of the line blocking*: If one packet sent by A is lost, all data following this packet is delayed until the loss is repaired

For which types of apps may TCP’s streaming service be an issue? (multiple answers are fine)

A. an app using http/1, where we have one TCP connection per object
B. an app using http/2, where we have one TCP connection per website
C. a real time video streaming application that sends a new packet every msec
D. None
E. I do not know

Go to web.speakup.info or download speakup app

Join room 46045

Solution

Answer F: (B and C) For http/2 with one single connection, head-of-the line blocking can occur: if one packet is lost in the transfer of one object of the page, the entire page download is delayed until the loss is repaired.

Head-of-the line blocking may also occur for a real-time streaming app and is probably even worse: with TCP, the loss of one packet delays all subsequent packets until the loss is repaired, whereas the live application might prefer to skip the lost packet and receive the most recent one. Such an app should use UDP.

Why both TCP and UDP ?

Most applications use TCP rather than UDP, as this avoids re-inventing error recovery in every application

But some applications do not need error recovery in the way TCP does it (i.e. by packet retransmission)

For example: Voice applications / Sensor data streaming
Q. why ?

For example: an application that sends just one message, like name resolution (DNS).
Q. Why ?

For example: multicast (TCP does not support multicast IP addresses)

Why both TCP and UDP ?

Most applications use TCP rather than UDP, as this avoids re-inventing error recovery in every application

But some applications do not need error recovery in the way TCP does it (i.e. by packet retransmission)

- For example: Voice applications / Sensor data streaming
  Q. why ?
  A. Delay is important for interactive voice, while packet retransmission may introduce too much delay in some cases.
  Sensor data streaming may send a new packet every few msecs, better to receive latest packet than to repeat a lost one.
- For example: an application that sends just one message, like name resolution (DNS).
  Q. Why ?
  A. TCP sends several packets of overhead before one single useful data message. Such an application is better served by a Stop and Go protocol at the application layer.
- For example: multicast (TCP does not support multicast IP addresses)