ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

The Transport Layer: TCP and UDP

Jean-Yves Le Boudec

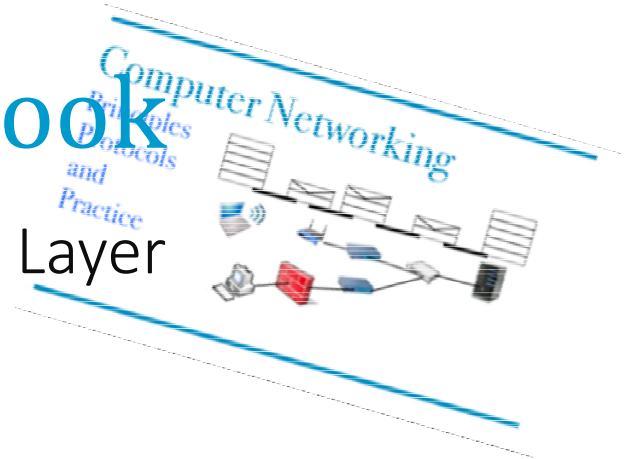
2018

Contents

1. The transport layer, UDP
2. TCP Basics: Sliding Window and Flow Control
3. TCP Connections and Sockets
4. More TCP Bells and Whistles
5. Where should packet losses be repaired ?

Textbook

Chapter 4: The Transport Layer



1. The Transport Layer

Reminder:

network + link + phy carry packets end-to-end

transport layer makes network services available to programs

is in end-systems only, not in routers

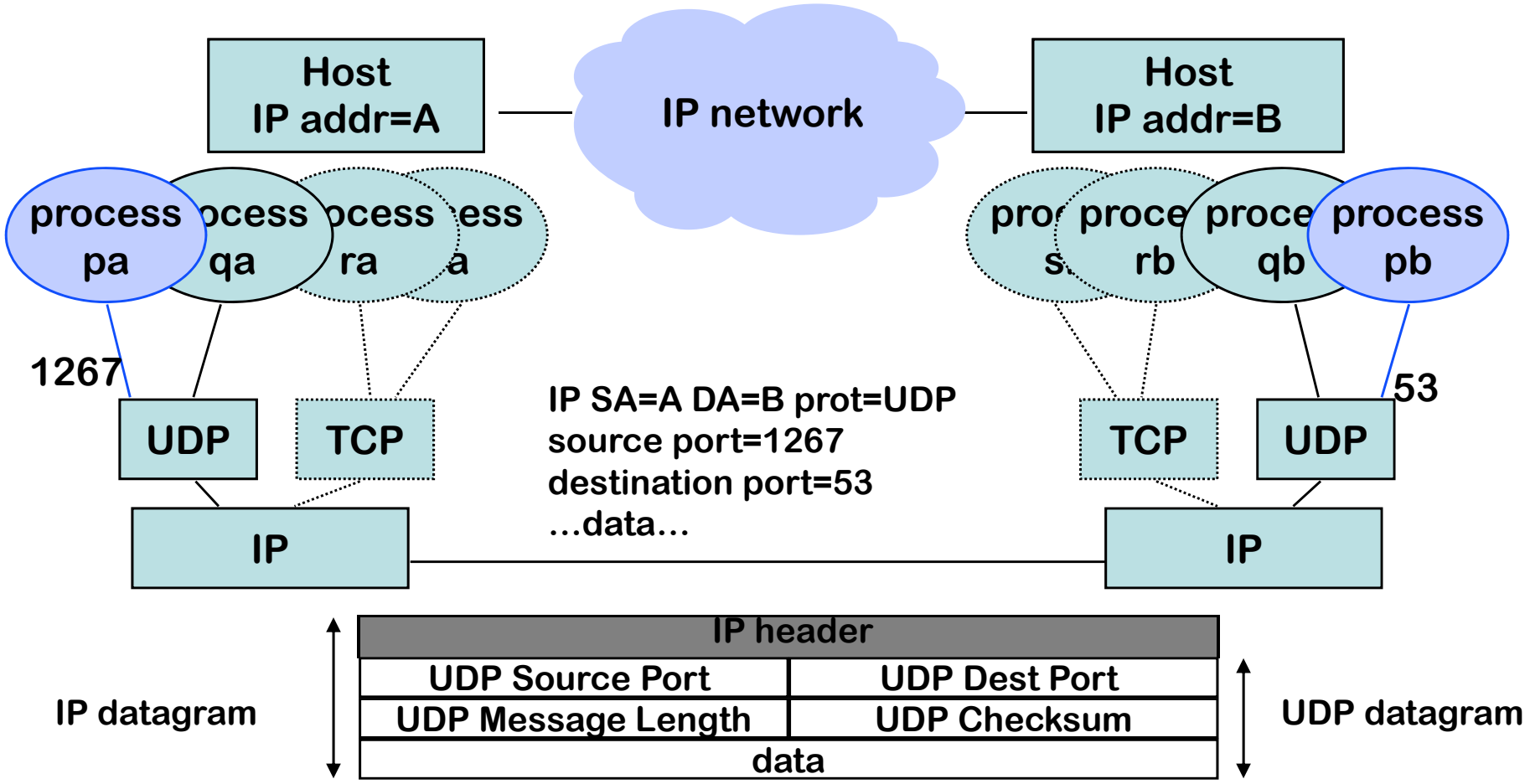
In TCP/IP there are mainly two transport layers

UDP (User Datagram Protocol):

TCP (Transmission Control Protocol): error recovery + flow control

There is no TCPv6 nor UDPv6, the same TCP and UDP are used over IPv4 and IPv6

UDP Uses Port Numbers



The picture shows two processes (= application programs) pa, and pb, 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 name 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 the protect 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 ports; for example, 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.

The UDP service is message oriented

UDP service interface

- one message, up to 65,535 bytes
- destination address, destination port, source address, source port
- destination address can be unicast or multicast

UDP service is message oriented

- UDP delivers exactly the message (called “Datagram”) or nothing
- consecutive messages may arrive in disorder
- message may be lost -- application must handle

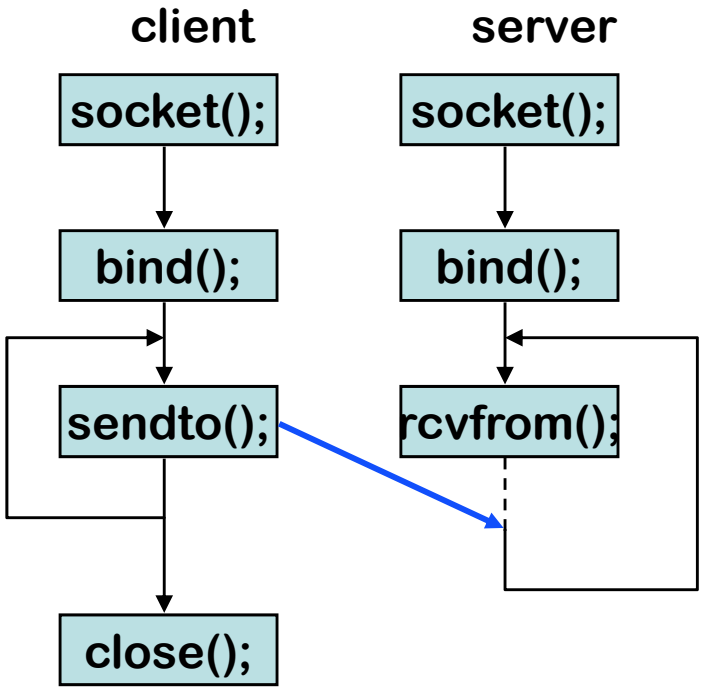
If a UDP message is larger than the possible maximum size for the IP layer, MTU, then fragmentation occurs at the IP layer – this is not visible to the application program

UDP is used via a Socket Library

The socket library provides a programming interface to TCP and UDP

The figure shows toy client and server UDP programs. The client sends one string of chars to the server, which simply receives (and displays) it.

- socket(AF_INET,…) creates an IPv4 socket and returns a number (=file descriptor) if successful; socket(AF_INET6,…) creates an IPv6 socket
- bind() associates the local port number with the socket
- sendto() gives the destination IP address, port number and the message to send
- recvFrom() blocks until one message is received for this port number. It returns the source IP address and port number and the message.




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

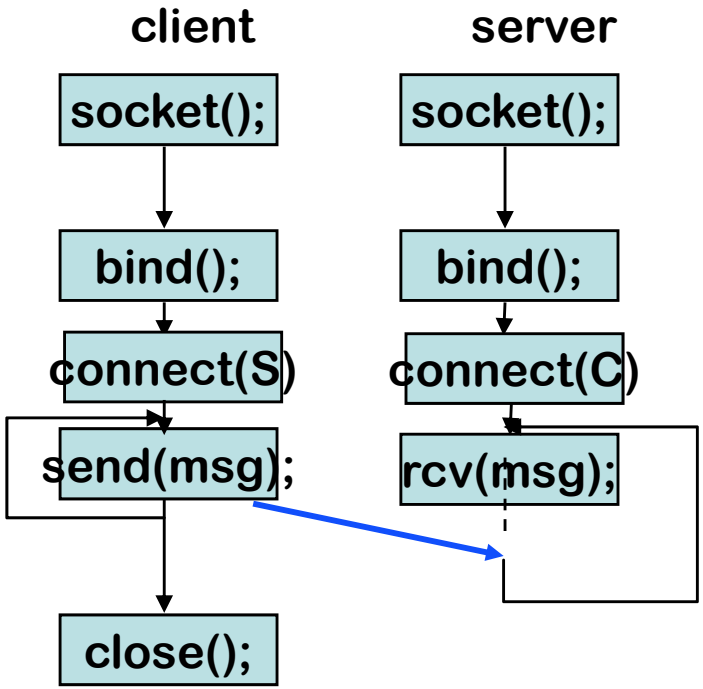
```
% ./udpServ &
%
```

“Connected” UDP Socket

UDP is connectionless, can send to / receive from any / multiple remote hosts on same socket

connect forces a UDP socket to send or receive only from one specific remote host.
send() (instead of sendto()) and recv() (instead of recvfrom()).

Such a UDP socket is called *connected,* but there is no connection (synchronization of state) as there is with TCP.




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

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

Is there a UDPv6 ?

There is no UDPv6 (nor TCPv6), as the UDP and TCP protocols are not affected by the choice of IPv4 or IPv6

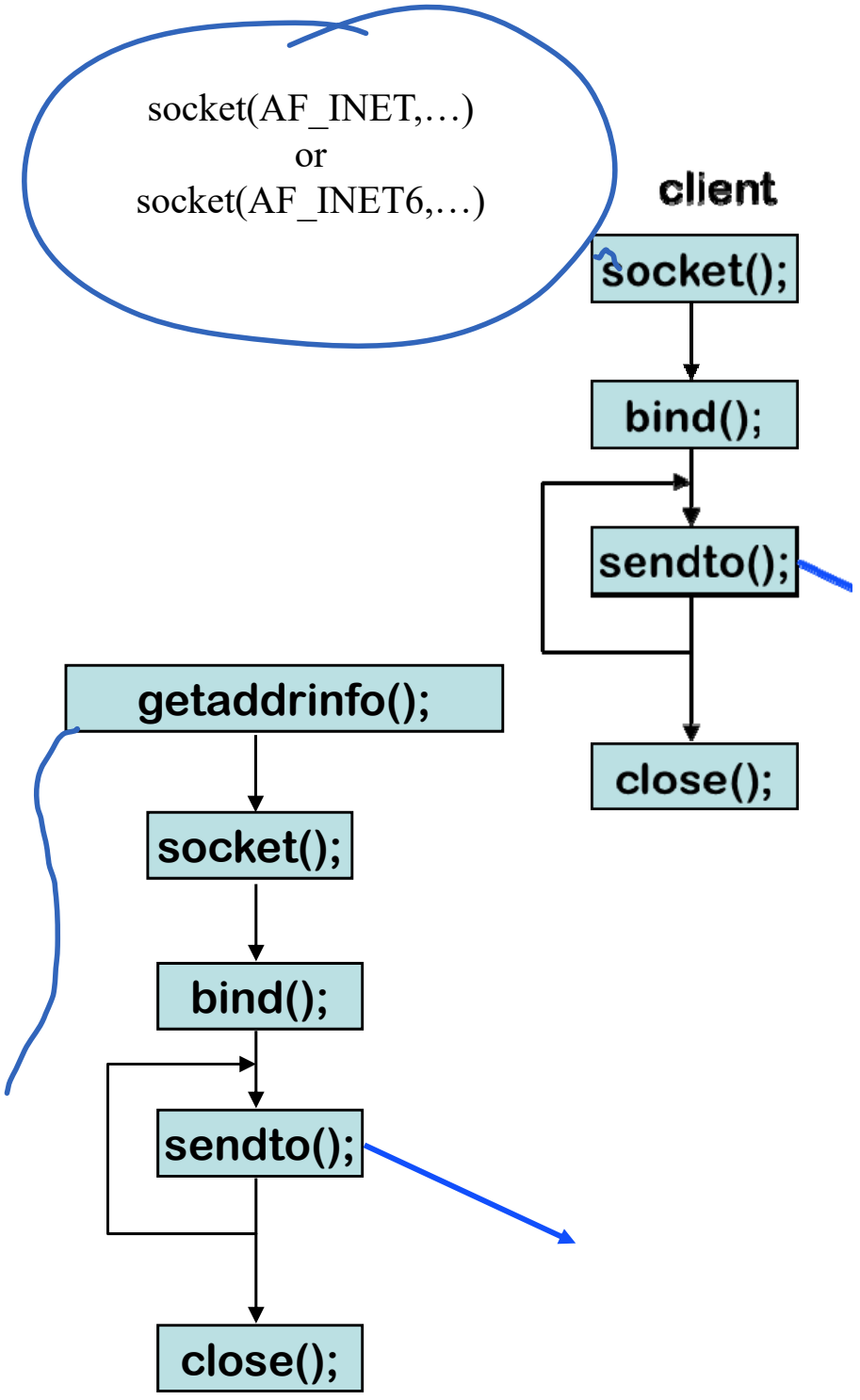
However, there are UDPv4 sockets and UDPv6 sockets, i.e. the service interfaces are affected.

An application program can decide to

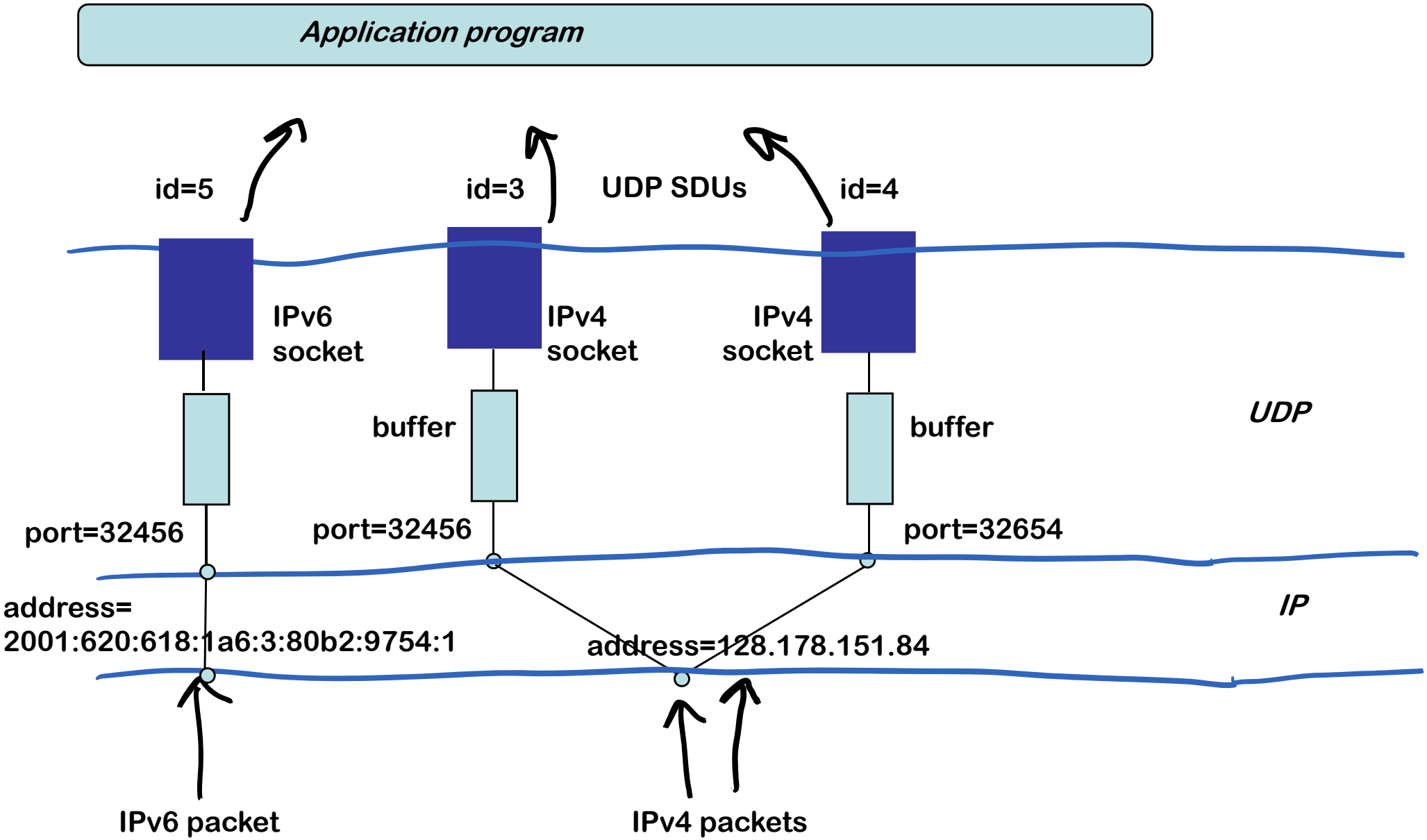
- use IPv4
- use IPv6
- or to support both (modern program)

Modern programs use DNS to know what is available;

If both IPv4 and IPv6 are available, some systems provide support to help decide which one is preferred.



How the Operating System views UDP



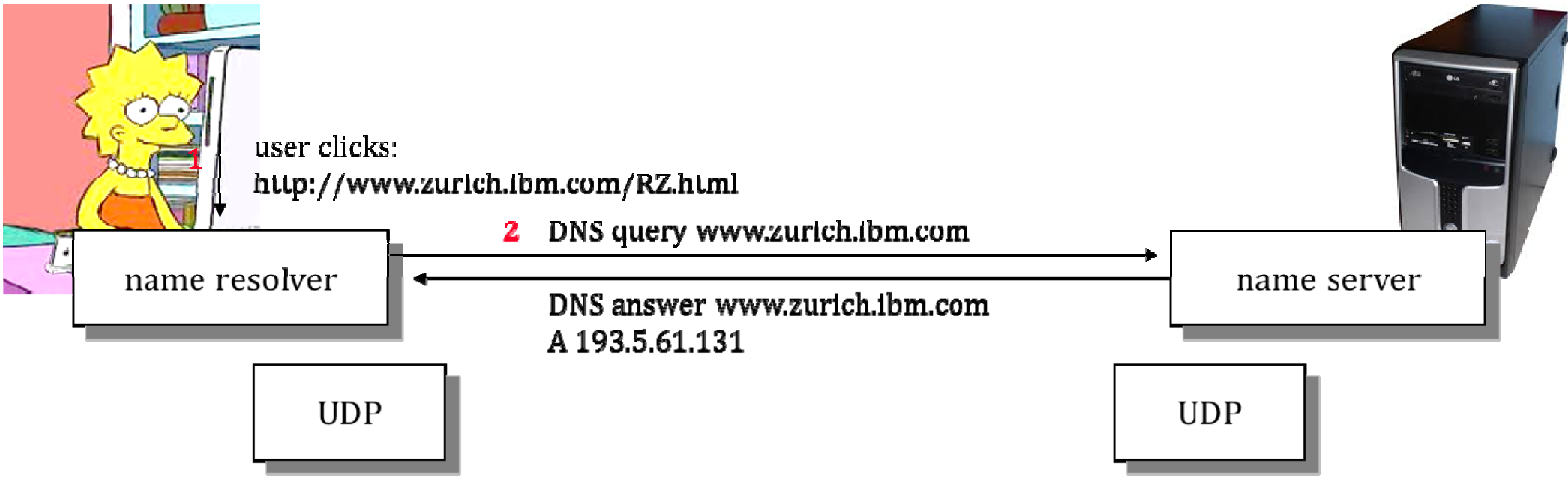
How the Operating System views UDP

On the sending side: Operating System sends the UDP datagram as soon as possible

On the receiving side: Operating System re-assembles UDP datagram (if required) and keeps it in buffer ready to be read. Packet is removed from buffer when application reads.

IPv6 sockets are in a different space than IPv4 sockets

Lisa'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

2. TCP Basics: Sliding Window and Flow Control

In the Internet, packets may be lost

- buffer overflow
- physical layer errors

UDP application must handle loss

TCP solves the problem once for all

TCP offers in-sequence, lossless delivery

What does TCP do ?

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

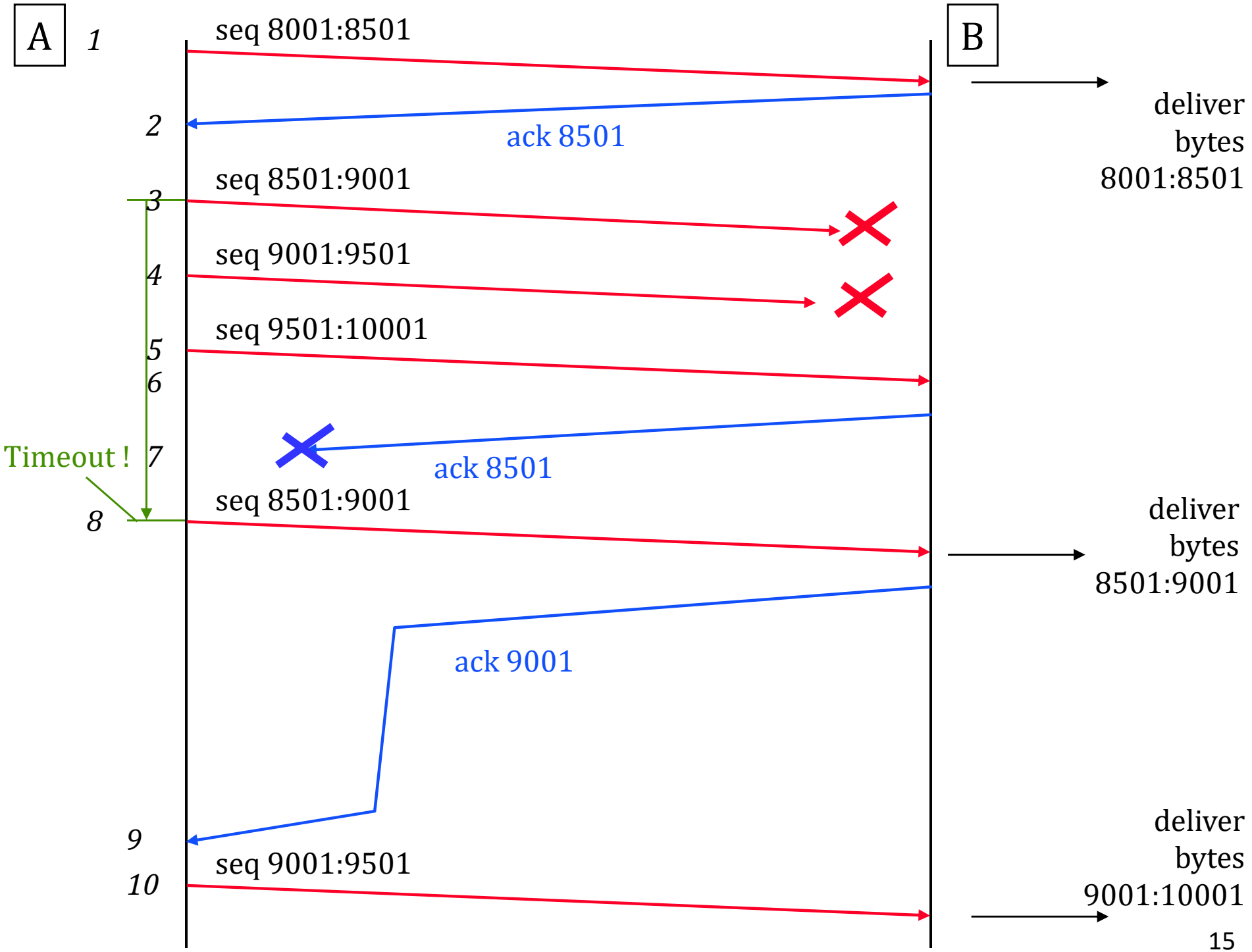
How does TCP work ?

data is numbered (per-byte sequence numbers)

a connection (=synchronization of sequence numbers) is opened between sender and receiver

TCP waits for acknowledgements; if missing data is detected, TCP re-transmits

TCP Basic Operation 1: showing 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. The maximum segment size is 1000 bytes. Ranges such as 8001:8501 mean bytes numbers 8001 to 8500.

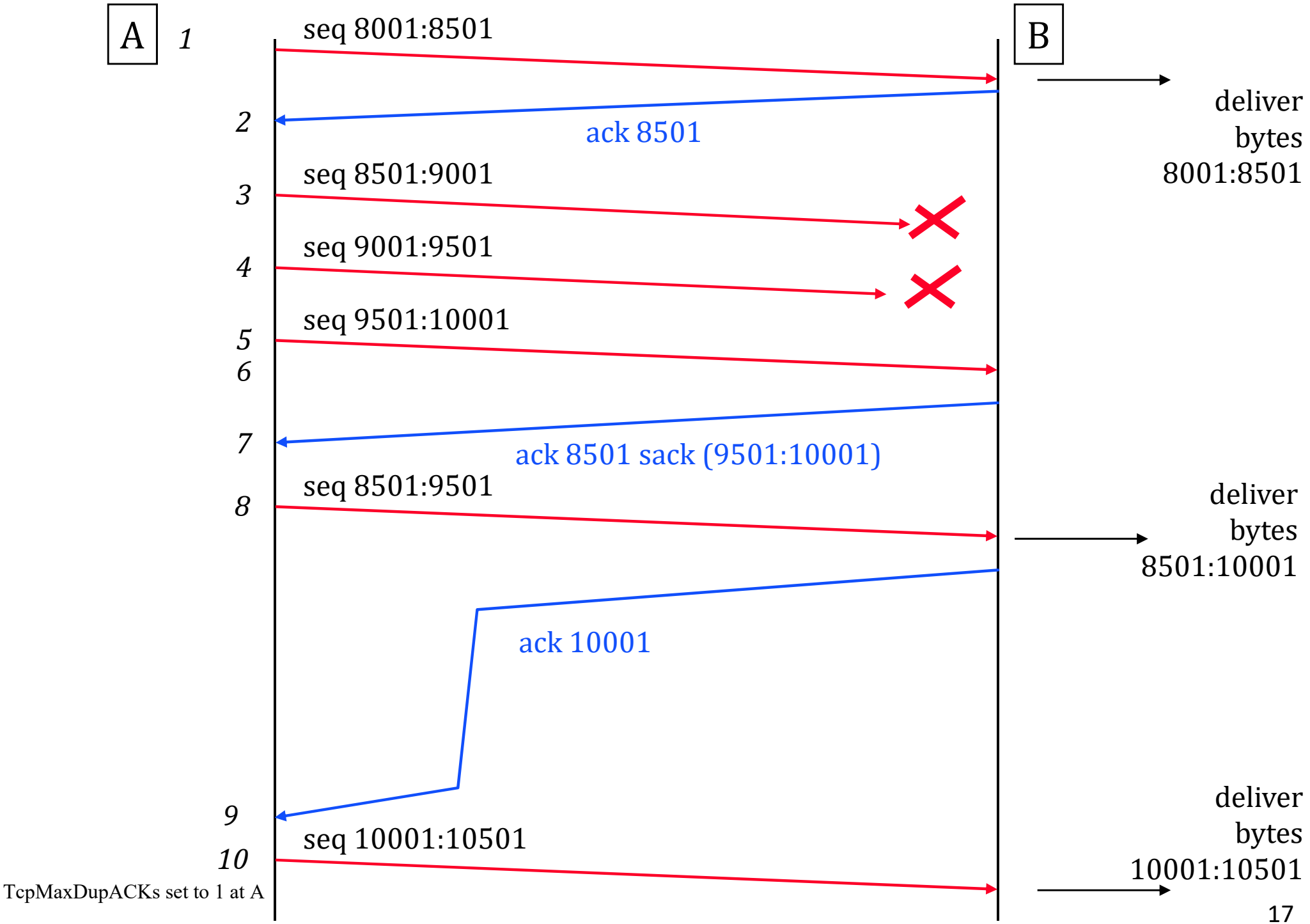
Packets 3, 4 and 7 are lost.

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.

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). A re-sends data that is detected as lost, i.e. bytes 8501:9001. When receiving packet 8, B can deliver to the application all bytes 8501:9001.

When receiving packet 10, B can deliver bytes 9001:10001 because packet 5 was received and kept by B in the receive buffer.

TCP Basic Operation 1: showing SEQ, ACK and SACK



In addition to the ACK field, most TCP implementation 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. The maximum segment size is 1000 bytes. Packets 3 and 4 are lost.

At line 6, B is 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. Since the maximum segment size is 1000 bytes, only one packet is sent.

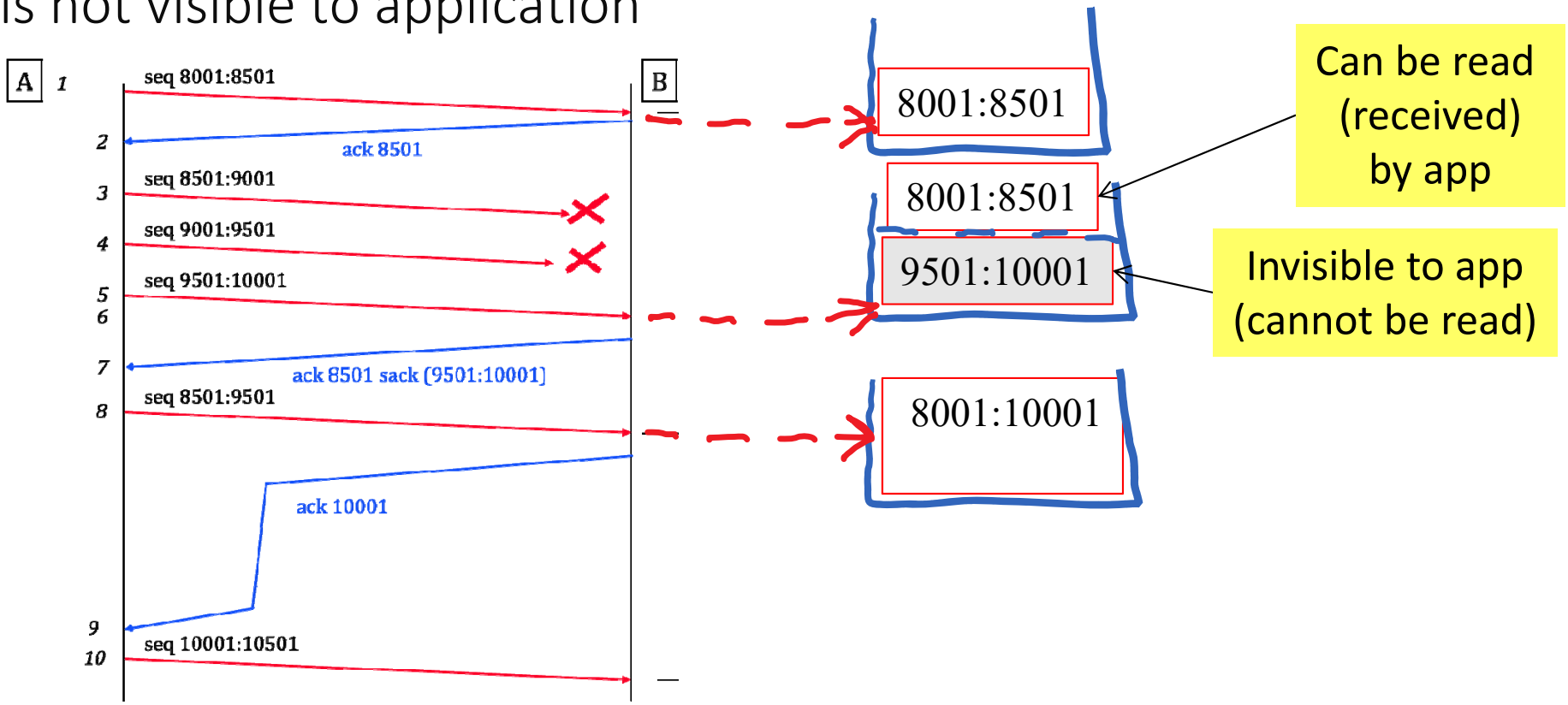
When receiving packet 8, B can deliver bytes 9001:10001 because packet 5 was received and kept in the receive buffer.

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 data

Packets may need re-sequencing; out-of-sequence data is stored but is not visible to application



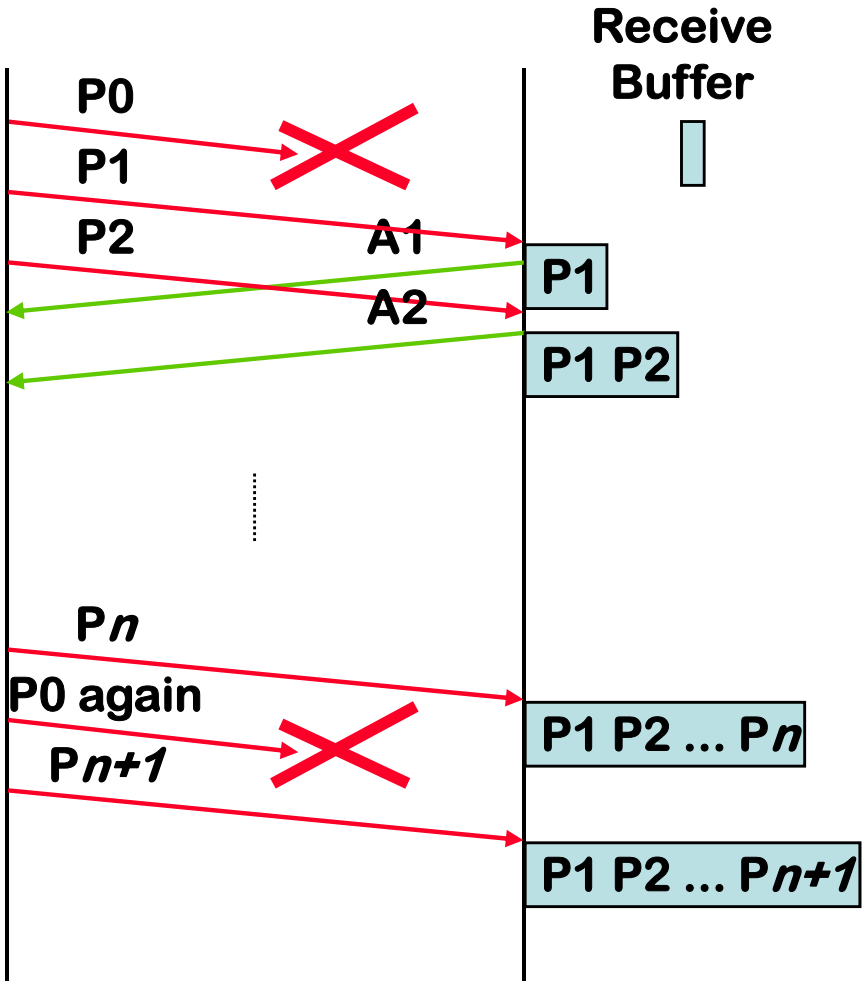
TCP uses a sliding window

The receive buffer may overflow if one piece of data "hangs"

- E.g. multiple losses affecting the same packet

This is why the sliding window was invented

- limits the number of data "on the fly"

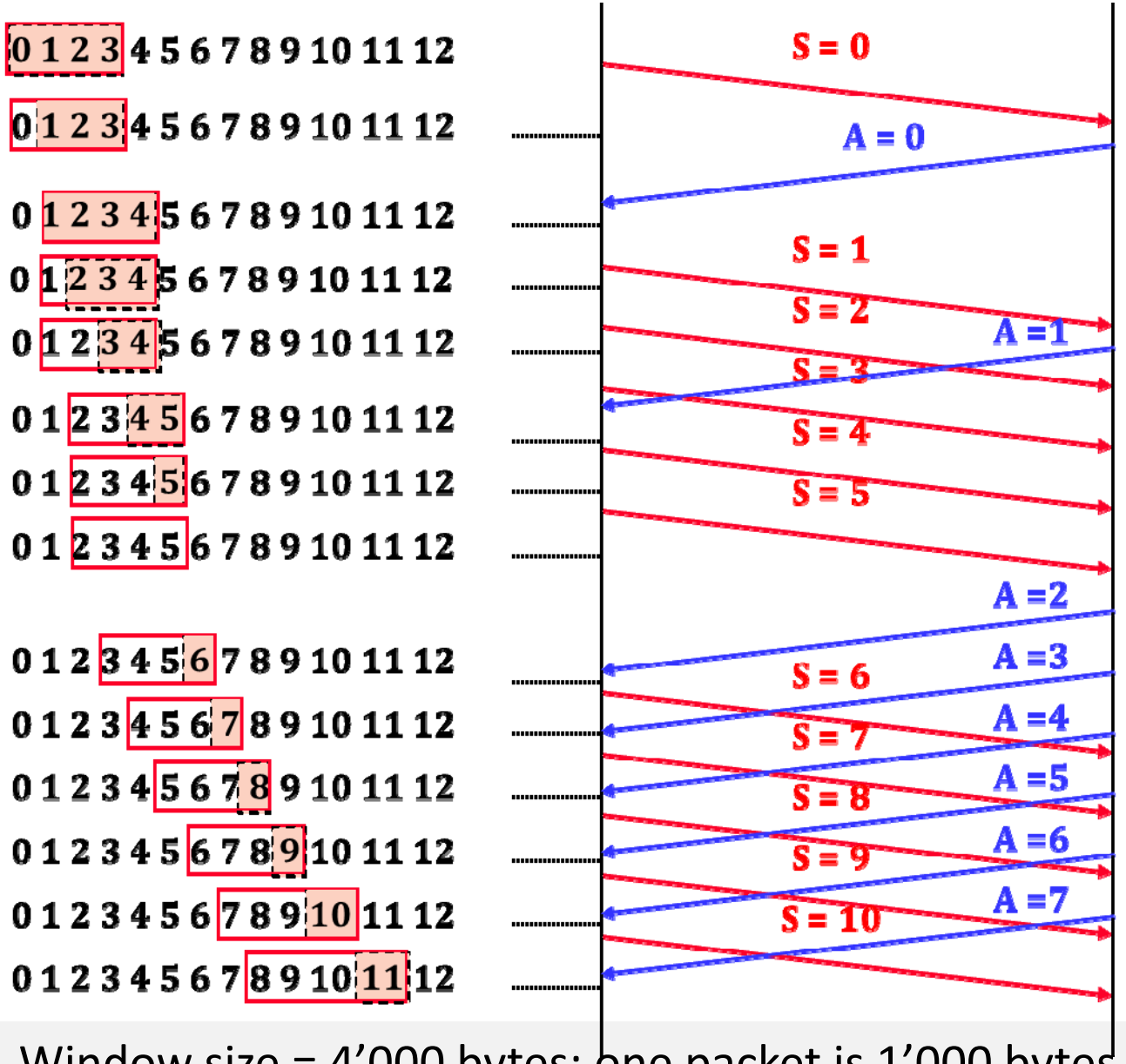


How the sliding window works

lower edge = smallest non acknowledged sequence number

upper edge = lower edge + window size

Only sequence numbers that are in the window may be sent

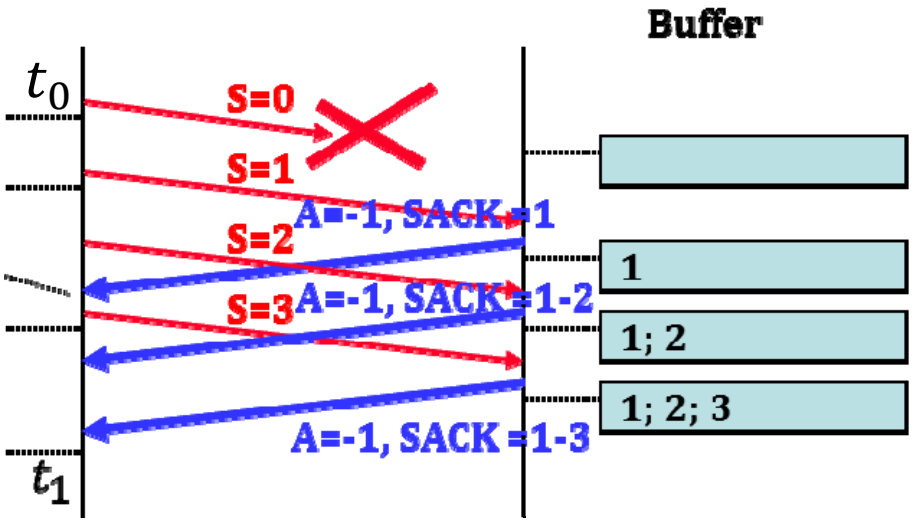


Window size = 4’000 bytes; one packet is 1’000 bytes

Window

Usable part of the window

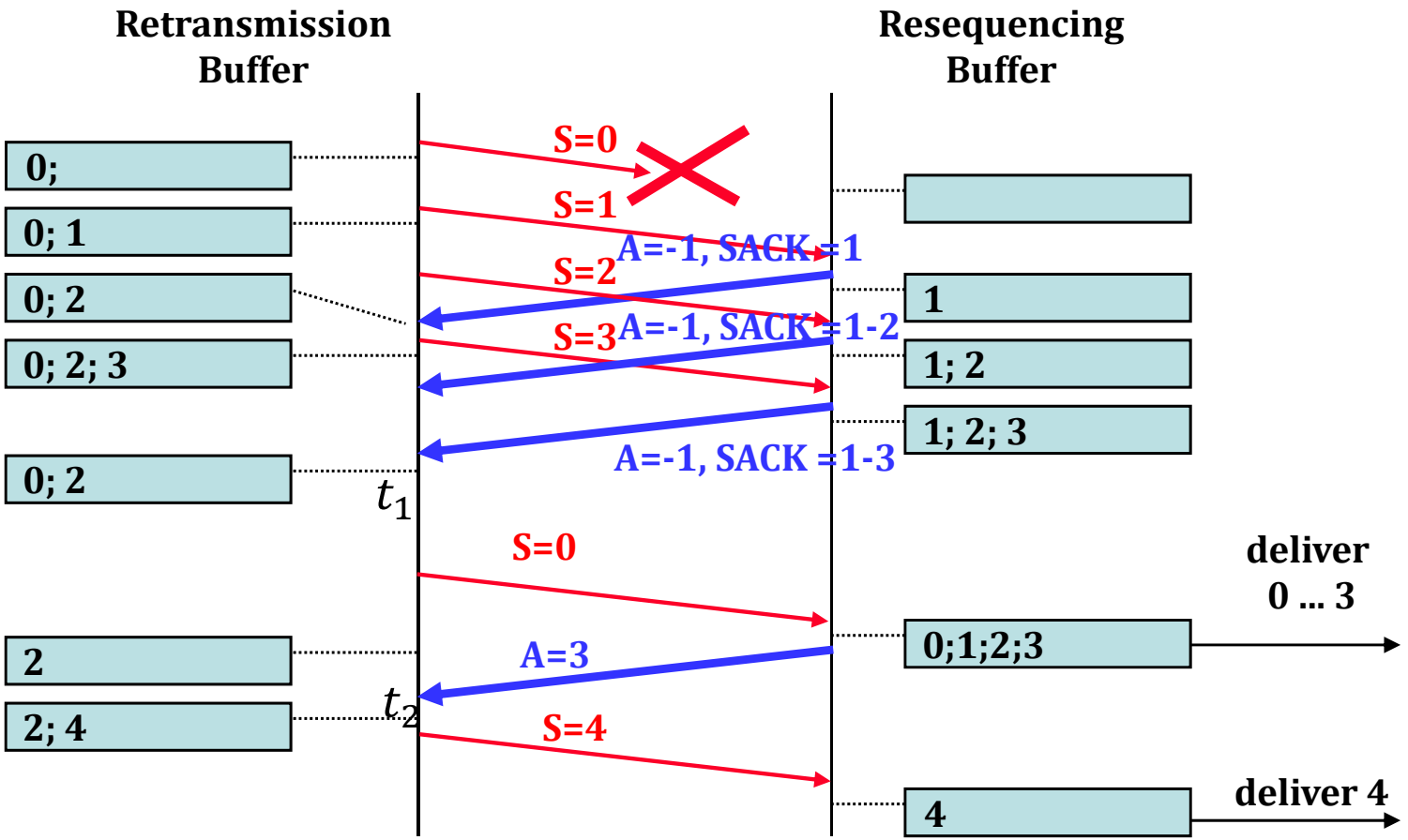
At time t_1, sender…



Window size = 4'000 bytes, one packet = 1'000 bytes
Sliding window was initialized at time t_0

A. … can send packet 4
B. … cannot send packet 4
C. It depends on whether data was consumed by application
D. Ich weiss nicht

Solution



Answer B.

The window size is 4'000 B, namely here 4 packets.

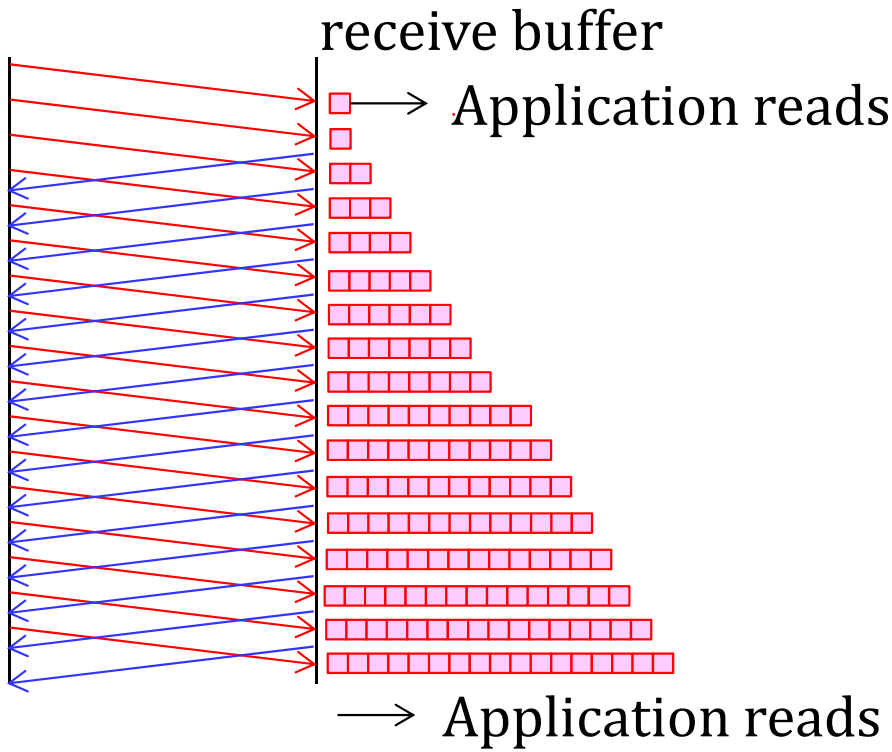
At time t_1 packets -1, 1, 2 and 3 are acked. The window is packets $[0\,;3]$. Packet 4 is outside the window and cannot be sent. It has to wait until the loss of packet 1 is repaired (at time t_2)

Sender also needs a buffer ("retransmission buffer"); its size is the window size.

Sliding Window is not sufficient to limit buffer size at receiver

Data that is received in-sequence remains in receive buffer until consumed by application (typically using a socket “read” or “receive”)

A slow application could cause buffer overflow



Window size = 4’000 bytes

One packet = = 1’000 bytes

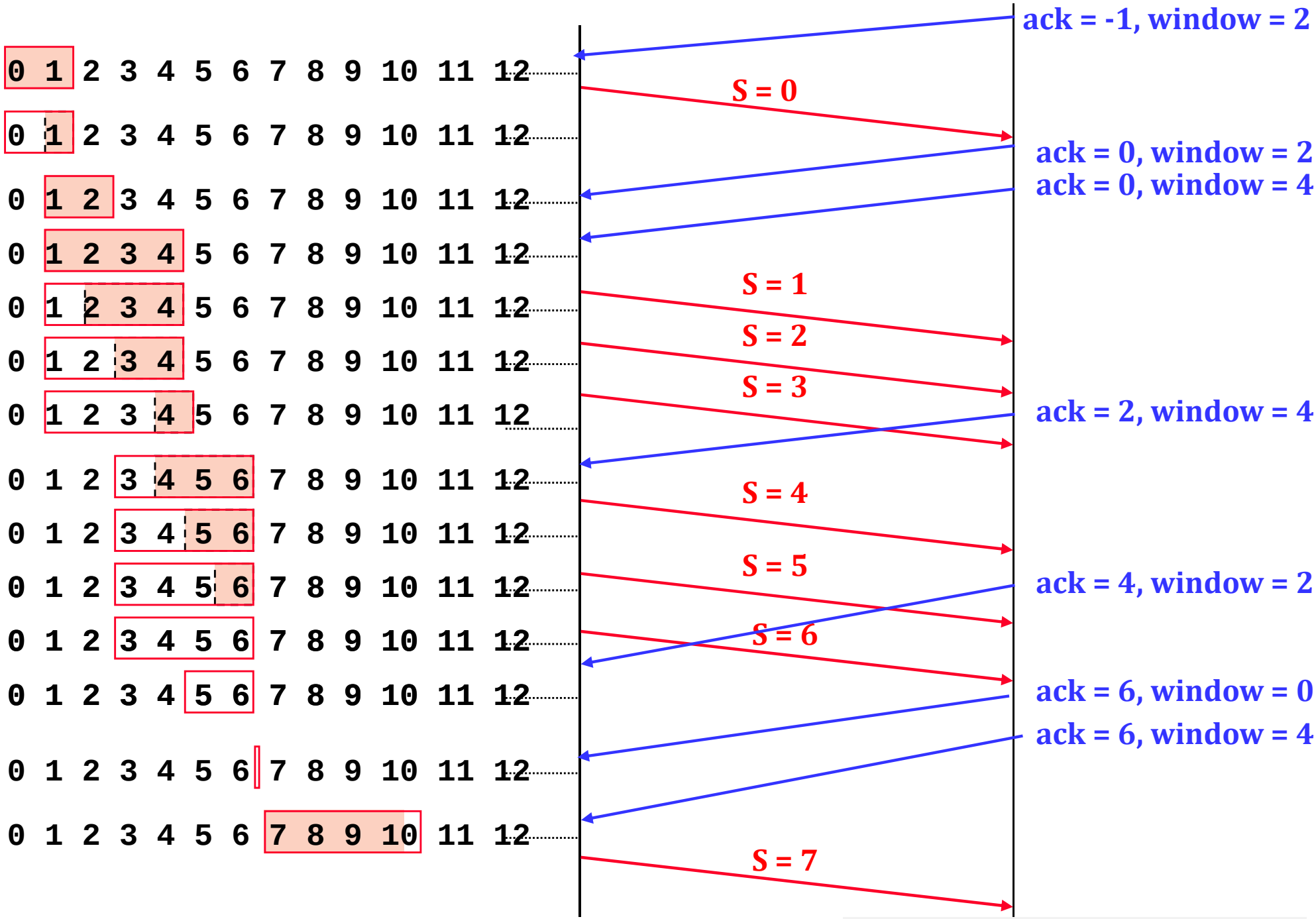
Window Flow Control is used to prevent receive-buffer overflow

TCP constantly adapts the size of the window by sending "window" advertisements back to the source.

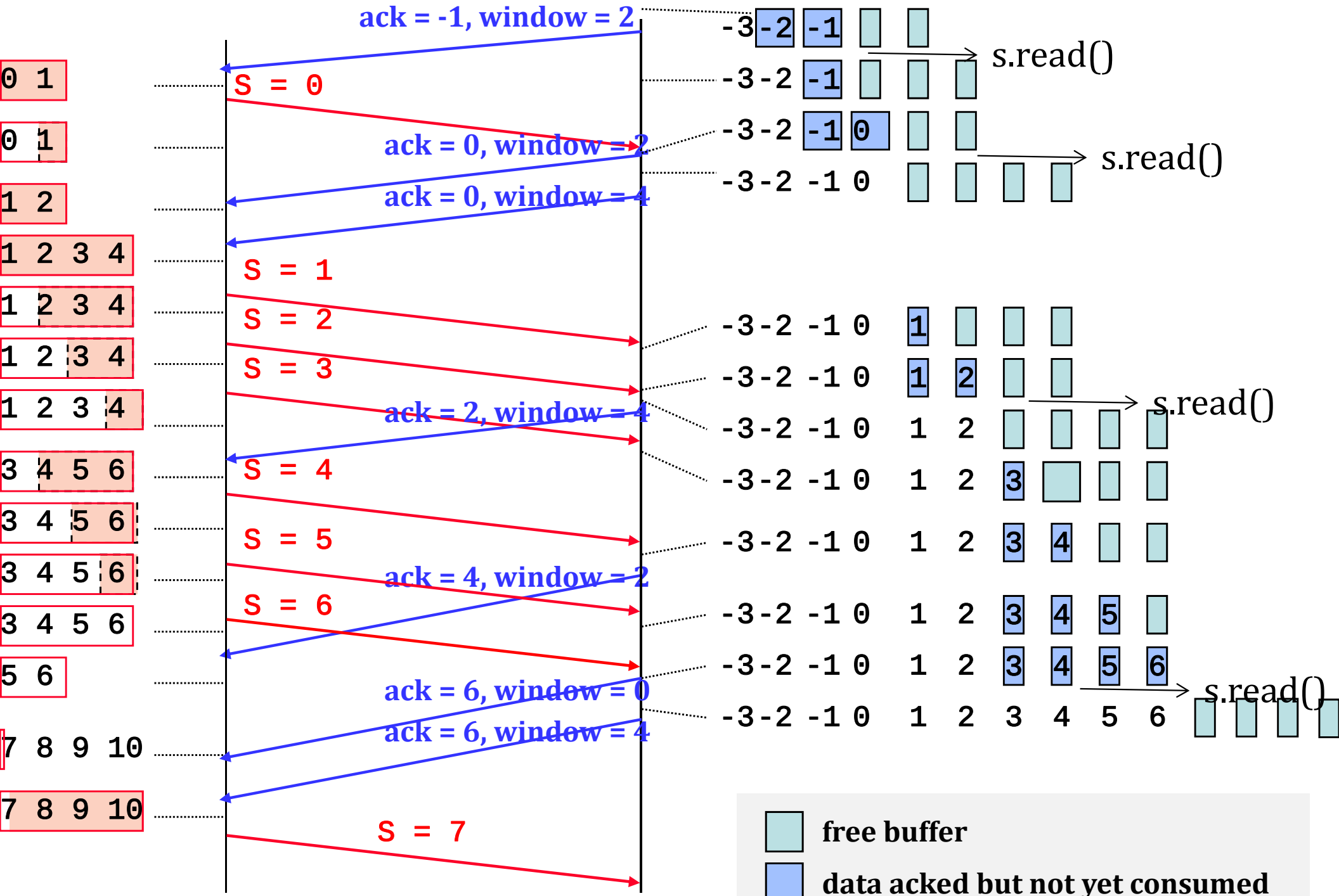
- Window size is set to available buffer size
- If no space in buffer, window size is set to 0

This is called "*Flow Control*" = adapt sending rate of source to speed of receiver

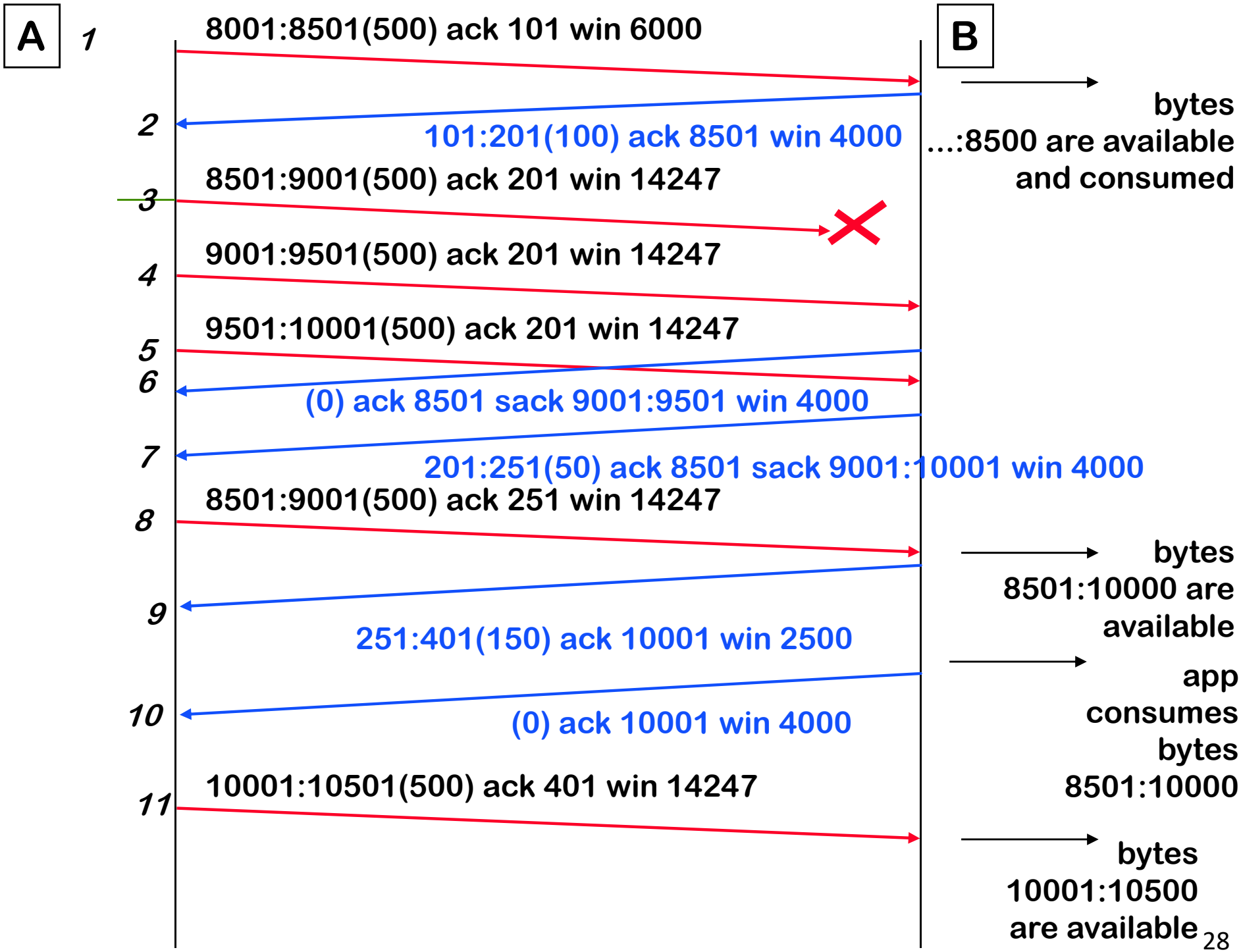
≠ Congestion Control (see later), which adapts rate of source to state of network



1 unit of data = 1'000 bytes
1 packet = 1'000 bytes


ack = -1, window = 2
S = 0
ack = 0, window = 2
ack = 0, window = 4
S = 1
S = 2
S = 3
ack = 2, window = 4
S = 4
S = 5
ack = 4, window = 2
S = 6
ack = 6, window = 0
ack = 6, window = 4
S = 7
s.read()
s.read()
s.read()
s.read()
free buffer
data acked but not yet consumed
1 unit of data = 1'000 bytes
1 packet = 1'000 bytes

TCP Basic Operation, Putting Things Together



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:

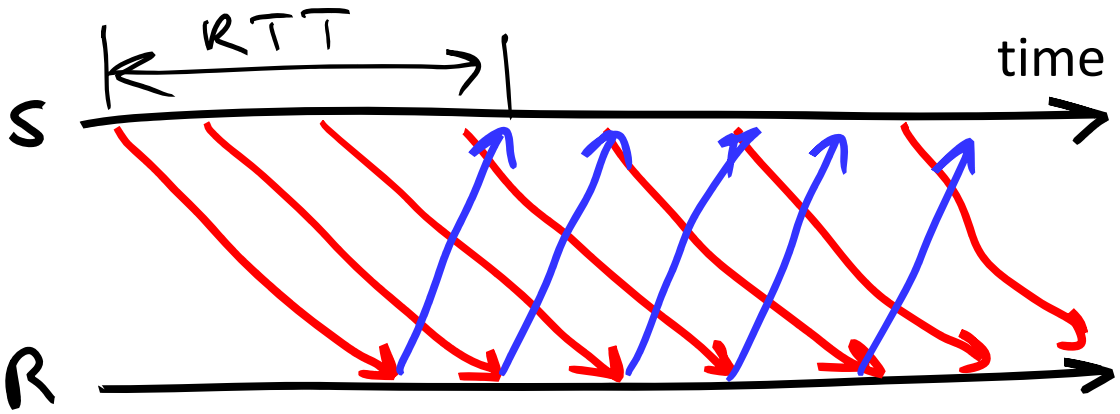
- firstByte”:”lastByte+1 “(“segmentDataLength”) ack” ackNumber+1 “win” offeredWindowSise. Note the +1 with ack and lastByte numbers.

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.

In the absence of loss, and on a link with capacity c packets per second, the window size required for sending at the maximum possible rate is…



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

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

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

D. None of the above

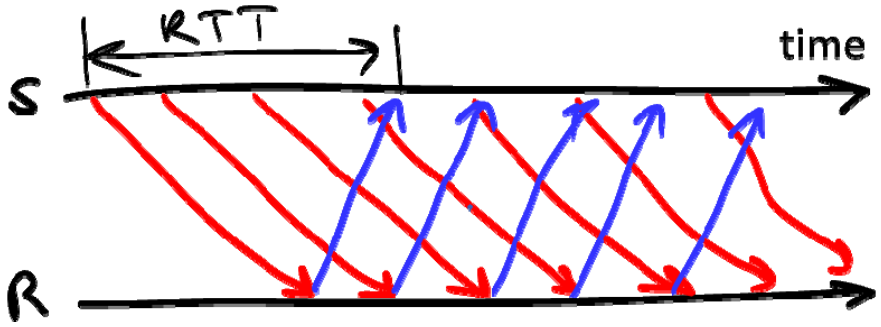
E. Non lo so

Solution

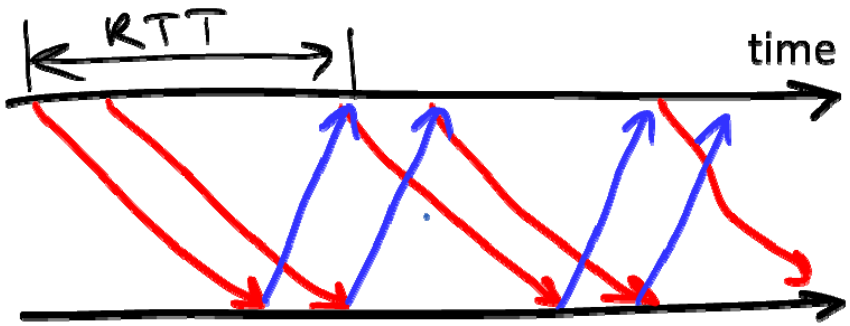
Answer A

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 Connections and Sockets

TCP requires that a connection (= synchronization) is opened before transmitting data

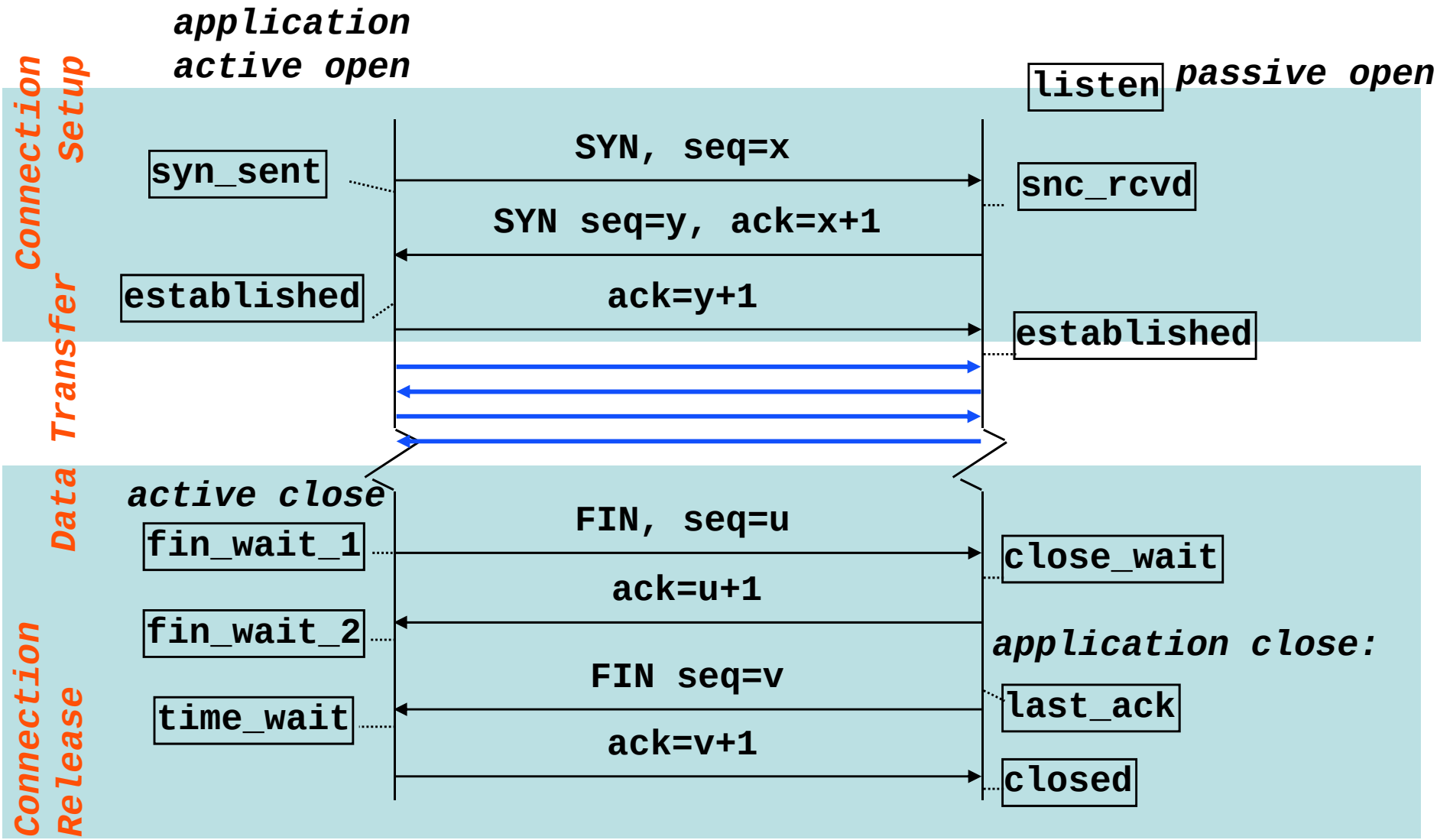
- Used to agree on sequence numbers and make sure buffers and window are initially empty

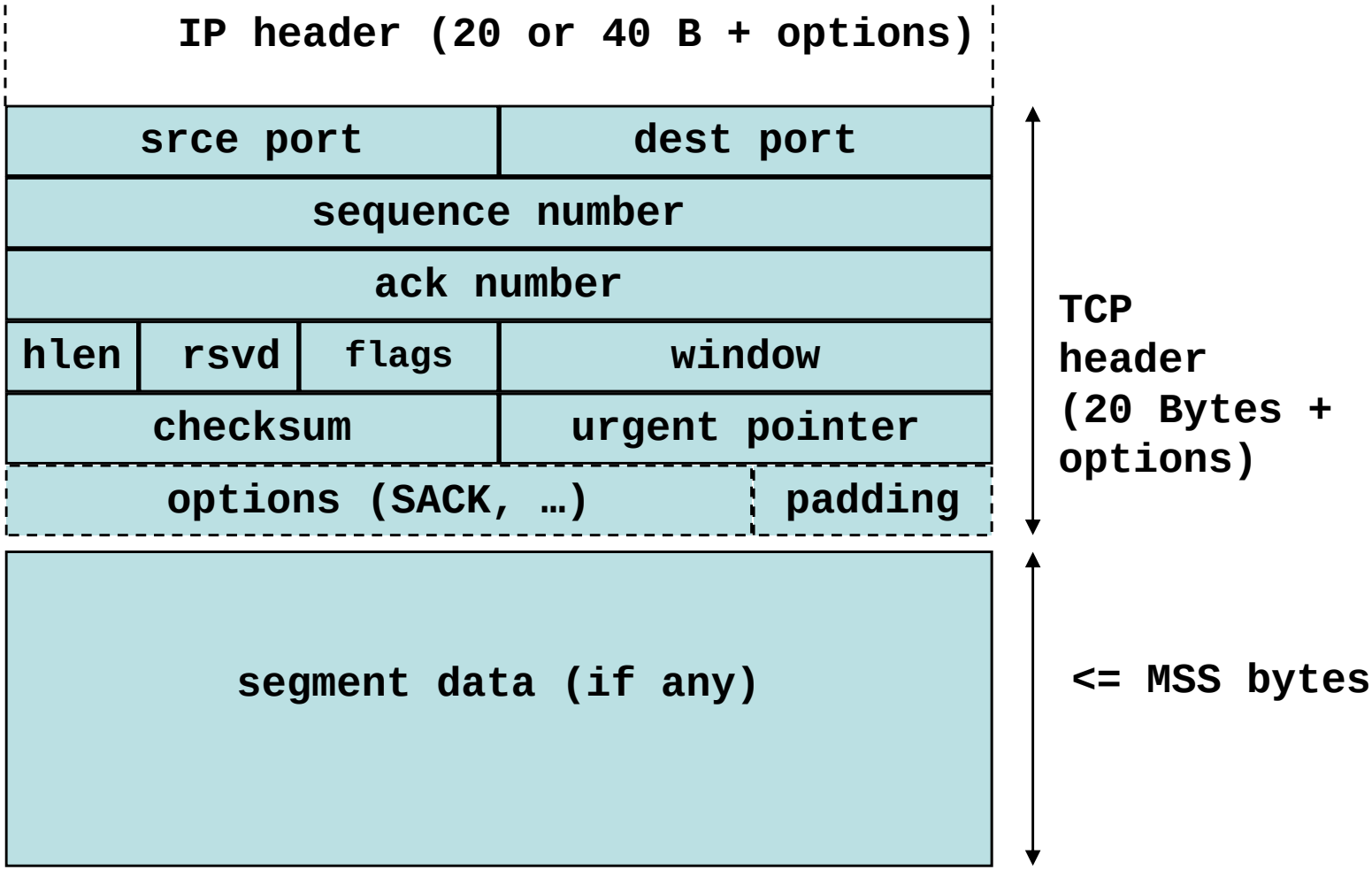
The next slide shows the states of a TCP connection:

- 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.
- The connection can be closed in a number of ways. The picture shows a graceful release where both sides of the connection are closed in turn.
- Remember that TCP connections involve only two hosts; routers in between are not involved.

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).

TCP Connection Phases





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

TCP Segment Format

The previous slide shows the TCP segment format.

- 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 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.
- SYN and FIN are used to indicate connection setup and close. They each consume 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 contain for example the Maximum Segment Size (MSS) normally in SYN segments (negotiation of the maximum size for the connection results in the smallest value to be selected) and SACK blocks.
- The checksum is mandatory
- The NS, CRW and ECN bits are used for congestion control (see congestion control module).

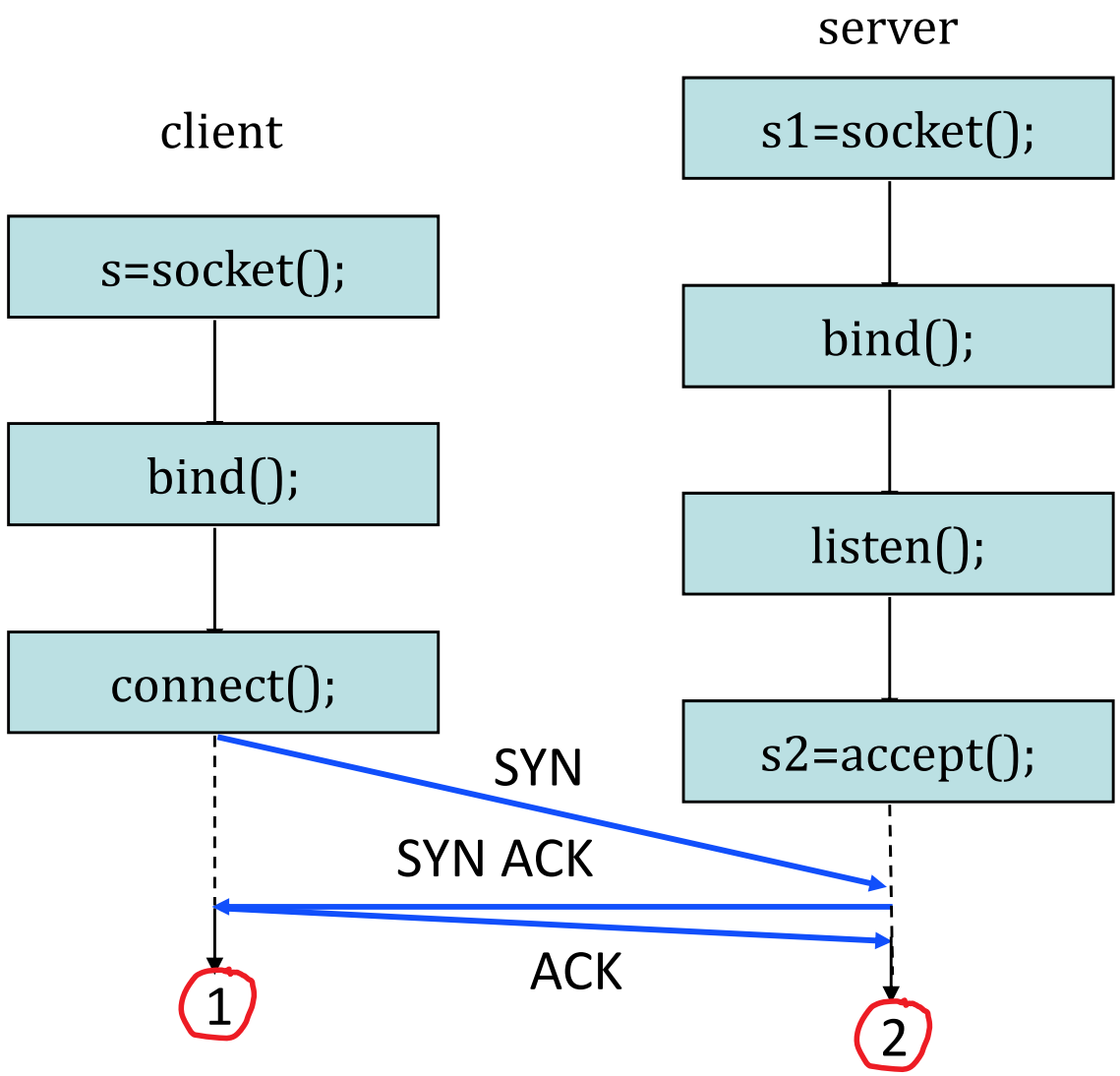
TCP Sockets

TCP is used by means of sockets, like UDP

However, TCP sockets are more complicated 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 (that side is called “client”)

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



The figure 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 number (=file descriptor) if successful;
socket(AF_INET6,…) creates an IPv6 socket

bind() associates the local port number with the socket

connect() associates the remote IP address and 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;

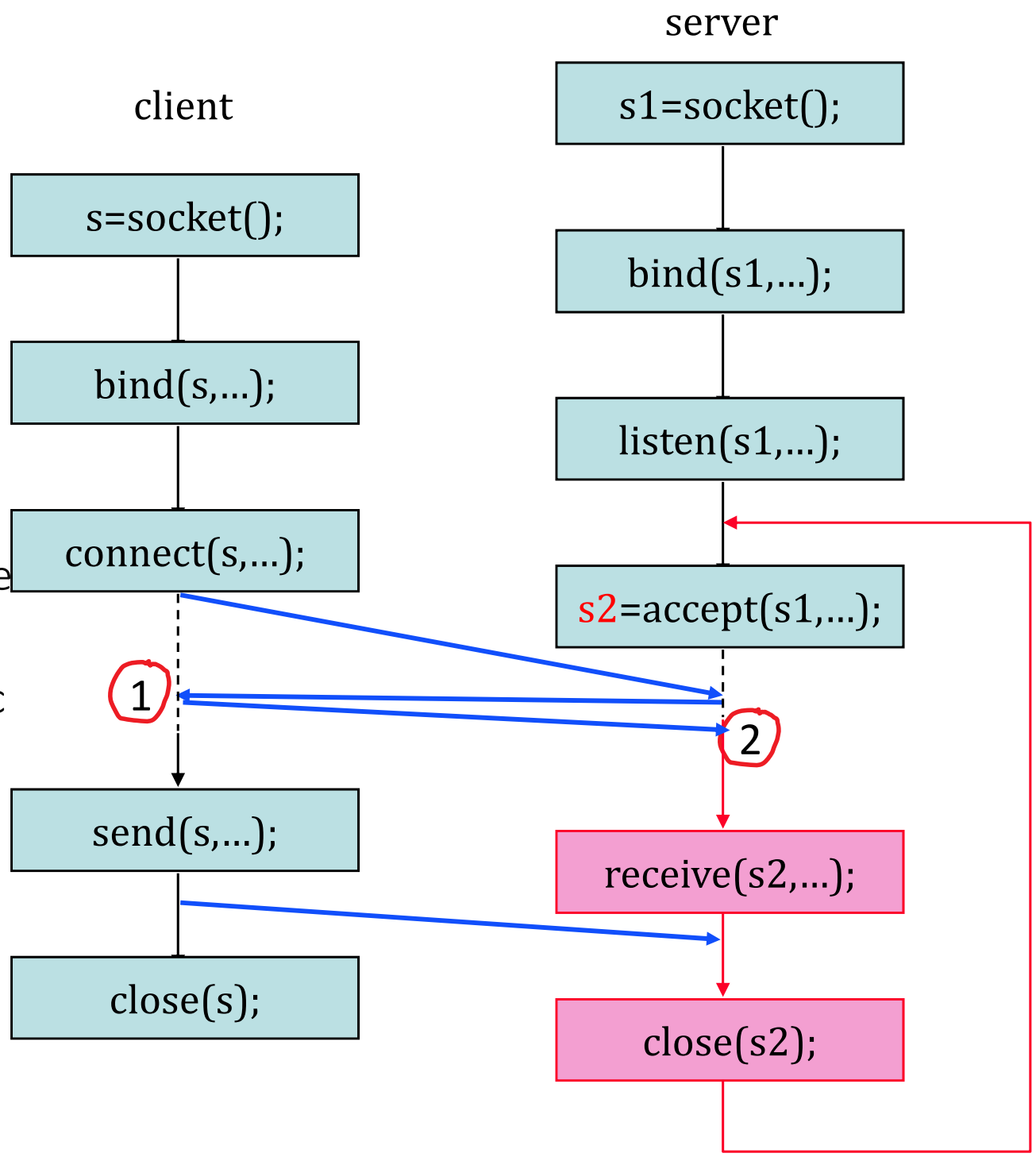
accept() blocks until a SYN packet is received for this local port number. It creates a new socket (in pink) and returns the file descriptor to be used to interact with this new socket

receive() blocks until one block of data is ready to be consumed on this port number. You must tell in the argument of receive how many bytes at most you want to read. It returns the number of bytes that is effectively returned and the block of data. It returns 0 when the connection was closed by the other end.

A New Socket is Created by Accept()

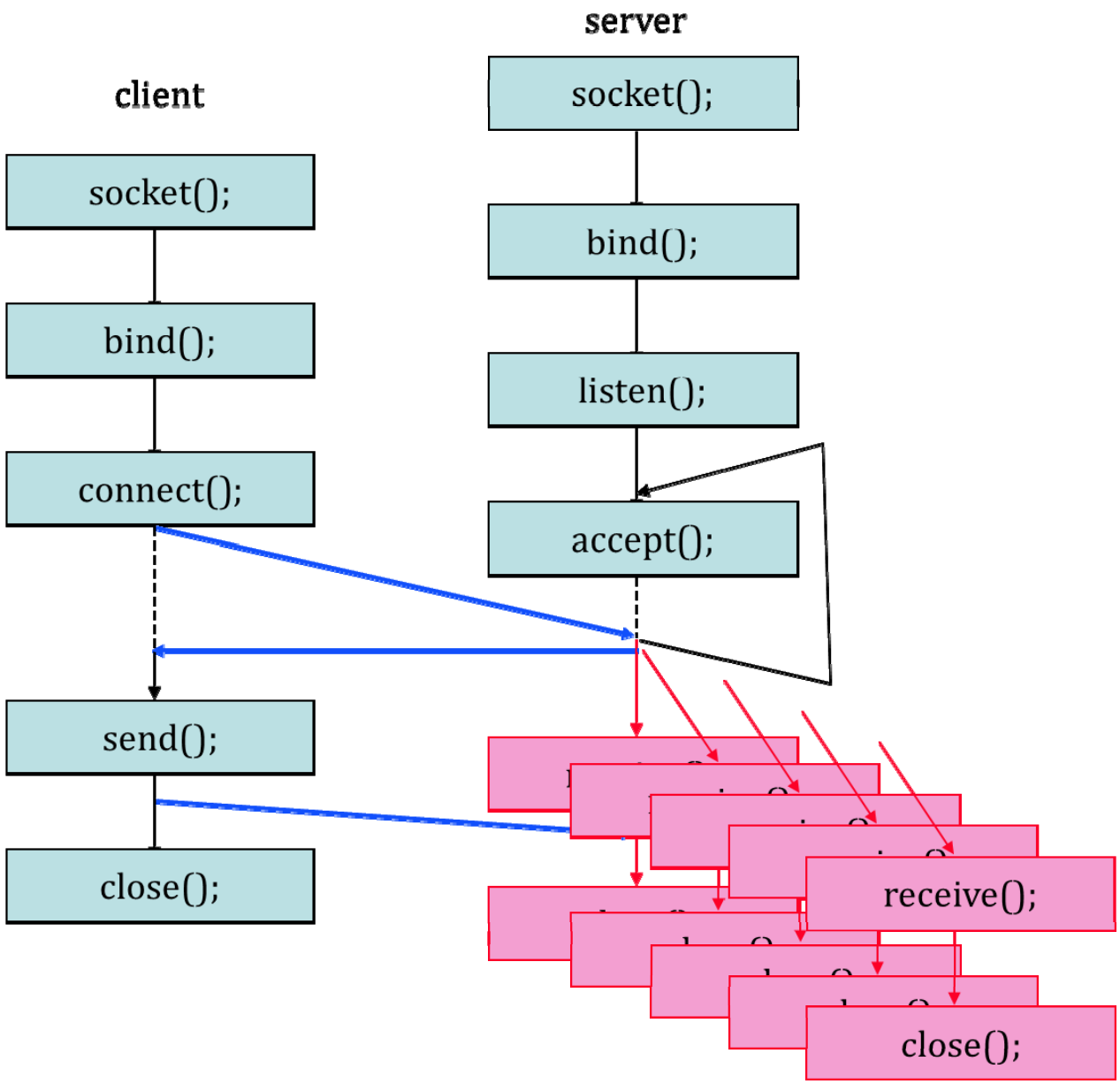
At 2, on server side, a new socket (s2) is created – will be used by server to send / receive data

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

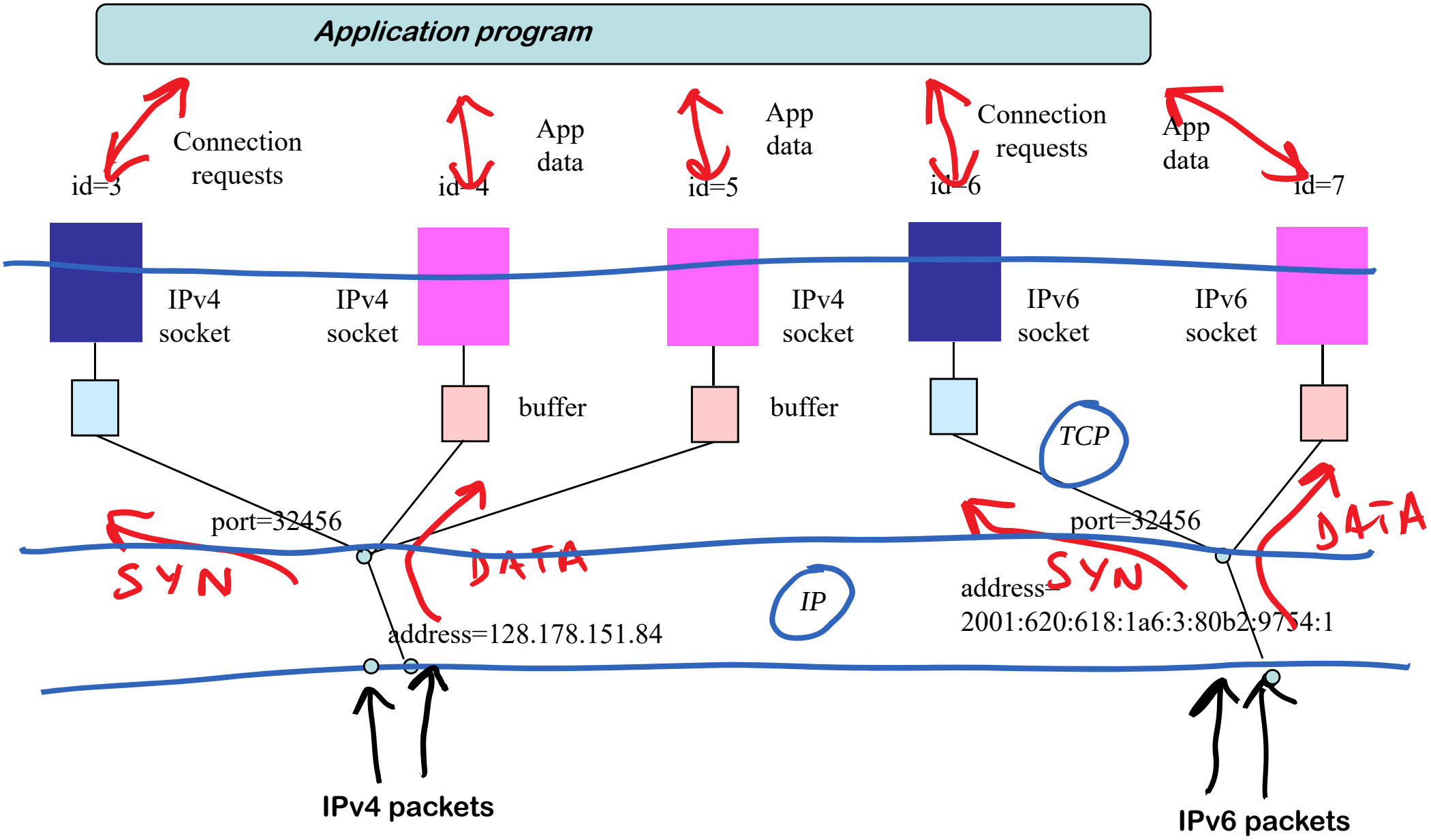


A More Typical Server

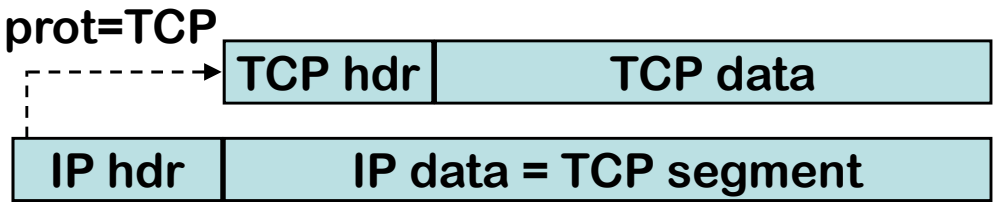
TCP Server typically uses parallel threads of execution to handle several TCP connections + to listen to incoming connections



How the Operating System views TCP Sockets



TCP Connections and Segments



TCP uses port numbers like UDP eg. TCP port 80 is used for web server. A TCP connection is identified by: **srce IP addr, srce port, dest IP addr, dest port.**

TCP-PDUs (called TCP segments) have a maximum size (called MSS). 536 bytes by default for IPv4 operation (576 bytes IPv4 packet), 1220 by default for IPv6 operation (1280 bytes IPv6 packets)

TCP, not the application, chooses how to segment data

TCP segments should not be fragmented at source

Modern OSs use TCP Segmentation Offloading (TSO) : the TCP code in the OS sends a possibly large block of data to the network interface card (NIC). Segmentation is performed in the NIC with hardware assistance (reduces CPU consumption of TCP).

TCP Offers a Streaming Service

Streaming Service: TCP accumulates bytes in send buffer until it decides to create a segment

independent of how application writes data

On receiver side, data accumulates in receive buffer until application reads it – data is not delineated, several small pieces of data sent by A may be received by B as a single block – and conversely, a single block sent by A may be received by B as multiple blocks.

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 is the streaming service a drawback ?

A. an app using http/1 with one TCP connection per object
B. an app using http/2 with one TCP connection in total
C. a real time streaming application that sends one new packet every msec
D. A and B
E. A and C
F. B and C
G. All
H. None
I. No lo sé

A TCP server is, by definition…

A. … an application program that does listen( ) and accept( ) on a TCP socket
B. … an application program that does receive( ) on a TCP socket
C. … an application program that does send( ) on a TCP socket
D. $\Delta\varepsilon\nu\ \xi\acute{\varepsilon}\rho\omega$

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 means, the loss of one packet delays all subsequent packets until the loss is repaired (whereas the application would prefer to skip the lost packet and receive the most recent one). Such an app should use UDP.

Answer A. A server program can send, receive or both.

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 / PMU 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 / PMU Streaming

Q. why ?

A. delay is important for interactive voice. Packet retransmission introduces too much delay in most cases. PMU streaming sends a new packet every 20 msec, better to receive latest packet than to repeat 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)

4. More TCP Bells and Whistles

TCP has been constantly improved since its inception in 1974. For example, problems to be solved are

- When to send a packet (application may write 1 byte into the socket; should TCP make one packet ?) -> Nagle's algorithm prevents making many small packets.
- When to send an ACK when there is no data to send in return ?
- When to increase the window size (silly window syndrome avoidance)?
- How to detect packet loss
- How to choose initial sequence numbers (SYN cookies) to avoid denial of service attack by SYN flooding
- How to avoid three way handshake

We will see only the last three in detail; see textbook section 4.3.3 for the ones we don't see here.

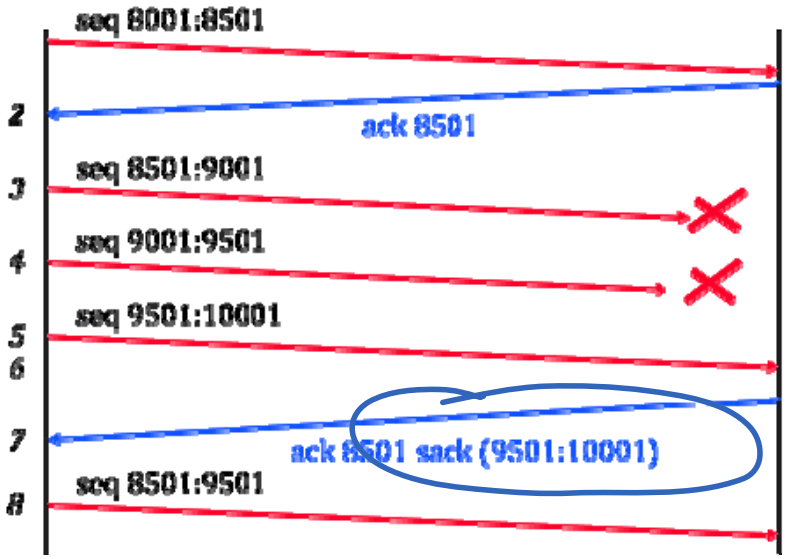
We could say that TCP declares a packet lost when a duplicate ACK is received with a SACK field. Is it a good idea ?

A. Yes because it is likely that there is some missing data

B. No as it may cause superfluous retransmissions (some data could simply be late -- out of order)

C. No because an ACK also could be lost

D. N'ouzhon ket.



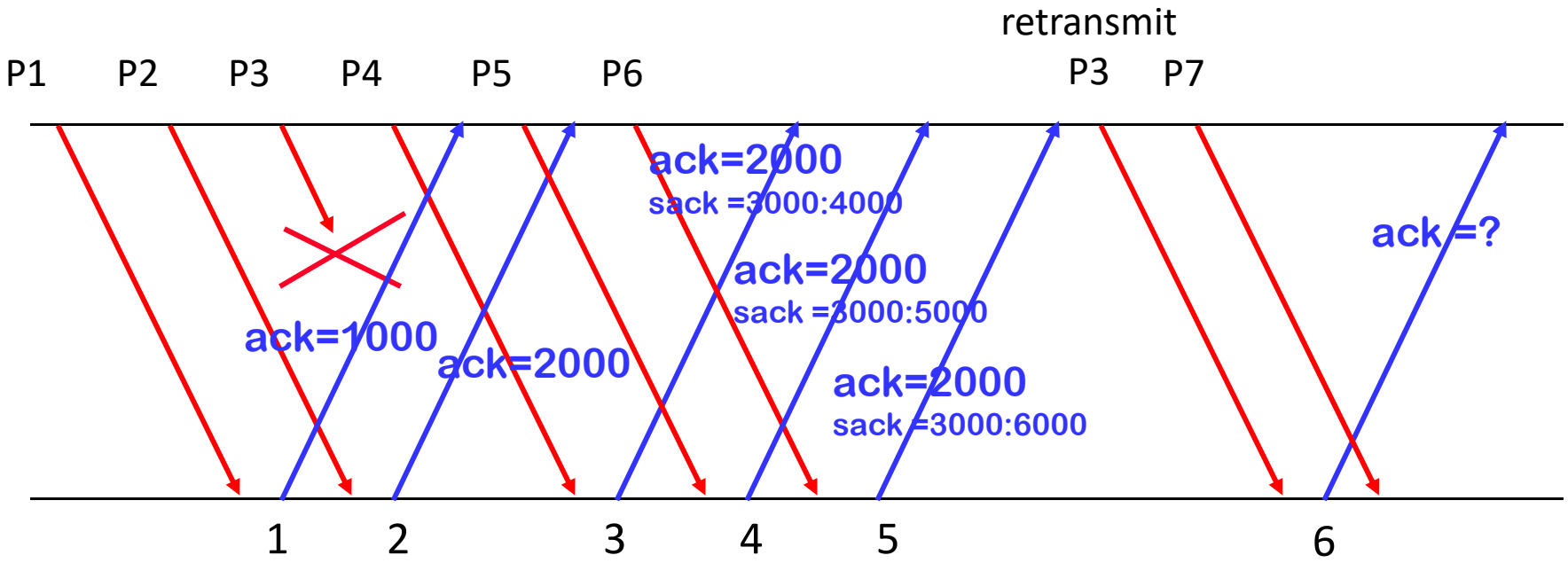
Fast Retransmit

Principle: when n duplicate ACKs are received, declare a loss

(Duplicate ACK = a TCP packet where the ACK value repeats a previously received ACK value)

The lost data is inferred from the SACK blocks

n = *TcpMaxDupACKs* is set by the Operating System (typically or 3)



all segments are 1000 bytes; TcpMaxDupACKs = 3

Loss Detection in TCP also uses timers

"Fast retransmit" detects most losses but not all

- bursts of losses are not detected
- last packets that are lost are not detected
- isolated packets that are lost are not detected

TCP also uses a retransmit timer, set for every packet transmission

- when one timer expires this is interpreted as a severe loss (loss of channel). All timers are reset and all data is marked as needing retransmission.

Round Trip Estimation

Why ? The retransmission timer must be set at a value slightly larger than the round trip time, but too much larger

What ? RTT estimation computes an upper bound RTO on the round trip time

How ?

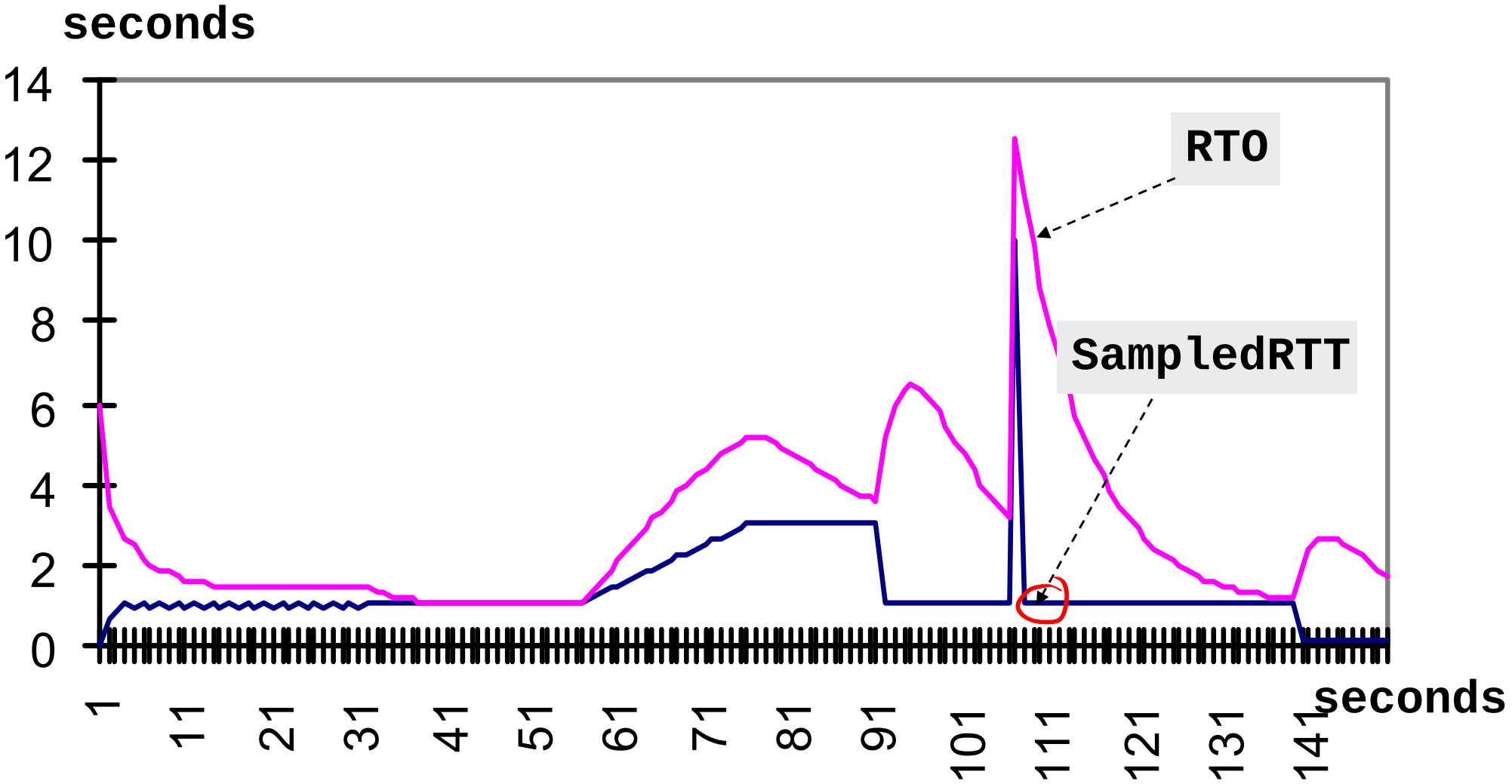
$$srtt := smoothed\ RTT$$
$$rtt := last\ measured\ RTT$$
$$rttvar := \ell^1 error\ bound$$
$$\alpha = \frac{1}{8}, \beta = \frac{1}{4}$$

$$rttvar = (1 - \beta) \times rttvar + \beta \times |srtt - rtt|$$

$$srtt = (1 - \alpha) \times srtt + \alpha \times rtt$$

$$rto = srtt + 4 \times rttvar$$

Sample RTO


seconds
14
12
10
8
6
4
2
0
RTO
SampledRTT
1
11
21
31
41
51
61
71
81
91
101
111
121
131
141
seconds

When does Fast Retransmit Fail ?

A. Extremely rarely, it is quasi-optimal
B. It fails to detect the loss extremely rarely, but it may often take a long time to detect.
C. When one of the last segments of an application layer block is lost, fast retransmit does not detect it.
D. It may often fail due to packet re-ordering
E. لا أعرف

Solution

Probably Answer C

When one of the last segments of a block is lost, fast retransmit cannot detect it since there is no packet transmission after the end of the block. The loss will be detected by timeout, which will take a long time. E.g.: query sent to a search engine.

Tail Loss Probe is a method that can be used to avoid such a problem: when the probe timeout (PTO, =2RTTs) expires, a "probe segment" is sent (a retransmission of a non ack'ed packet), in order to trigger new acks and fast retransmit.

Packet re-ordering is not rare but is usually very small (less than 1 ms) as it is due to load balancing inside machines. Equal cost multipath avoids to send packets of a TCP connection on different paths when per flow load balancing is enabled.

RACK (Recent ACK)

RACK is an alternative to Fast Retransmit. Bases retransmission decisions on timings, not on sequence numbers.

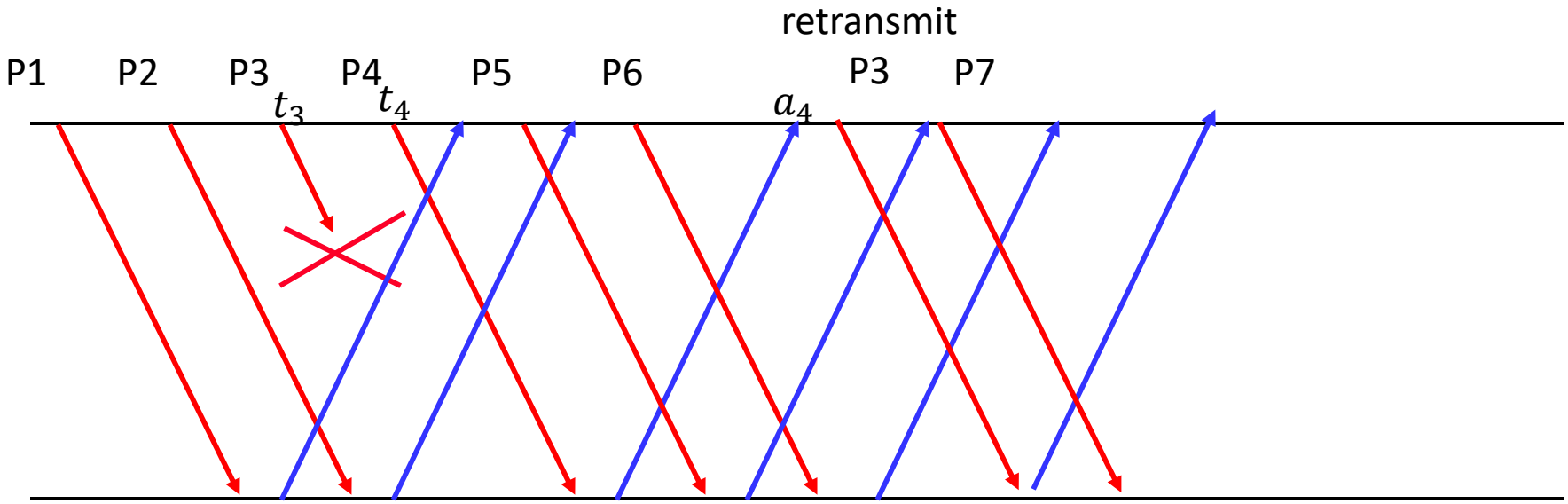
Assumes sender records all packet transmission times.

Sender declares a packet with send time t_1 as lost whenever an ack is received for a packet sent at time $t_2 > t_1$ +reo_wnd

Furthermore, a RACK-timeout = RACK-RTT + reo_wnd is started at every packet transmission; packet is declared lost if timer expires.

RACK-RTT is the RTT measured for the last acked packet

reo_wnd (reordering window) is adapted based on re-ordering statistics. In example, reo_wnd = 1msec.



Assume $t_4 - t_3 > 1$msec.

At time a_4 , P4 is acked (with a SACK block), but no ack for P3 was received; P3 is declared lost because we assume here that re-ordering is at most by 1 msec.

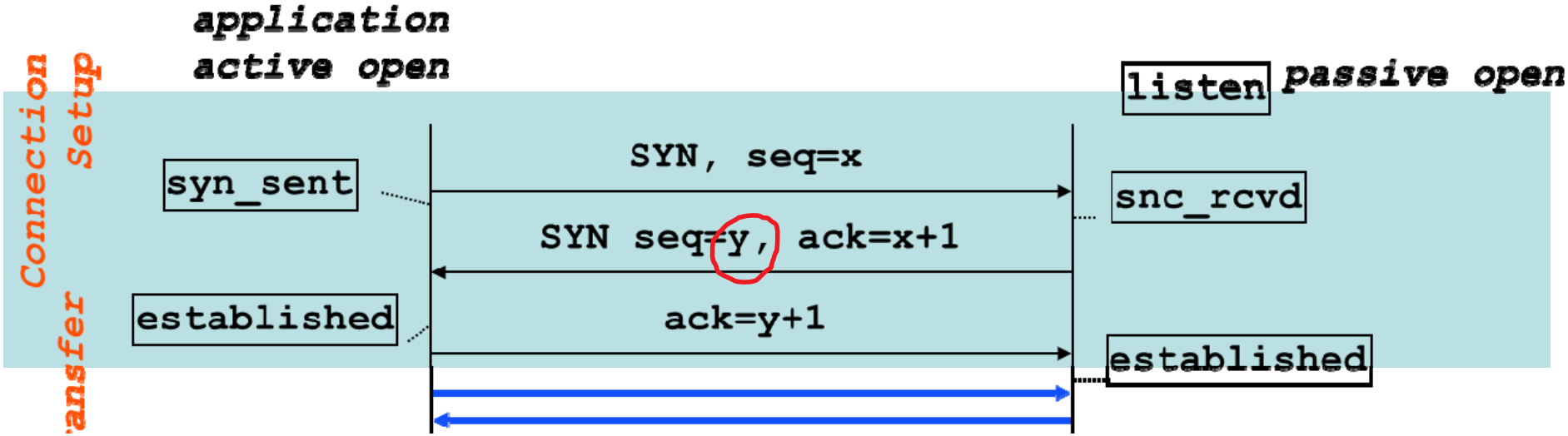
SYN Cookies

Why ? mitigate impact of SYN flood attack: lots of bogus SYN packets from invalid source addresses sent to a server.

When a TCP server receives a SYN packet, it should remember the details of the connection (source IP address, port, seq number) and stores them in kernel space. If SYNs are bogus, they remain stored until timeout occurs. The listen queue is full and legitimate SYN packets are dropped. Server is out !

What ? with SYN cookies, TCP server does not keep state information after receiving a SYN packet. State is encoded in the Seq Number field, using a cryptographic function and returned to client (the “cookie”). If SYN is valid, 3rd ack contains the state in the ACK.

SYN Cookies Encode State in Seq of SYN ACK



State (called *SYN cookie*) is written in y

y = (5 bit) t mod 32 ||(3 bits) MSS encoded in SYN ||(24 bits) cryptographic hash of secret server key, of t (timestamp) and client IP address and port number, the server IP address and port number.

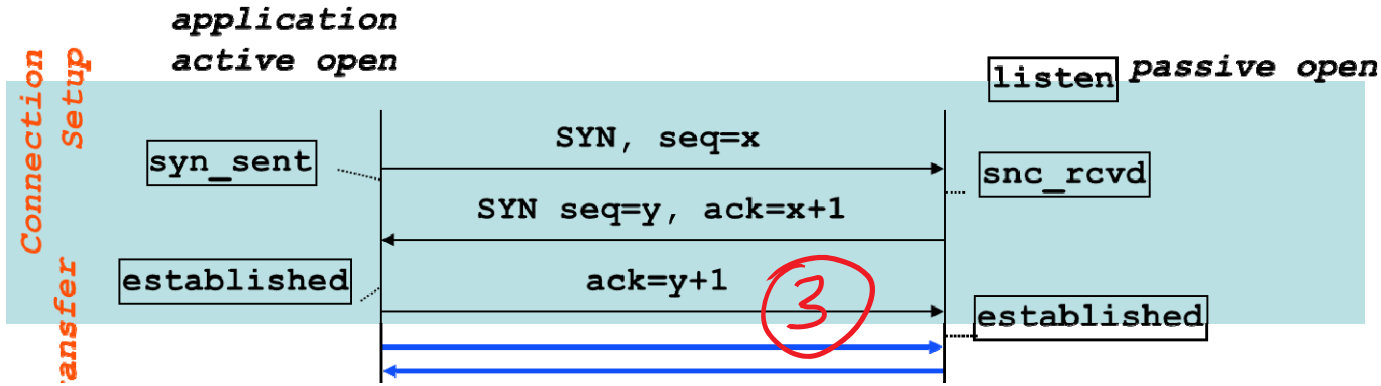
Server drops state and sends SYN cookie=y in SYN ACK. Client sends ack=y+1. Server verifies that hash is valid; if so creates socket, using the MSS recovered from the cookie.

If SYN was bogus, no ack follows and damage is reduced to loss of computation but no loss of listen queue availability.

If the ACK (3) is never sent, a server that does *not* implement SYN cookies will

1) retransmit SYN ACK

2) keep state information until timeout occurs



A. 1

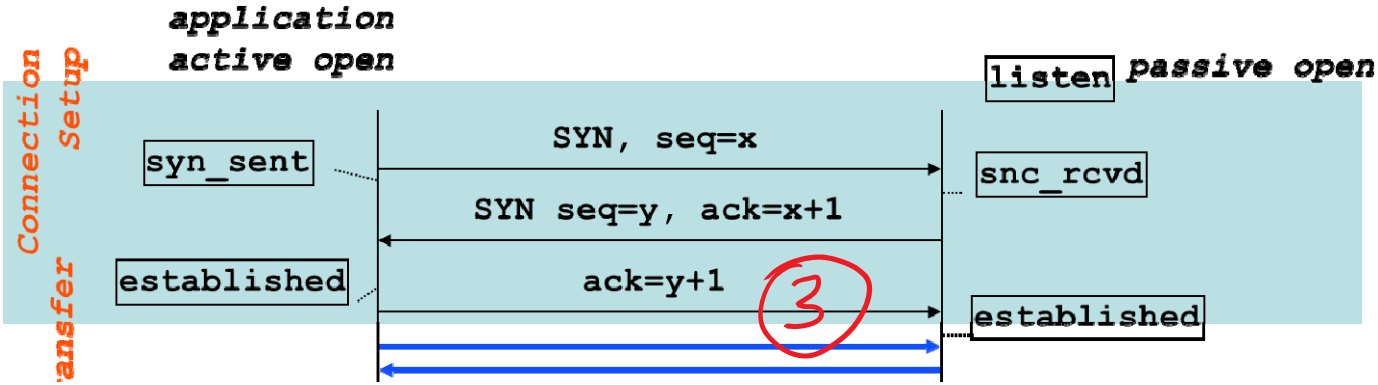
B. 2

C. 1 and 2

D. None

E. 我不知道

Solution



Answer C. Server that does not implement SYN Cookies: When the server sends the SYN-ACK packet, it does all the usual stuff done by TCP to test whether a packet containing some data is lost. Note that this packet contains no data but it is treated as if it would (this is why the ACK(3) number is y+1 and not y). If SYN-ACK is lost, server receives no ack and retransmits the SYN-ACK.

If the server implements SYN Cookies, it keeps no state information after sending the SYN-ACK; therefore, if the SYN-ACK is lost or if the ACK (3) is lost, the server does not retransmit.

In most cases, the client application sends data when the SYN ACK is received so even if the ACK(3) is lost, the next segment of data serves as an ACK and the application on the client side will detect the loss. Applications that do not have this property are for example SSH or mySQL. Such applications hang on the client side if ACK(3) is lost. Hopefully such apps implement a timeout to detect such deadlocks.

With SYN cookies, the response time of SYN-ACK is...

A. Larger than without SYN cookies
B. Smaller than without SYN cookies
C. The same
D. I weiss nid

Solution

Answer A. The SYN-cookie-enabled server must perform a verification of the cryptographic hash and create the state, which is more time consuming than without syn cookies.

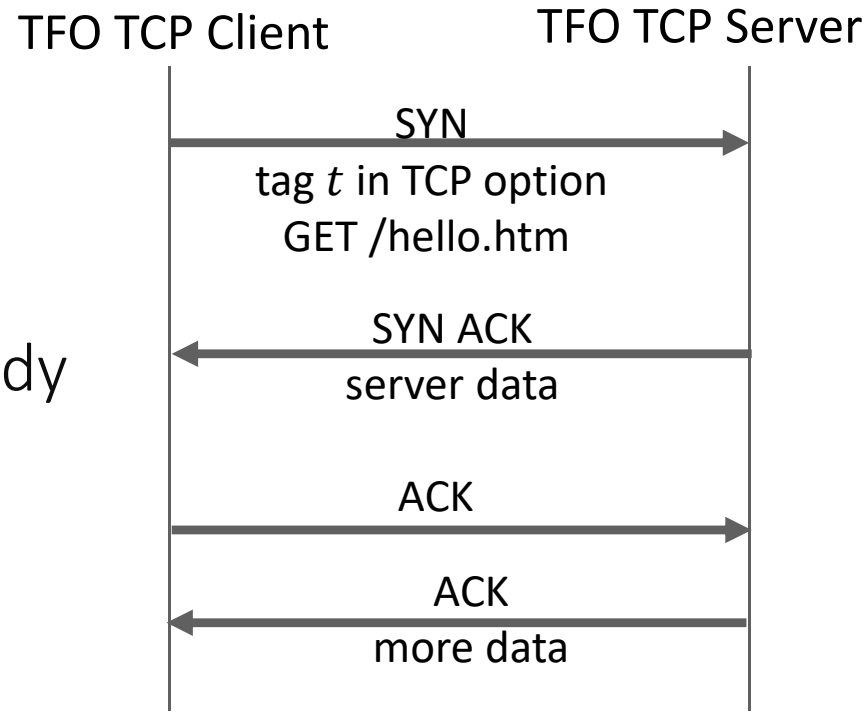
TCP Fast Open (TFO)

Why ? Avoid latency of 3-way handshake when opening repeated connections.

How? In first SYN_ACK, TCP client receives and caches a cookie that contains authentication tag $t = MAC(k, c)$ computed by server with secret key k (unknown to client) and client IP address c.

Client can send data in SYN packet.

When receiving SYN and tag t, server knows that this client is a real one and not spoofed. Server can send data already in SYN-ACK.



MAC= Message Authentication Code

5. Error Recovery

We have seen *how* TCP repairs losses

We now discuss *why* this is so, and sometimes why it is not so

The Layered Model Transforms Errors into Packet Losses

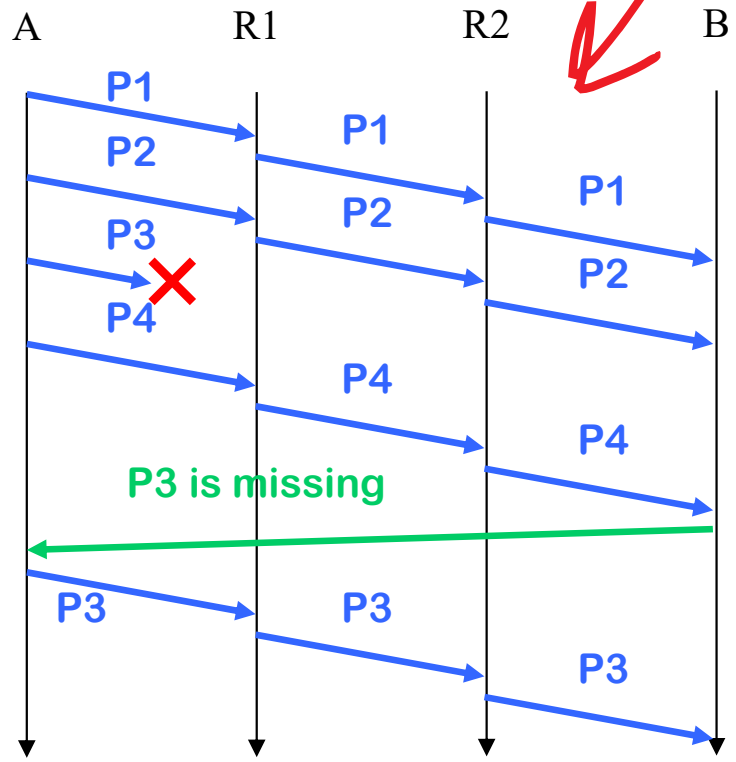
Packet losses occur due to

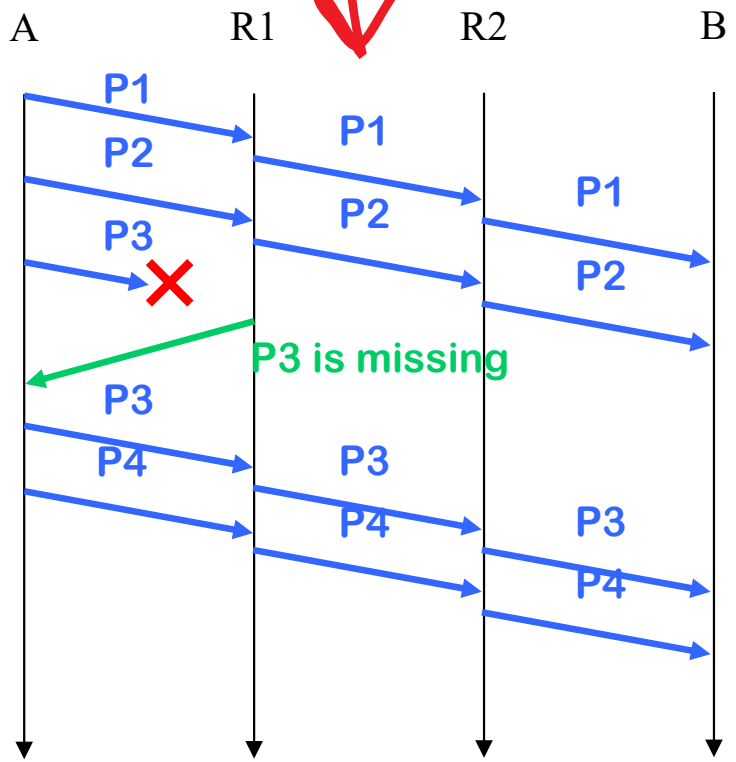
- error detection by MAC
- *buffer* overflow in bridges and routers
- Other exceptional errors may occur too

Therefore, packet losses must be repaired.

This can be done either

- *end to end* : host A sends 10 packets to host B. B verifies if all packets are received and asks for A to send again the missing ones.
- or hop by hop



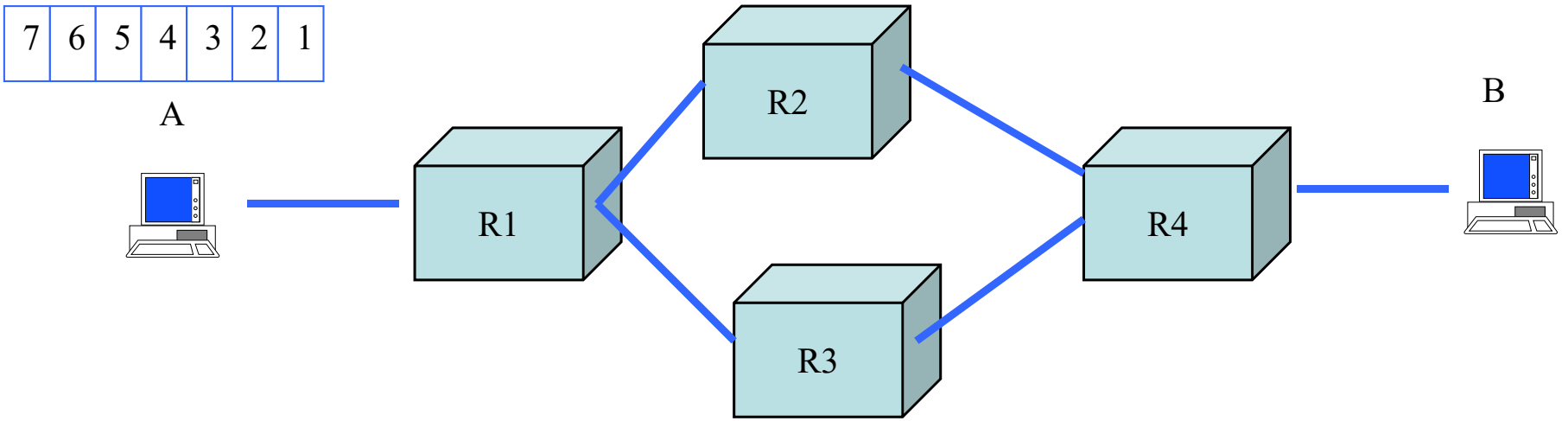


The Case for End-to-end Error Recovery

The end-to-end philosophy of the internet says: keep intermediate systems as simple as possible

IP packets may follow parallel paths, this is incompatible with hop-by-hop recovery.

- R2 sees only 3 out of 7 packets but should not ask R1 for re-transmisison



The Case for Hop-By-Hop Error Recovery

There are also arguments in favour of hop-by-hop strategy. To understand them, we will use the following result.

Capacity of erasure channel: consider a channel with bit rate R that either delivers correct packets or loses them. Assume the loss process is stationary, such that the packet loss rate is $p \in [0\,;1]$. The capacity is $R(1-p)$ packets/sec.

This means in practice that, for example, over a link at 10Mb/s that has a packet loss rate of 10% we can transmit up to 9 Mb/s of useful data.

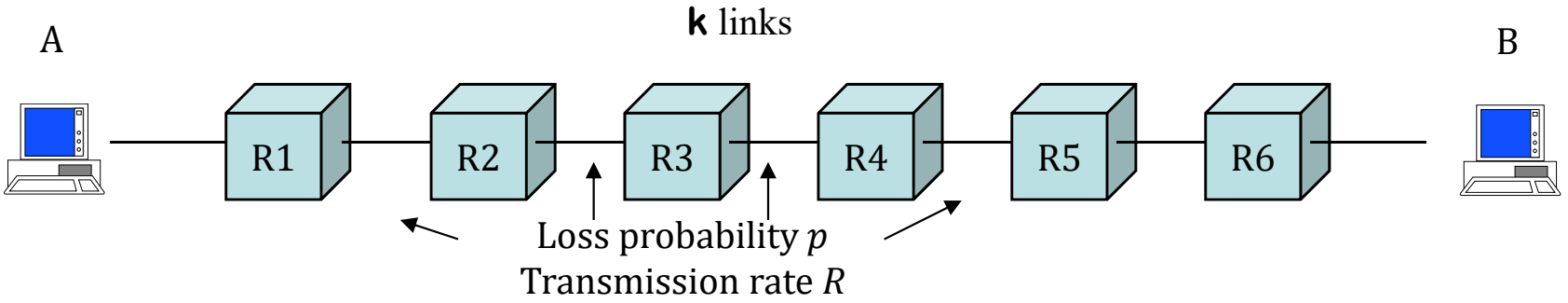
Furthermore, this capacity is obtained by a scheme (such as TCP) which retransmits lost packets.

The Capacity of the End-to-End Path

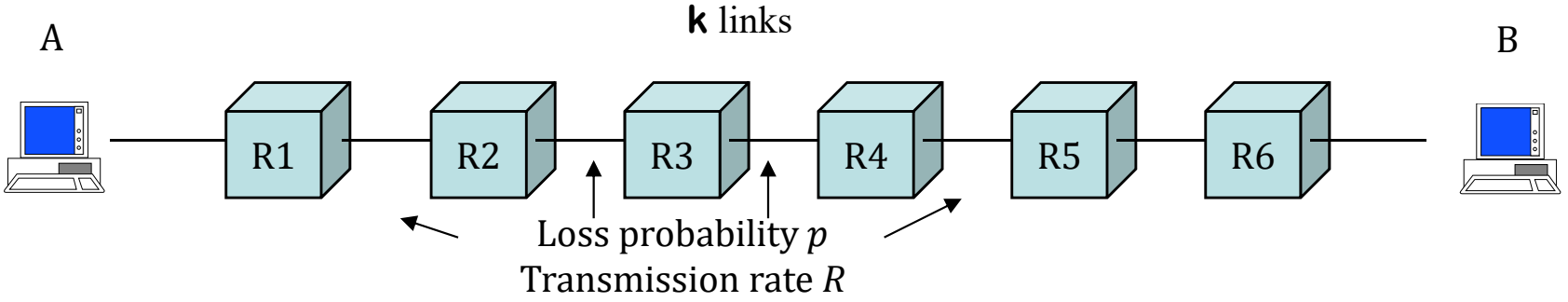
We can now compute the capacity of an end-to-end path with both error recovery strategies.

Assumptions: same packet loss rate p on k links; same nominal bit rate R. Losses are independent.

Q. compute the capacity with end-to-end and with hop by hop error recovery.



The capacity C_1 with hop-by-hop error recovery is …



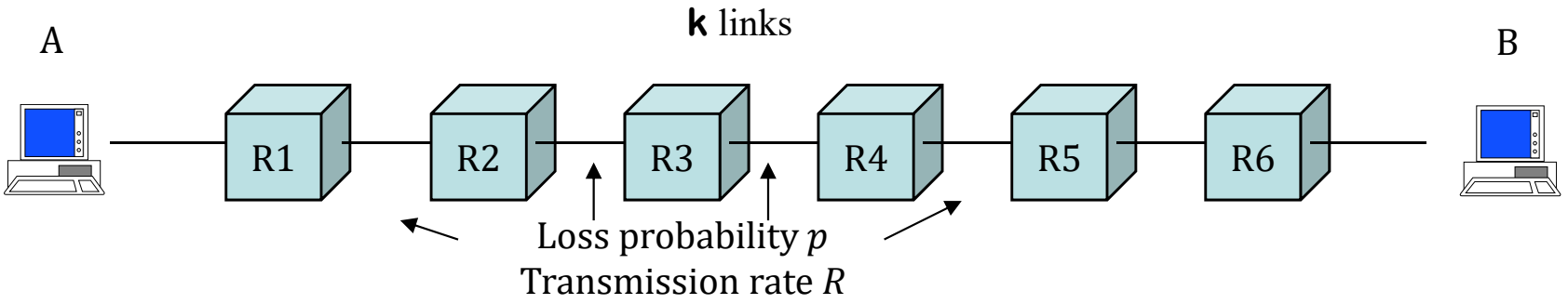
A. $C_1 = R(1-p)^k$

B. $C_1 = R(1-p)$

C. $C_1 = R(1-kp)$

D. Não sei

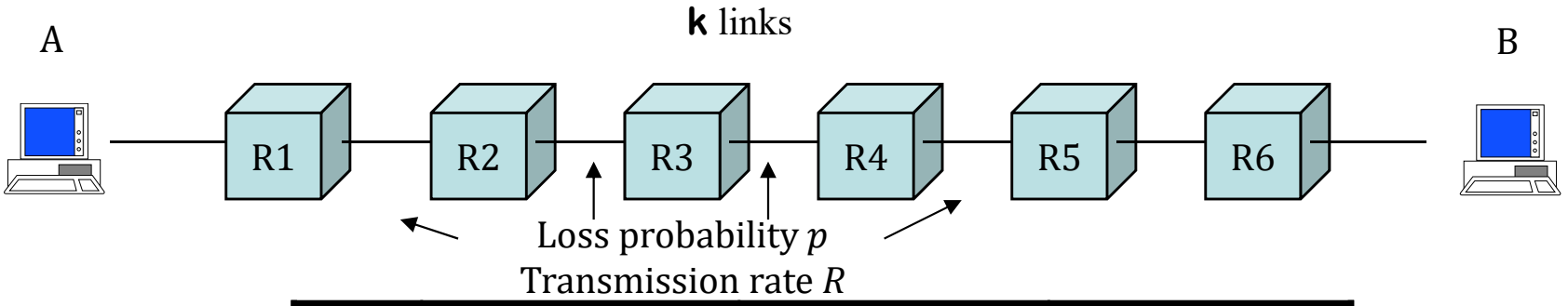
The Capacity of the End-to-End Path



1. With hop-by-hop error recovery:
   Capacity of one hop after error recovery is $R(1-p)$
   The capacity of the end-to-end path is also $R(1-p)$ [the capacity of a concatenation of loss-less segments is the min of the capacity of each segment]

2. With end-to-end recovery
   The probability that a packet is not lost is $(1-p)^k$
   The probability that a packet is lost is $q = 1-(1-p)^k$
   The capacity of the path is $R(1-q) = R(1-p)^k$

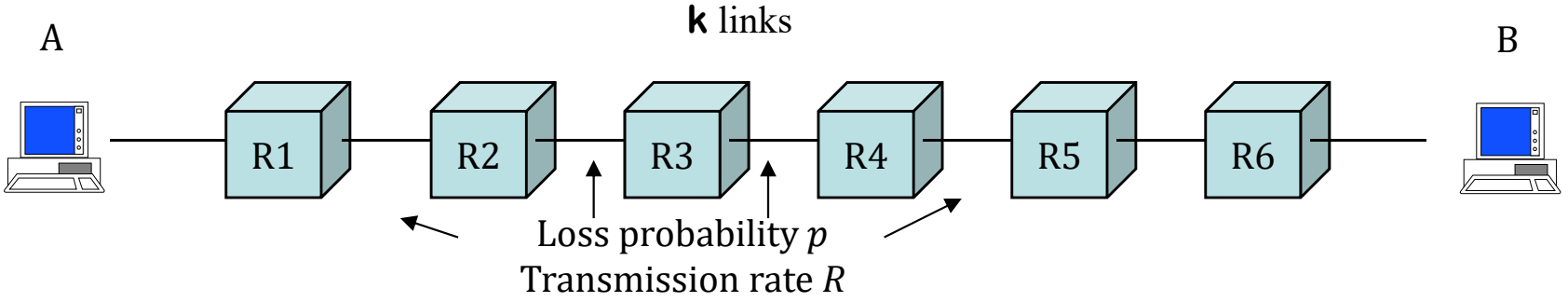
End-to-end Error Recovery is Inefficient when Packet Error Rate is high



k	Packet loss rate	C_1 (end-to-end)	C_2 (hop-by-hop)
10	0.05	0.6 R	0.95 R
10	0.0001	0.9990 R	0.9999 R

Q. How can one reconcile the conflicting arguments for and against hop-by-hop error recovery ?

End-to-end Error Recovery is Inefficient when Packet Error Rate is high



k	Packet loss rate	C_1 (end-to-end)	C_2 (hop-by-hop)
10	0.05	0.6 R	0.95 R
10	0.0001	0.9990 R	0.9999 R

Q. How can one reconcile the conflicting arguments for and against hop-by-hop error recovery ?

A. Repair losses locally only on links with high loss rates, i.e. wireless

Where is Error Recovery located in the TCP/IP architecture ?

The TCP/IP architecture assumes that

1. The MAC layer provides error—free packets to the network layer
2. The packet loss rate at the MAC layer (between two routers, or between a router and a host) must be made very small. It is the job of the MAC layer to achieve this.
3. Error recovery must also be implemented end-to-end.

Thus, packet losses are repaired:

At the MAC layer on lossy channels (wireless)
WiFi repairs losses with a repetition mechanism similar to TCP but simpler, window = 1 packet

In the end systems (transport layer by TCP or application layer if UDP is used).

Conclusion

The transport layer in TCP/IP exists in two flavours

- reliable and stream oriented : TCP
- unreliable and message based: UDP

TCP uses : sliding window and selective repeat; window flow control; congestion control – see later

TCP offers a strict streaming service and requires 3 way handshake

Other transport layer protocols exist but their use is marginal: e.g. SCTP (reliable + message based)

Some application layer frameworks are a substitute to TCP for some applications: e.g. QUIC (reliable and “message” based – see Appli), websockets (see lab).