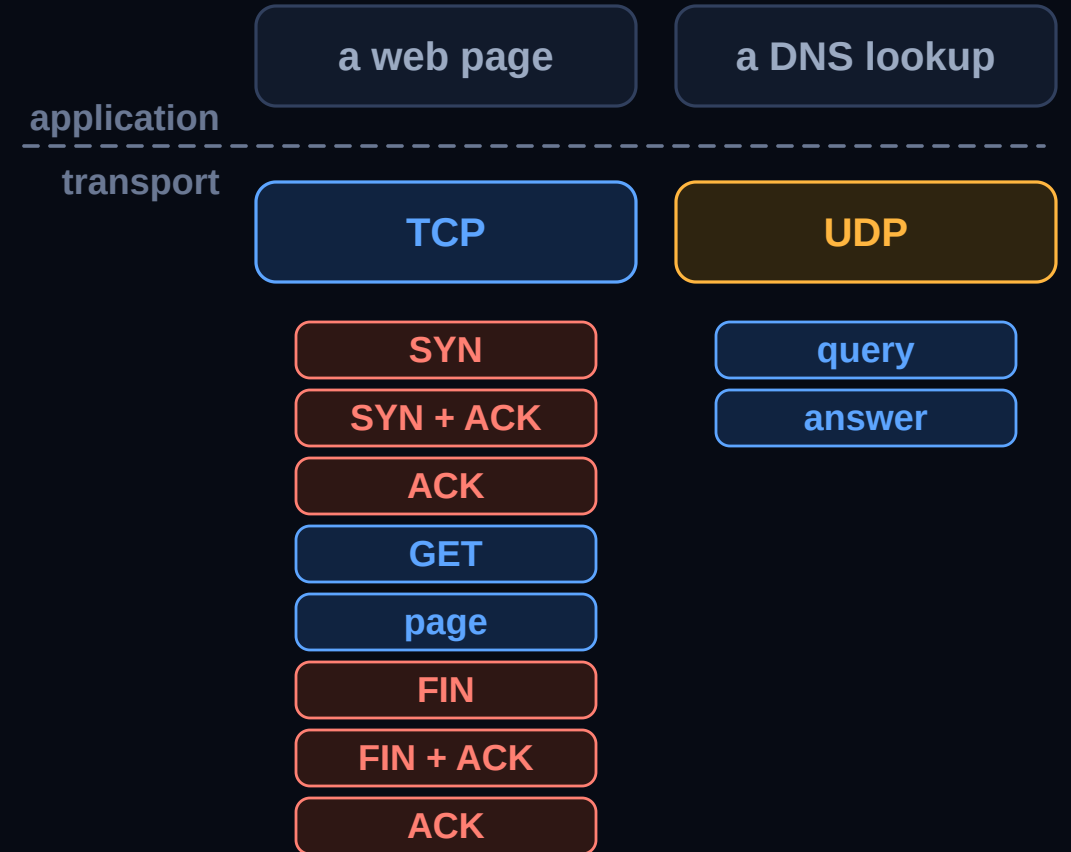


The Real TCP and UDP

The last lectures built the mechanisms. This one looks at the real protocols that carry them: how TCP opens, runs and closes a connection, what UDP leaves out, and why the newest transport, QUIC, is built on UDP.

Yaxiong Xie



Two real transport protocols.

One keeps a connection, with a beginning and an end; the other keeps nothing.

What has this chapter built so far?

reliable transfer

- checksum
- ACKs and sequence numbers
- timers, adaptive timeout
- windows: Go-Back-N, selective repeat
- duplicate ACKs, fast retransmit

congestion control

- cwnd
- AIMD
- slow start
- ECN

what those lectures assumed

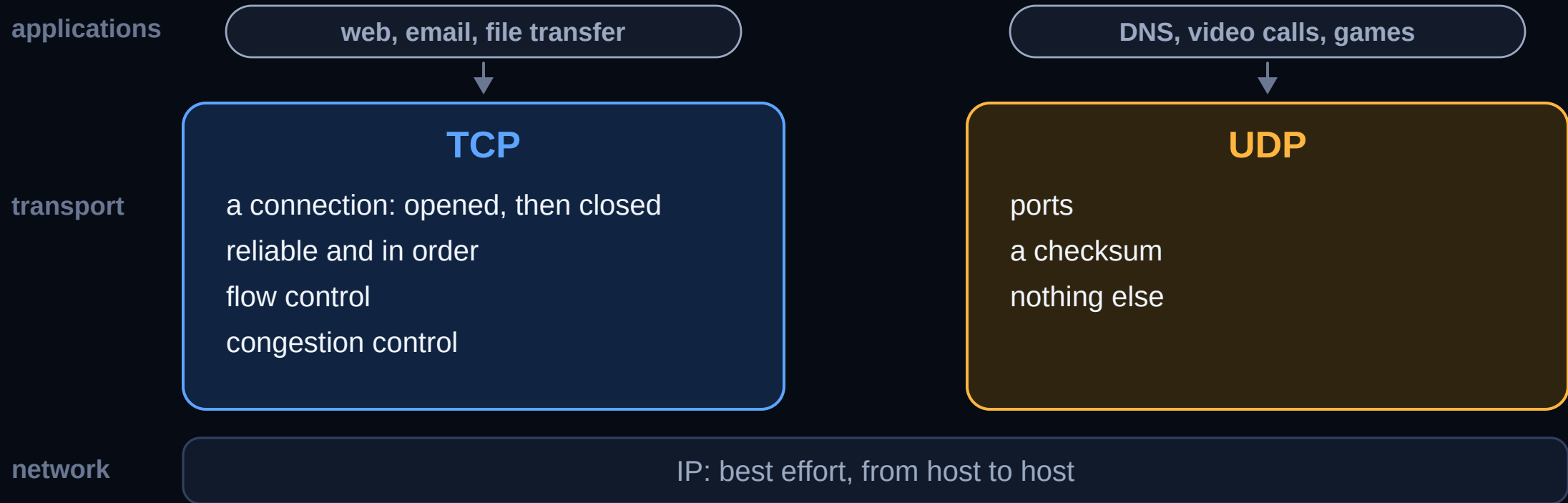
A and B already connected, never closed

packets numbered from 0

B always had room

We built the pieces; a real protocol cannot make those three assumptions.

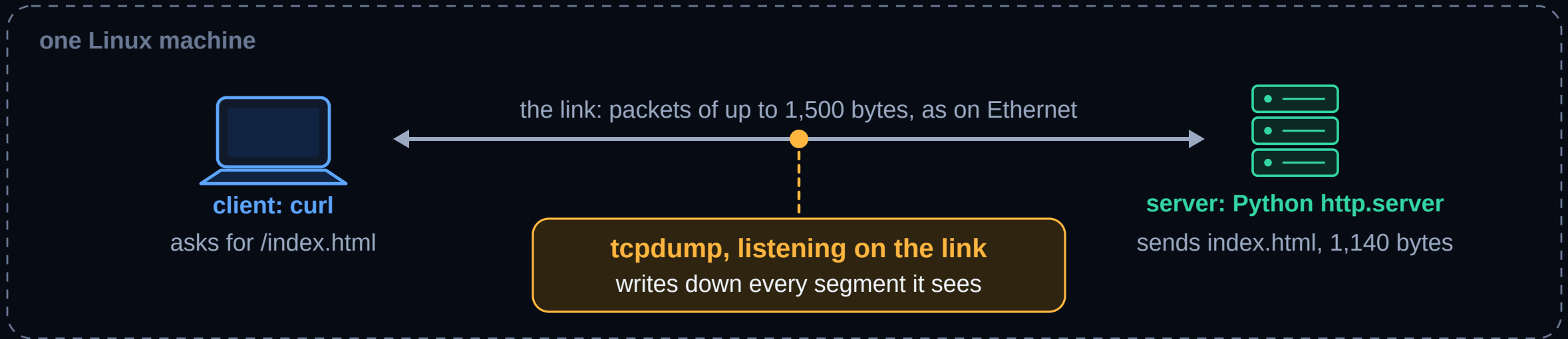
Which real protocols carry these mechanisms?



This lecture: the real TCP and UDP, and QUIC, TCP rebuilt over UDP: 40.8% of websites already serve pages over it.

TCP carries every mechanism of this chapter; UDP carries almost none.

How do we watch a real TCP connection?



server	<code>python3 -m http.server 8080</code>	serves this folder
client	<code>curl --http1.1 http://127.0.0.1:8080/index.html</code>	fetches the page once
recorder	<code>tcpdump -i lo 'tcp port 8080'</code>	every TCP segment

Fetch one small page, and record every segment that crosses the link.

What does a real TCP connection look like?

☒ drawn in this chapter ☐ never drawn

what tcpdump recorded: 12 segments, one per row

#	from → to	flags	seq	ack	len	win	options, payload
1	client → server	SYN	0		0	64240	mss 1460 · sackOK · wscale 10
2	server → client	SYN, ACK	0	1	0	65160	mss 1460 · sackOK · wscale 10
3	client → server	ACK	1	1	0	63	
4	client → server	PSH, ACK	1	1	87	63	GET /index.html
5	server → client	ACK	1	88	0	64	
6	server → client	PSH, ACK	1	88	188	64	HTTP/1.0 200 OK
7	client → server	ACK	88	189	0	63	
8	server → client	PSH, ACK	189	88	1140	64	(the page)
9	client → server	ACK	88	1329	0	66	
10	server → client	FIN, ACK	1329	88	0	64	
11	client → server	FIN, ACK	88	1330	0	66	
12	server → client	ACK	1330	89	0	64	

Every line carries fields this chapter has never drawn.

How many of these segments carry the page?

#	from → to	flags	len	payload	
1	client → server	SYN	0		opening the connection 3 segments, no data
2	server → client	SYN , ACK	0		
3	client → server	ACK	0		
4	client → server	PSH, ACK	87	GET /index.html	the request and the page 6 segments
5	server → client	ACK	0		
6	server → client	PSH, ACK	188	HTTP/1.0 200 OK	
7	client → server	ACK	0		
8	server → client	PSH, ACK	1140	(the page)	
9	client → server	ACK	0		
10	server → client	FIN , ACK	0		closing the connection 3 segments, no data
11	client → server	FIN , ACK	0		
12	server → client	ACK	0		

Half only open or close: a TCP connection has a beginning and an end.

What does a real UDP exchange look like?

Same machine, same recorder, another application: the client asks a DNS server for `www.example.com`.

#	from → to	ports	len	what it carries
1	client → server	51000 → 53	33	query: <code>www.example.com?</code>
2	server → client	53 → 51000	49	answer: <code>192.0.2.10</code>

✗ no opening

✗ no ACK

✗ no closing

If the answer is lost, the client simply asks again.

The DNS server is a few lines of Python (dnslib) on port 53; 192.0.2.10 is an address reserved for examples.

UDP sends a datagram and is done: no connection to open or close.

Five questions

☐ the simplification it removes

01 How does a TCP connection begin?

already connected

from 0

Opening

02 How does a TCP connection end?

never closed

Closing

03 What if the receiver cannot keep up?

always had room

Flow control

04 What if an application does not want all of this?

UDP

05 What if transport is rebuilt above UDP?

QUIC

01 to 03 are the real TCP, 04 is UDP, and 05 is QUIC: TCP's job rebuilt on UDP.

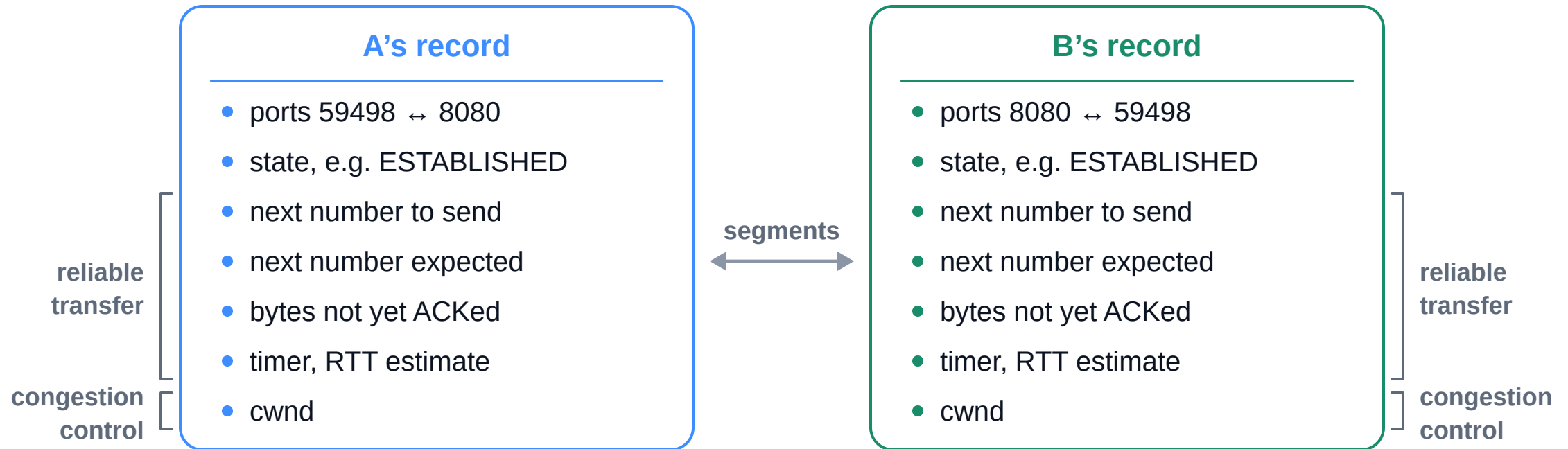
How does a TCP connection begin?

01

Opening a connection

In the trace, three segments went by before any data. What must two hosts set up before the first byte, and why does it take three segments?

What is a TCP connection?



Routers keep no record of it; NATs and firewalls on the path do. RFC 9293 calls the record the TCB.

A connection is a record kept at both ends: opening creates it, closing frees it.

1 Opening

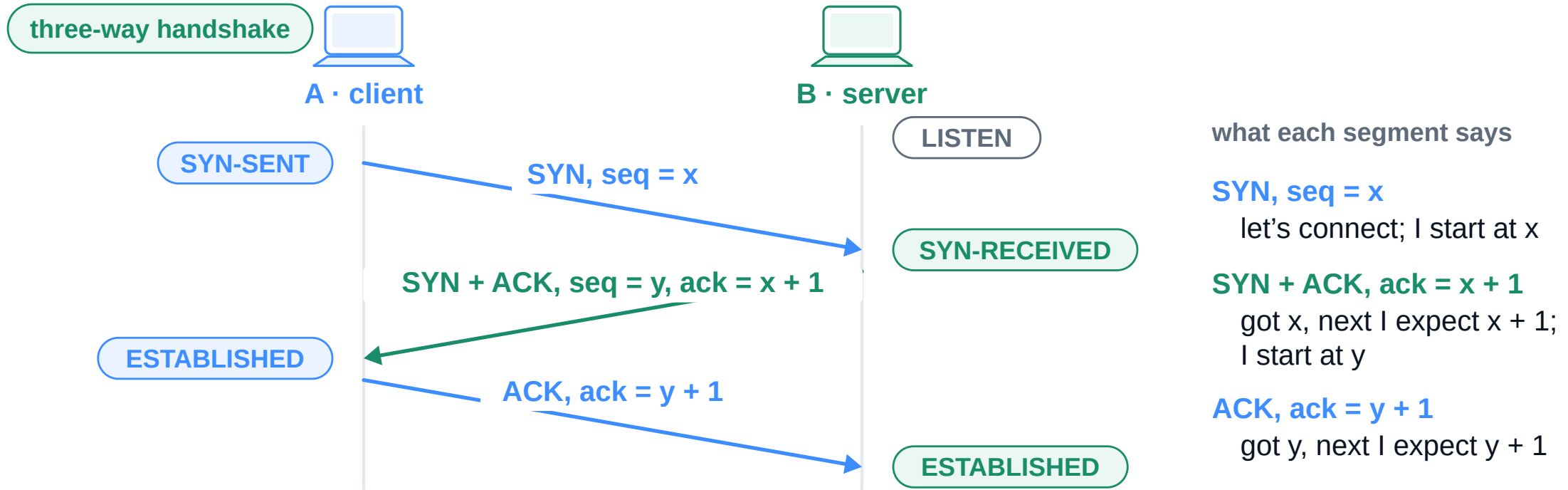
2 Closing

3 Flow control

4 UDP

5 QUIC

How does TCP open a connection?



Each ACK answers the other side's number plus one: seq = x is answered by ack = x + 1.

TCP opens with a three-way handshake: each side's number is sent and acknowledged.

1 Opening

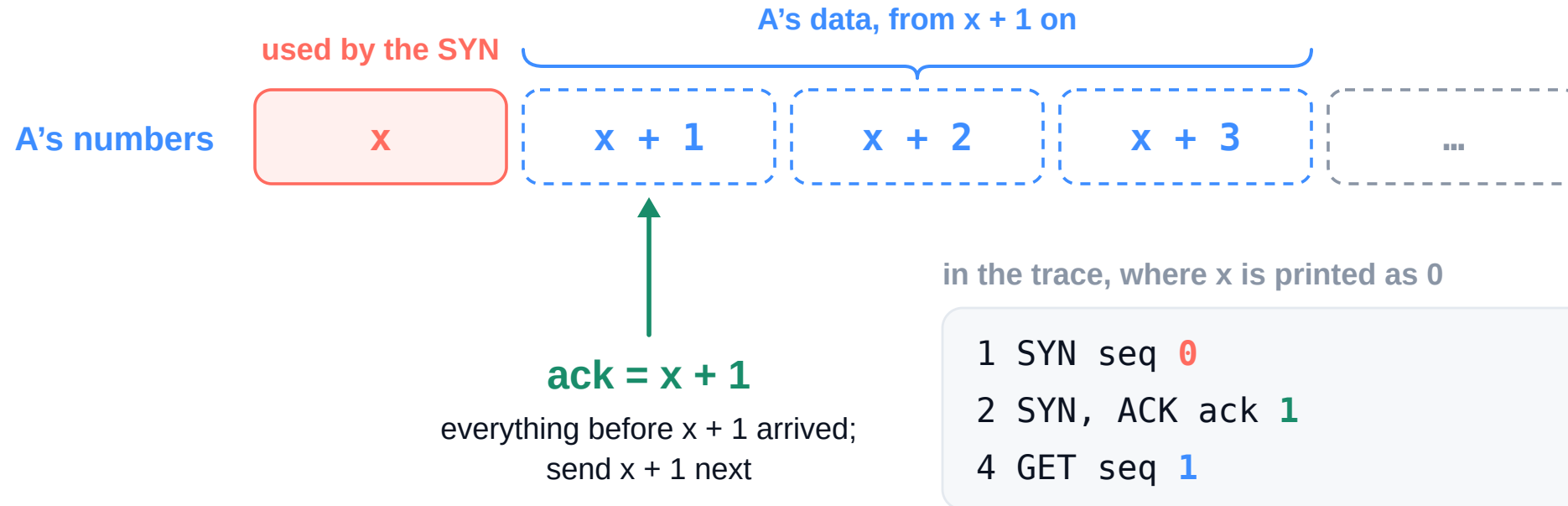
2 Closing

3 Flow control

4 UDP

5 QUIC

What does $\text{ack} = x + 1$ mean?



An ACK names the next number expected; the SYN used up x , so B asks for $x + 1$.

1 Opening

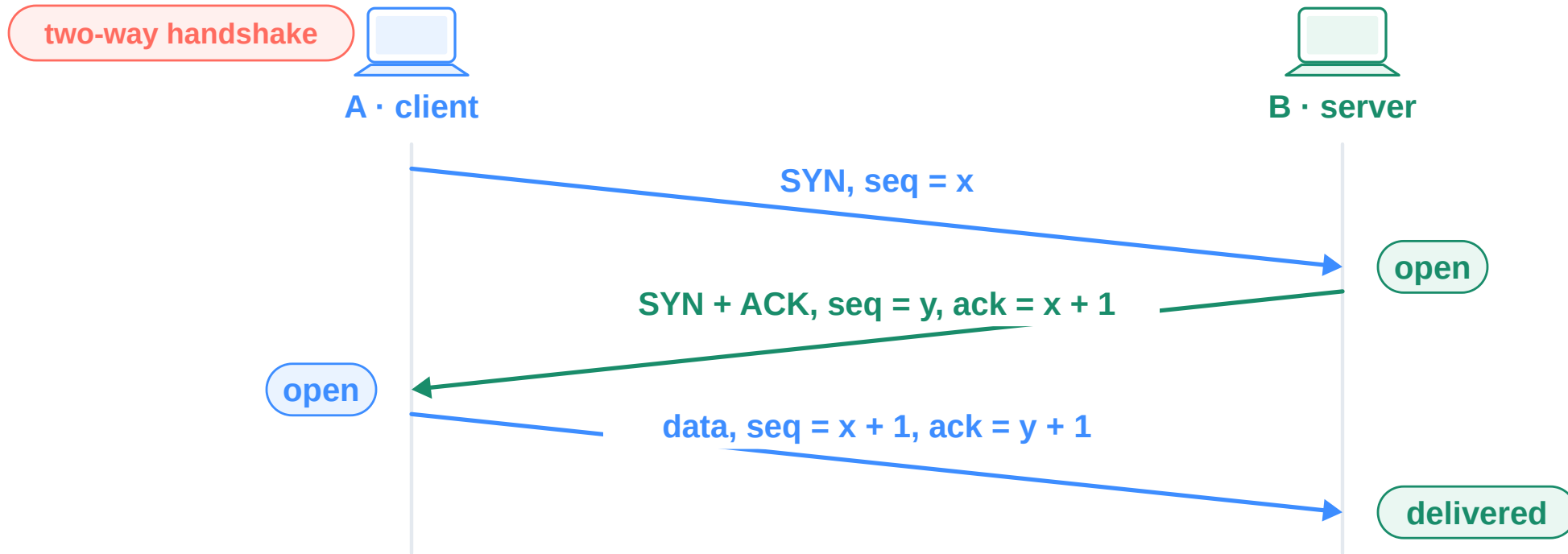
2 Closing

3 Flow control

4 UDP

5 QUIC

Would a two-way handshake be enough?



Drop the third segment: B opens as soon as the SYN arrives, and A's first data still acknowledges y.

When nothing goes wrong, a two-way handshake looks like enough.

1 Opening

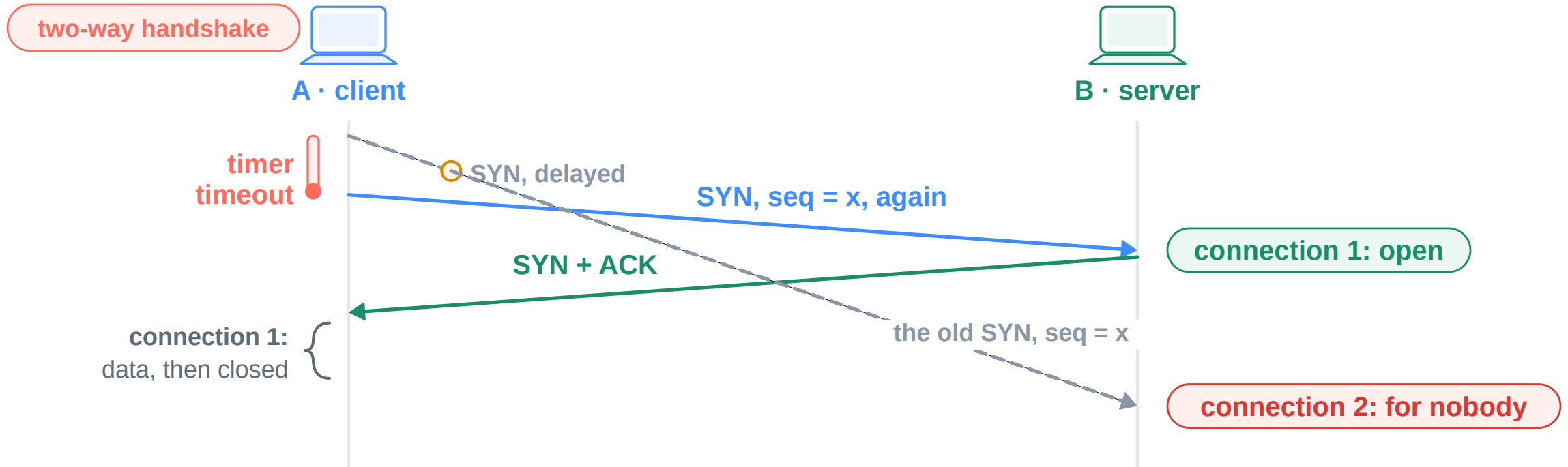
2 Closing

3 Flow control

4 UDP

5 QUIC

What if an old SYN arrives?



A asked once. Its first SYN was only delayed, so the timeout sent a second copy, and each copy opens a connection at B.

With a two-way handshake, one request from A opens two connections at B.

1 Opening

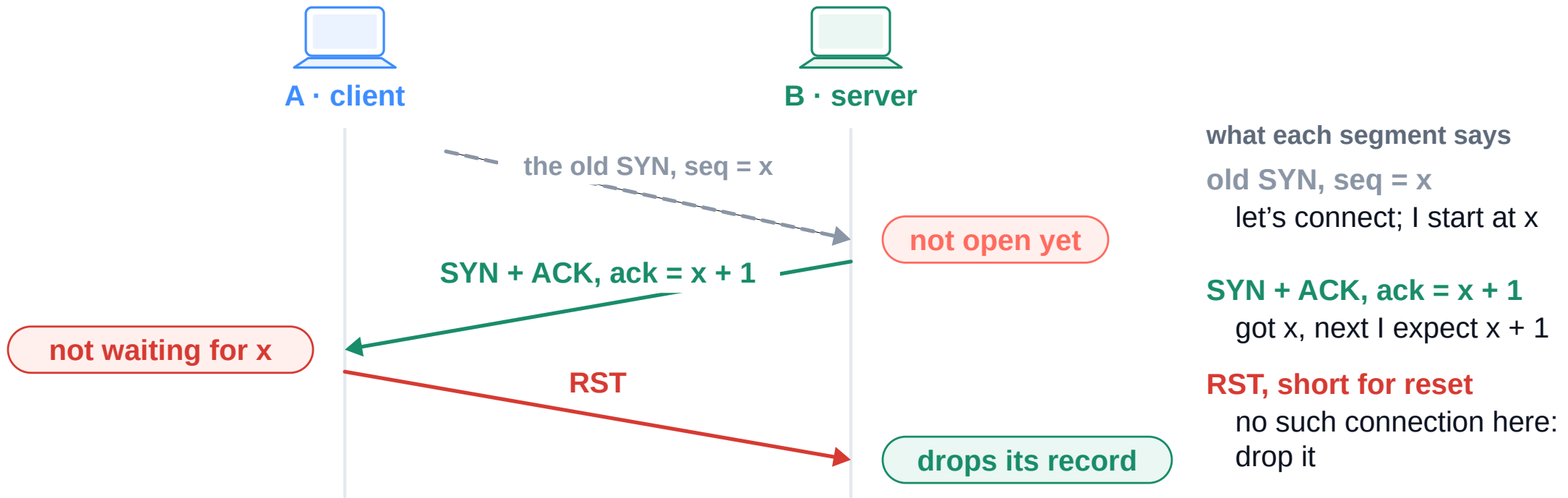
2 Closing

3 Flow control

4 UDP

5 QUIC

How does the third segment fix it?



RFC 9293: "The principal reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion."

B opens only on A's third segment, so A, the side that knows, can refuse with an RST.

1 Opening

2 Closing

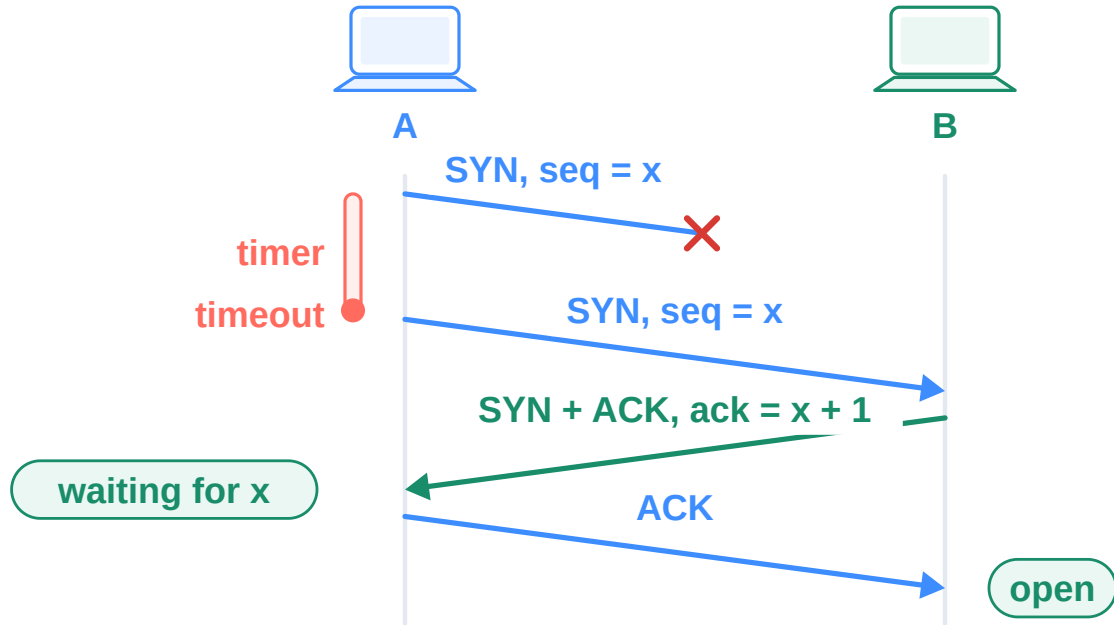
3 Flow control

4 UDP

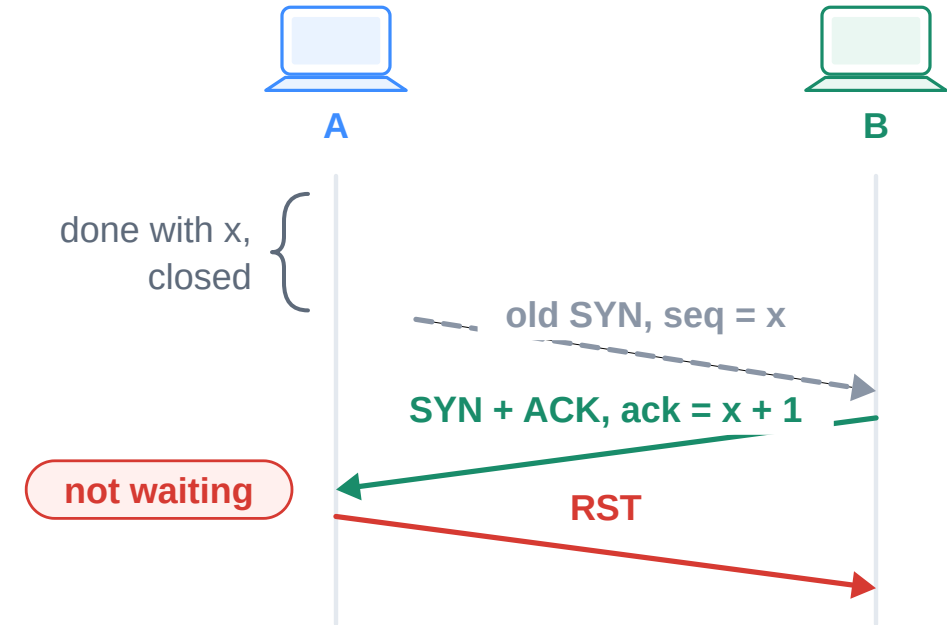
5 QUIC

Is a retransmitted SYN an old one?

a retransmission: A still waits



an old copy: A has moved on



Both copies carry the same x , and B cannot tell them apart. A can: it knows which SYN it is still waiting for.

A completes the SYN it is still waiting for, and refuses a copy it has moved past.

1 Opening

2 Closing

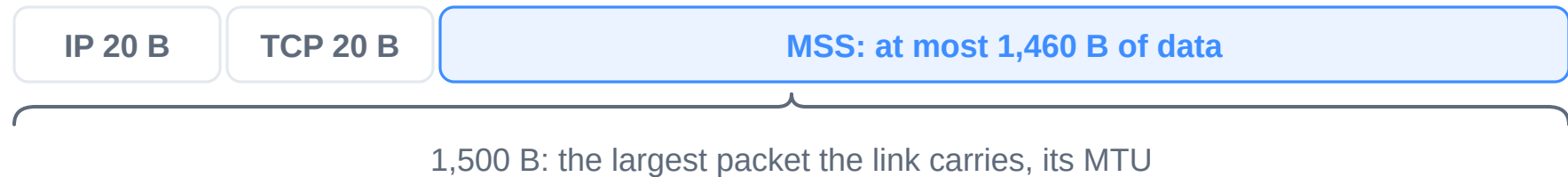
3 Flow control

4 UDP

5 QUIC

What else does the handshake agree on?

	A's SYN	B's SYN + ACK	what it sets
starting number	$x = 2,153,146,557$	$y = 1,006,549,976$	where each side's numbers begin
MSS	1,460	1,460	the most data per segment each side takes
SACK	permitted	permitted	selective ACKs, used if both SYNs offer it



The trace printed x and y as 0: tools subtract them to keep the numbers short.

Each SYN carries its sender's values; an option is used only if both SYNs carry it.

1 Opening

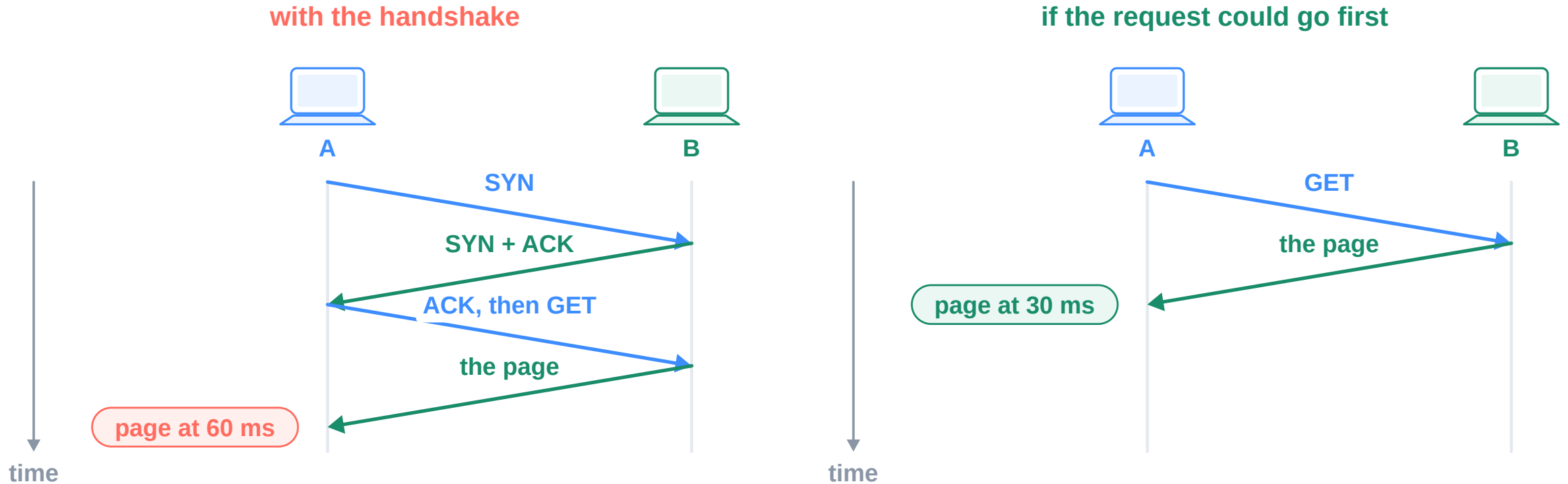
2 Closing

3 Flow control

4 UDP

5 QUIC

What does the handshake cost?



Every new connection waits one round trip before its first request.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

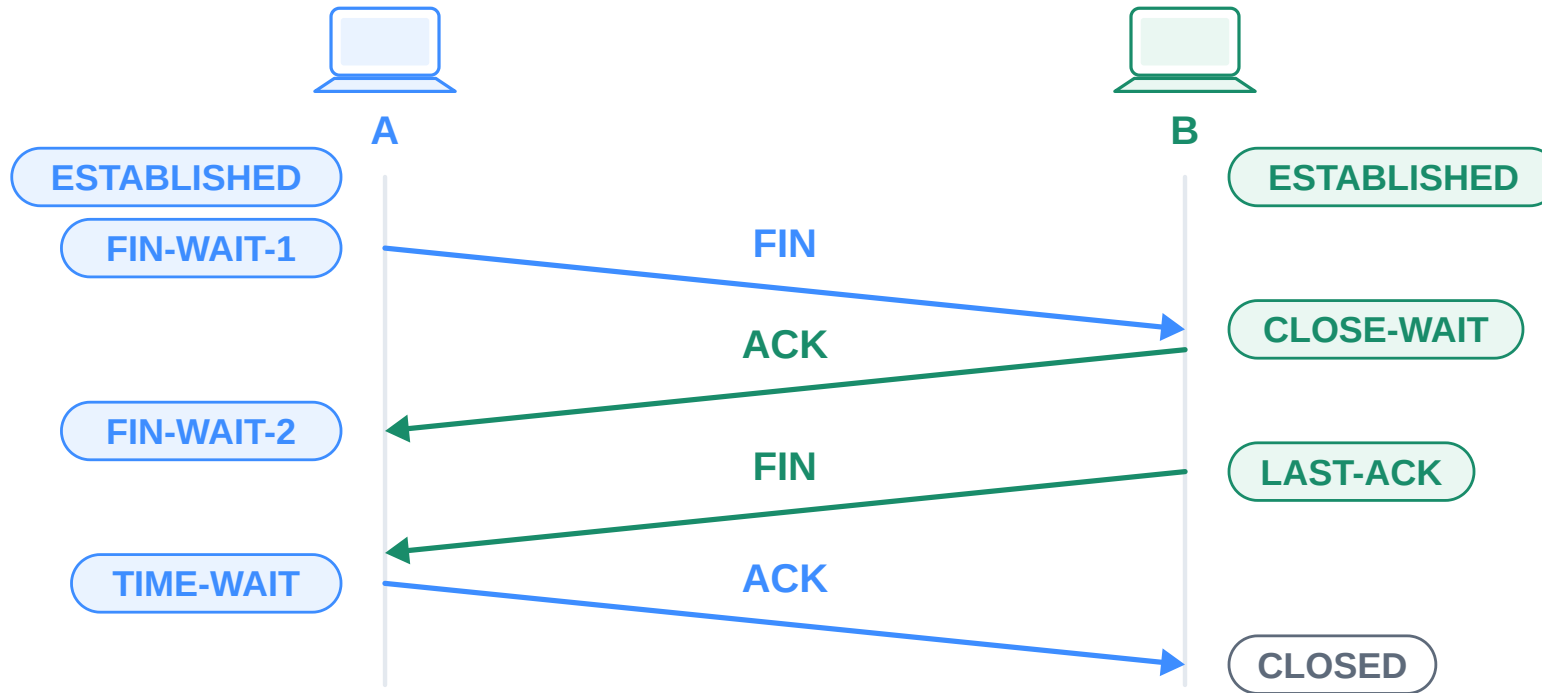
How does a TCP connection end?

02

Closing a connection

Closing frees the record at each end. Before it does, each end must be sure the other has sent its last byte.

How does TCP close a connection?



what each segment says

FIN: I have no more to send.

ACK: I have your FIN.

FIN: I have no more either.

ACK: I have your FIN.

The chips are RFC 9293's states. In TIME-WAIT, A waits in case its last ACK is lost and B sends its FIN again.

Each side sends a FIN when it is done sending, and the other acknowledges it.

1 Opening

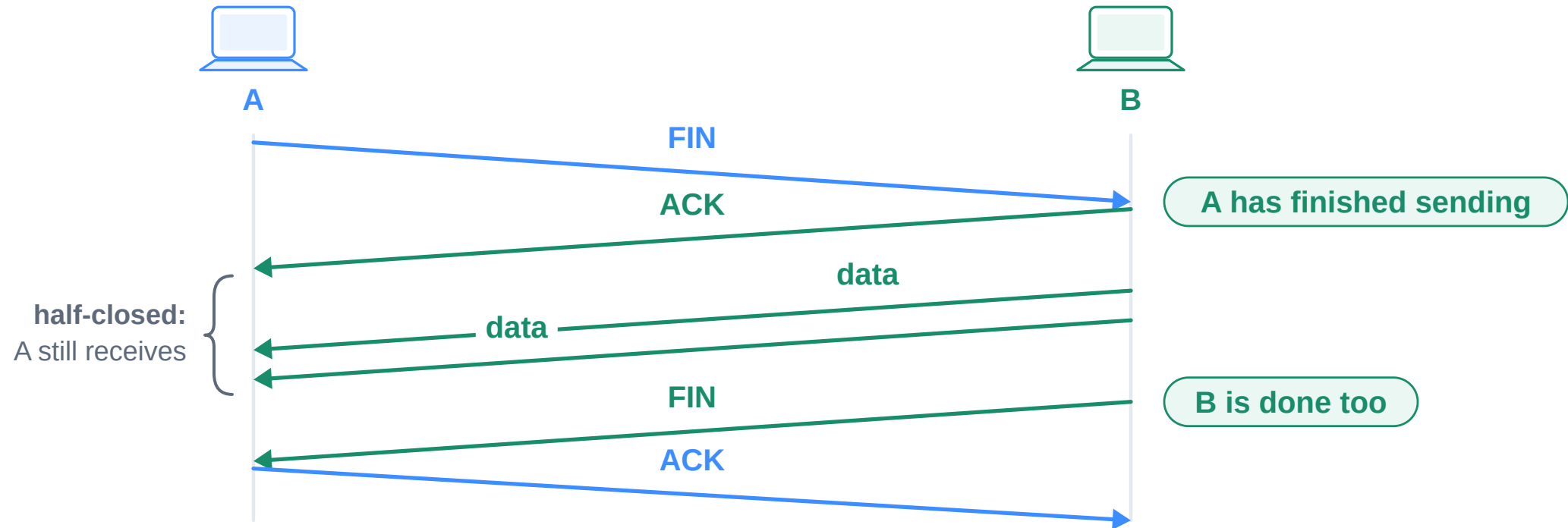
2 Closing

3 Flow control

4 UDP

5 QUIC

Why four segments, when opening took three?



If B is done too when A's FIN arrives, its FIN rides with its ACK, as its SYN did: the trace closed in three segments.

B acknowledges A's FIN at once, but sends its own FIN only when it is done.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

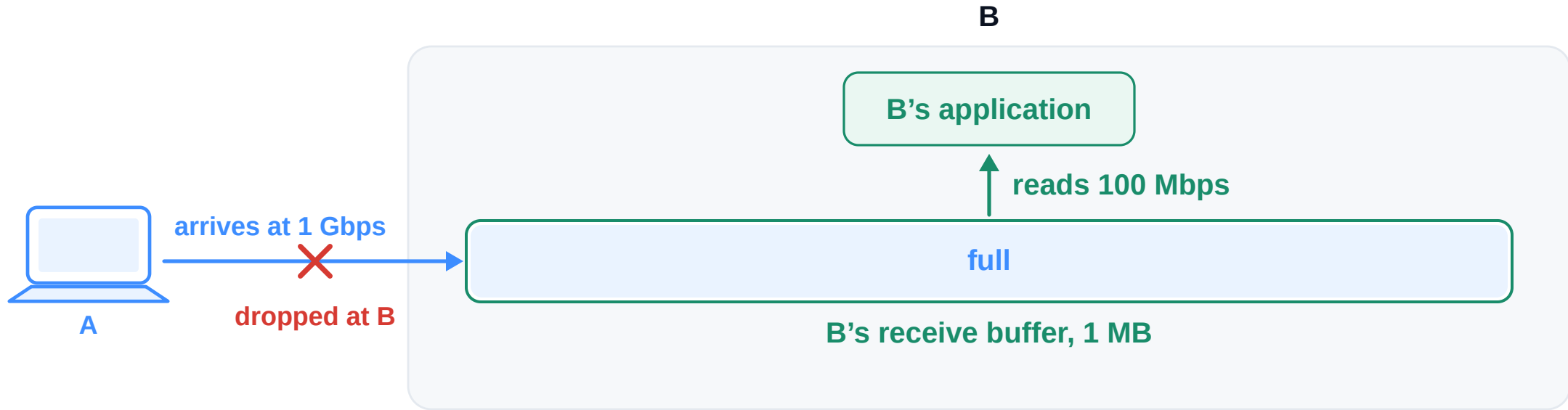
What if the receiver cannot keep up?

03

Flow control

The congestion-control lectures assumed that B always had room. Here B's application reads slowly, its buffer fills, and A needs a second limit.

What if B's application reads slowly?



$$8 \text{ Mb} \div (1 \text{ Gbps} - 100 \text{ Mbps}) = 8.9 \text{ ms}$$

One key press plays the fill in slow motion: 8.9 ms shown as 4 seconds. 1 MB is 8 Mb; the buffer gains 900 Mbps.

B's buffer fills in 8.9 ms; every packet after that is dropped at B.

1 Opening

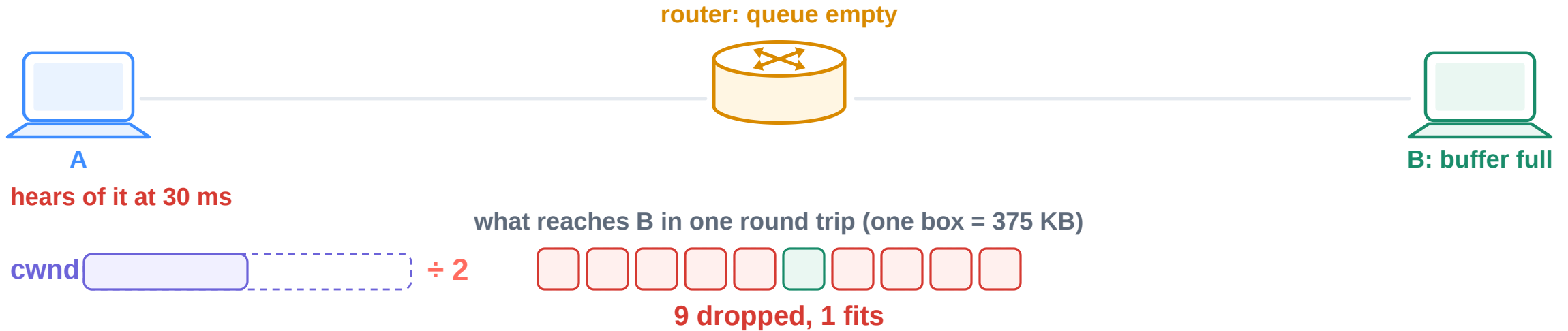
2 Closing

3 Flow control

4 UDP

5 QUIC

Why not let the loss tell A?



1 3.4 MB dropped at B

a round trip of data arrives before A hears

2 cwnd halved for nothing

A blames the network, whose queue is empty

One round trip (30 ms) shown as 6 seconds. $1 \text{ Gbps} \times 30 \text{ ms} = 3.75 \text{ MB}$ arrive; the application reads 0.375 MB: one box in ten.

Loss at the receiver wastes a round trip of data and is read as congestion.

What if every ACK says how much room is left?



3 The application reads 300 KB: win 300 KB, and A may send again

$$\text{rwnd} = \text{buffer} - (\text{received} - \text{read})$$

Every ACK carries rwnd, B's free space, and A never sends more than that.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

How does A combine this with cwnd?

what the network allows

cwnd = 2,500 packets = 3.75 MB

what B can hold



rwnd = 200 KB



window = min(cwnd, rwnd) = 200 KB

rate $\leq$ window $\div$ RTT = 200 KB $\div$ 30 ms = **53 Mbps**

The congestion-control lectures assumed B always had room, so the window there was cwnd alone.

A's window is the smaller of what the network allows and what B can hold.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

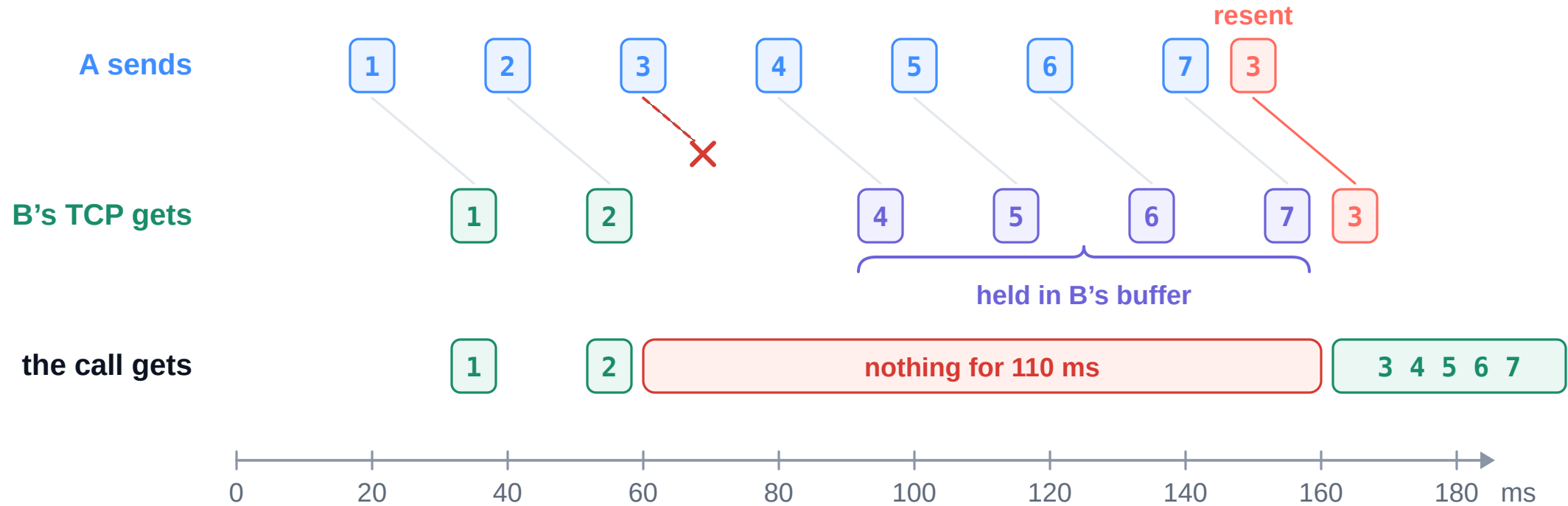
What if an application does not want all of this?

04

UDP

Every guarantee TCP gives is paid for in waiting. Here is an application that cannot wait, and what is left when the guarantees go.

What does one lost packet do to a live call?



A frame every 20 ms, RTT 30 ms; three duplicate ACKs make A resend 3. One key press plays it 50 times slower.

Head-of-line blocking: frames that arrived in time wait for the one that was lost.



What does each TCP guarantee cost?

Opening

One round trip before the first byte
paid by every new connection, before any data

In order

A loss holds back everything behind it
head-of-line blocking, as in the call

Reliable

Data is resent even when it is too late to use
a frame that arrives 90 ms late is not worth playing

Rate

TCP's congestion control sets the sending rate
not the application, which may know its media better

Every guarantee is paid for in waiting, and some applications cannot wait.

1 Opening

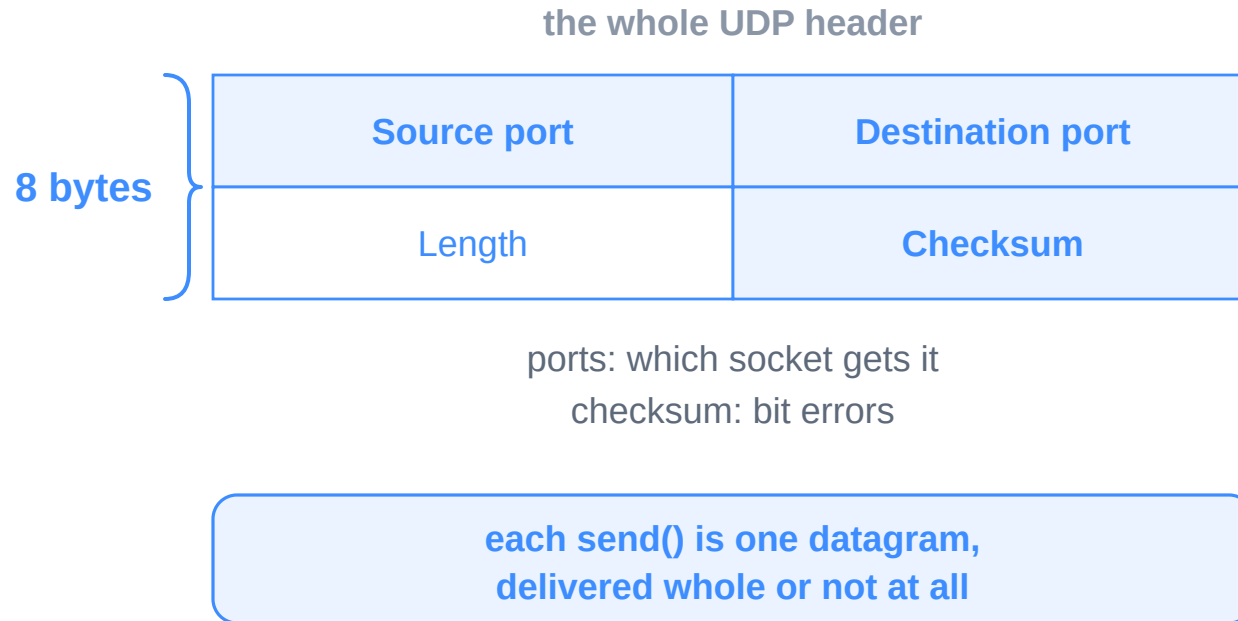
2 Closing

3 Flow control

4 UDP

5 QUIC

What is left without them?



what UDP leaves out

- ✗ no handshake
- ✗ no sequence numbers or ACKs
- ✗ no retransmission
- ✗ no rwnd, no cwnd

UDP adds ports and a checksum to IP; everything else is the application's.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

What do applications on UDP add back?

application	what it adds back	what it leaves out
DNS lookup	a timeout and a retry	a connection, order
live call (RTP)	numbers, timestamps, its own rate	resending late frames
online game	the latest state in every packet	resending old state
clock sync (NTP)	timestamps in each reply	everything else

RTP (RFC 3550) numbers and timestamps every packet, so a receiver can reorder and time them without TCP.

An application on UDP rebuilds only the mechanisms it needs.

1 Opening

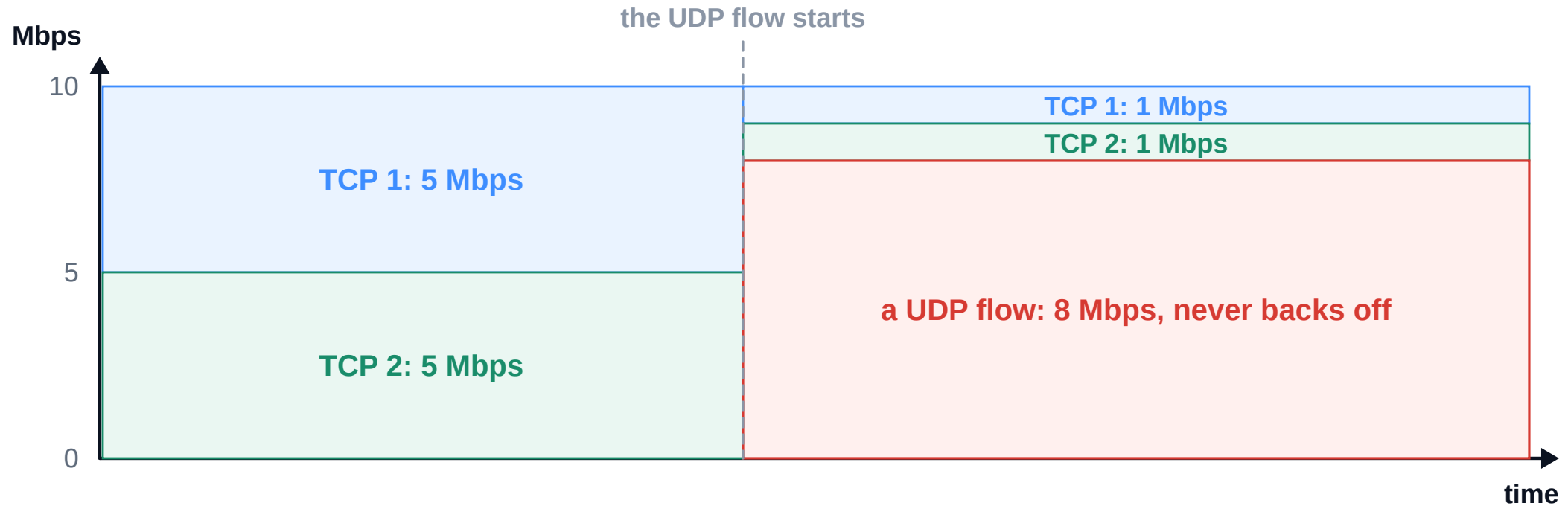
2 Closing

3 Flow control

4 UDP

5 QUIC

Can a UDP sender send as fast as it likes?



A model of one 10 Mbps bottleneck. RFC 8085: an application that does not use congestion-controlled transport SHOULD control its rate.

UDP never slows a sender down, so the application must slow itself down.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

What if transport is rebuilt above UDP?

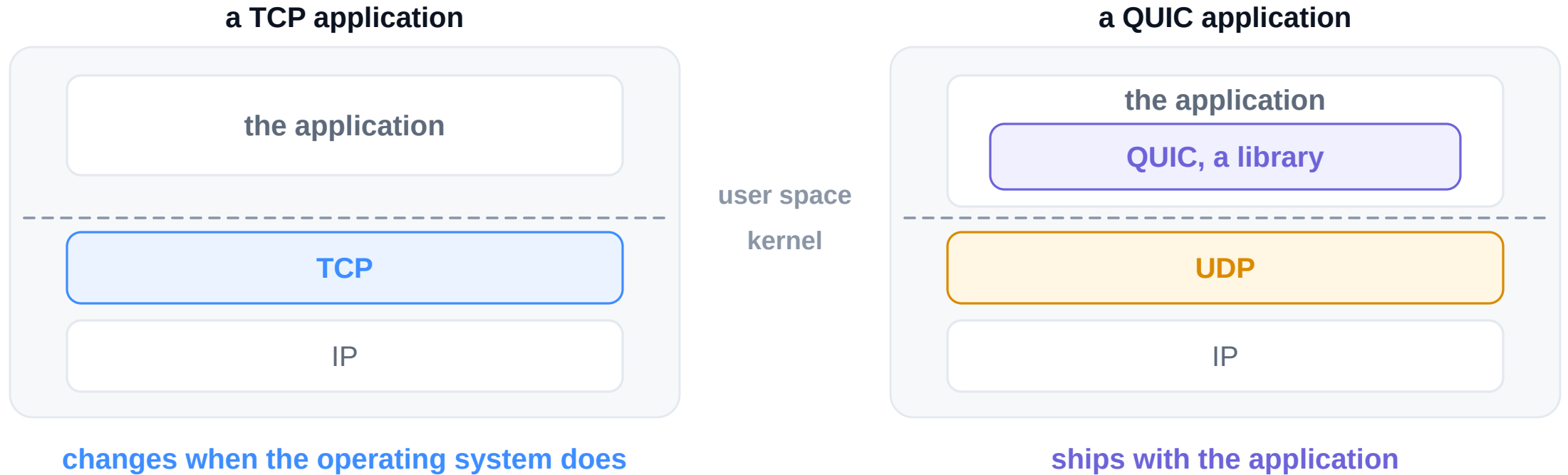
05

QUIC

TCP is hard to change: it lives in every operating system's kernel, and boxes on the path read its header and break what they do not know. QUIC does TCP's job inside the application, over UDP.

Where does QUIC run?

PROBLEM TCP lives in the kernel, so every change waits for every operating system.



QUIC is a library in the application; the kernel and the path see only UDP.

1 Opening

2 Closing

3 Flow control

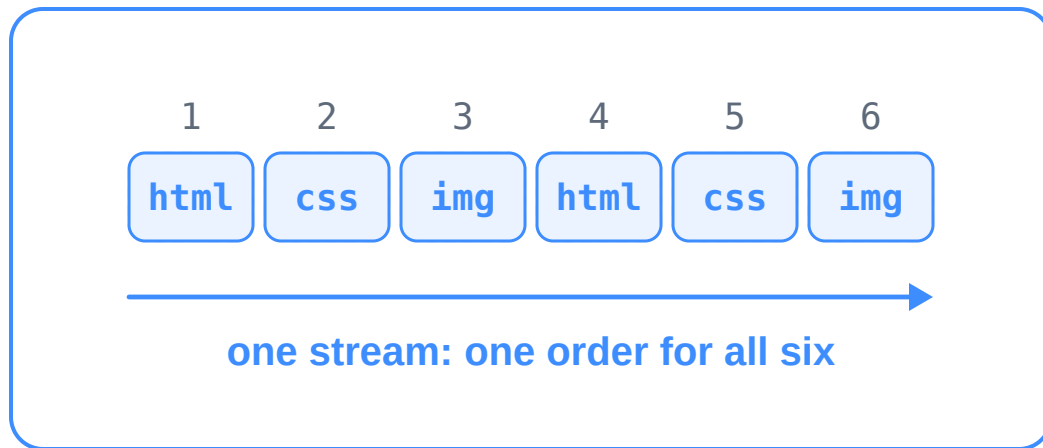
4 UDP

5 QUIC

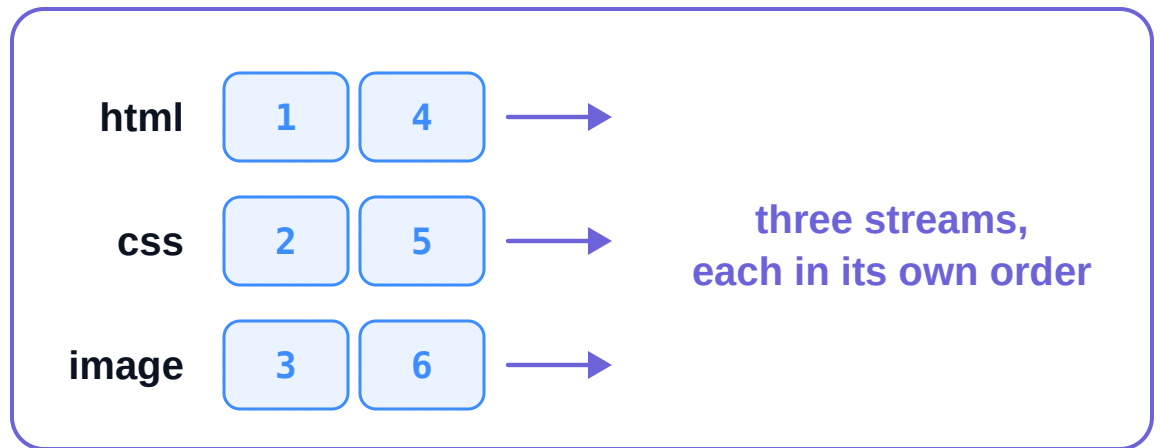
What is a stream, and what is a connection?

PROBLEM A page is many objects, but a TCP connection delivers all of them in one order.

a TCP connection



a QUIC connection



stream: bytes handed to the application in the order they were sent

connection: a record at each end, set up once by a handshake; its streams share it

A TCP connection is one stream; a QUIC connection carries many, each in its own order.

1 Opening

2 Closing

3 Flow control

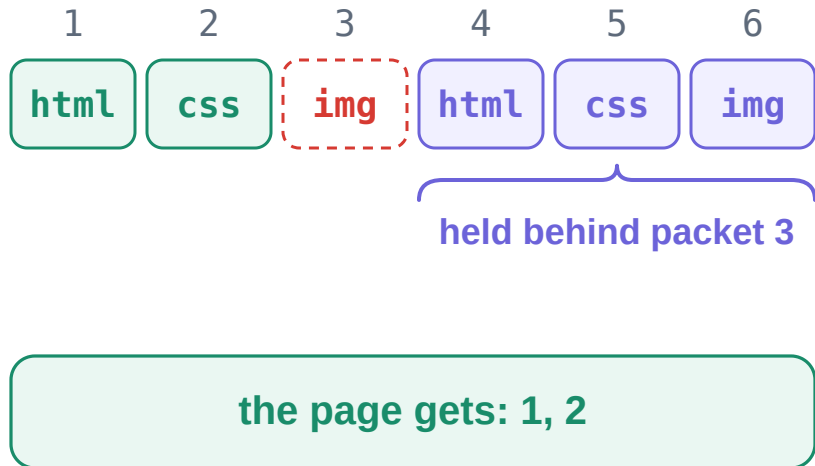
4 UDP

5 QUIC

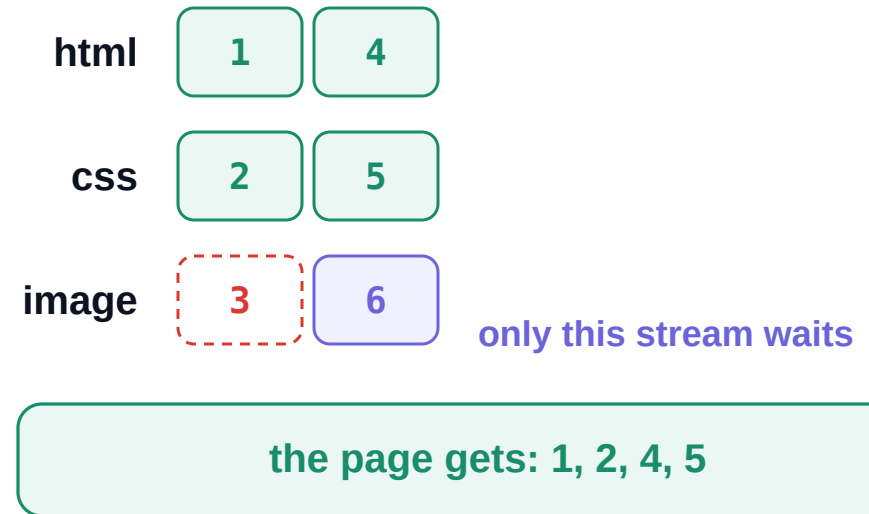
Does one lost packet still stall the page?

PROBLEM Over TCP, one loss held back every frame of the live call behind it.

TCP: one ordered byte stream



QUIC: three streams



Each QUIC stream is ordered on its own, so a loss delays only its stream.

1 Opening

2 Closing

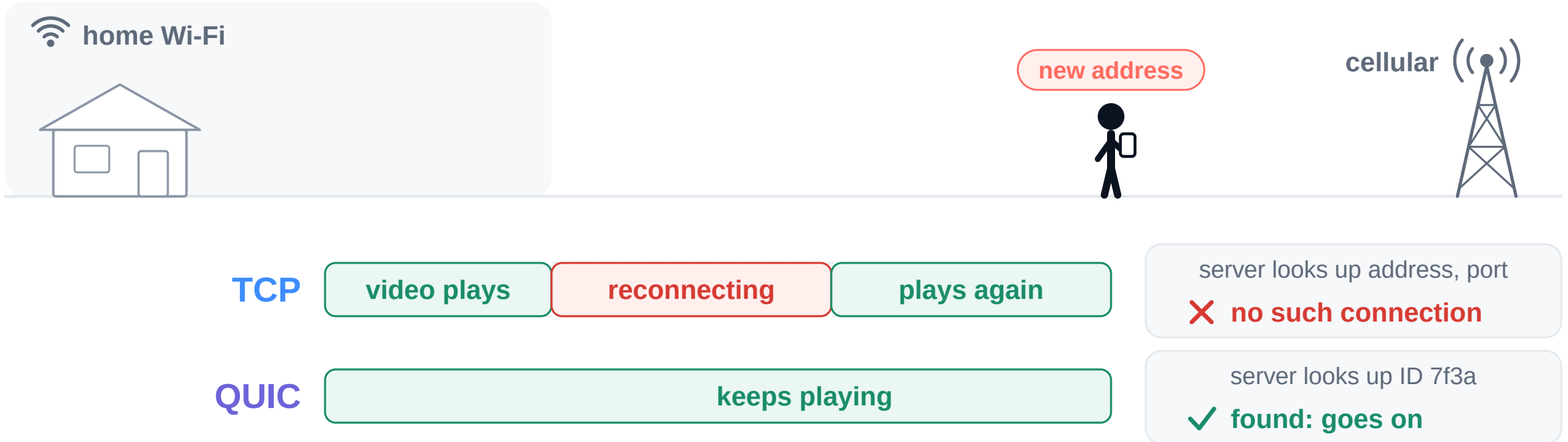
3 Flow control

4 UDP

5 QUIC

What if the phone changes networks?

PROBLEM TCP names a connection by addresses and ports, and a moving phone gets a new address.



A connection ID names a QUIC connection, so it survives a new address.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

What does user space cost?

CPU

More work per byte than TCP in the kernel

Google, 2017: about twice the CPU of HTTPS over TCP, after tuning

Trust

Each application ships its own congestion control

nothing in the network checks it

Reach

Some networks block UDP altogether

3 to 5% of networks: the client falls back to TCP

User space buys freedom to change, paid in CPU, trust and reach.

1 Opening

2 Closing

3 Flow control

4 UDP

5 QUIC

Five answers

01

a record at each end, opened by a three-way handshake; A refuses old SYNs

Opening

02

a FIN from each side, and each FIN acknowledged

Closing

03

every ACK carries rwnd; the window is $\min(\text{cwnd}, \text{rwnd})$

Flow control

04

ports and a checksum; the rest, rate included, is the application's

UDP

05

TCP's job done in the application over UDP, with many streams per connection

QUIC

Each answer is a mechanism of this chapter, packaged into a protocol.