



UNIVERSITY OF
WATERLOO

The transport layer: TCP and QUIC

CS 456/656, Fall 2026

Mina Tahmasbi Arashloo

The main decisions in a transport protocol

The main decisions in a transport protocol

1. Get the data to the right application

- App interface: one byte stream? Several streams? Independent messages?
- Channel ID in the metadata
- Establishing and closing channels

The main decisions in a transport protocol

1. Get the data to the right application

- App interface: one byte stream? Several streams? Independent messages?
- Channel ID in the metadata
- Establishing and closing channels

2. Segmentation and reassembly

- Segment ID in the metadata

The main decisions in a transport protocol

1. Get the data to the right application

- App interface: one byte stream? Several streams? Independent messages?
- Channel ID in the metadata
- Establishing and closing channels

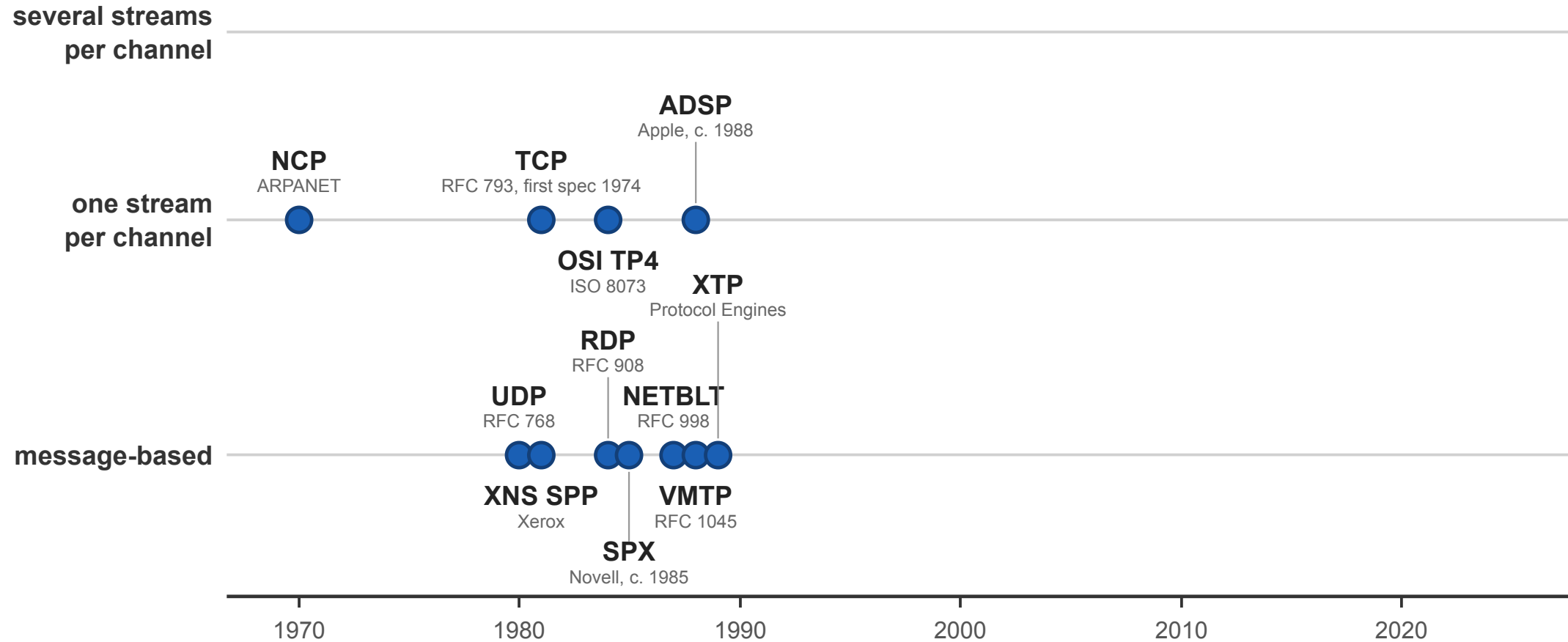
2. Segmentation and reassembly

- Segment ID in the metadata

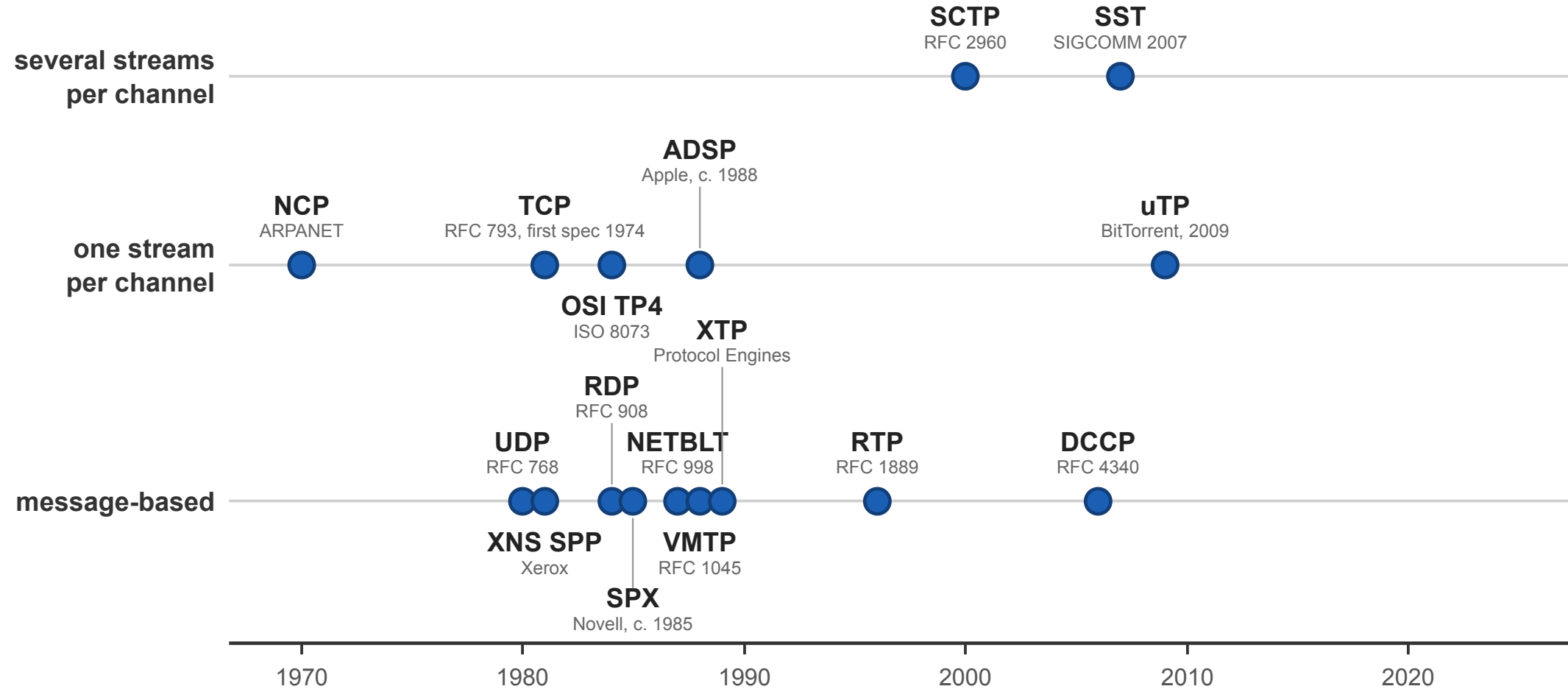
3. Deal with packet corruption, loss, and slowdown

- Mechanisms: checksum, feedback (positive, negative, selective, cumulative, ...), timers, sliding windows, flow control, congestion control, ...

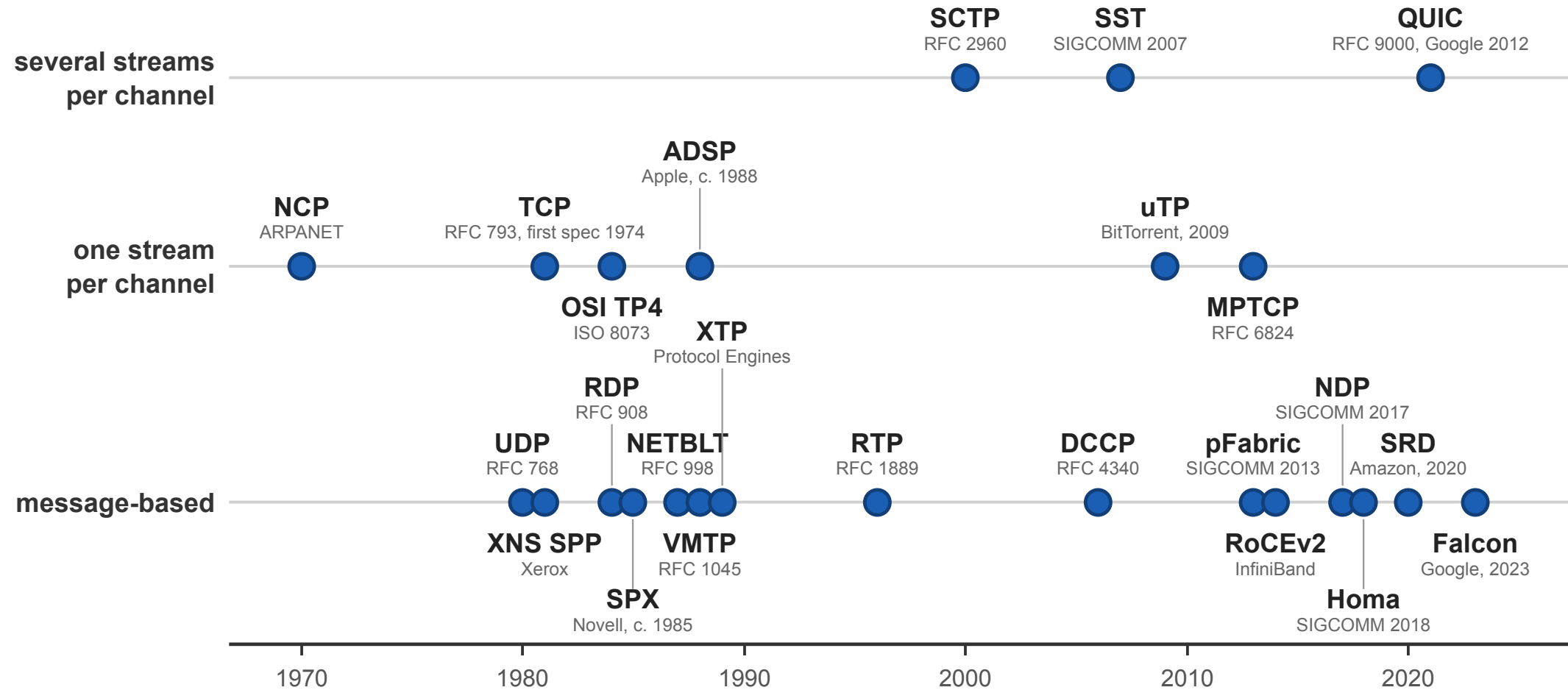
Transport protocols over the years



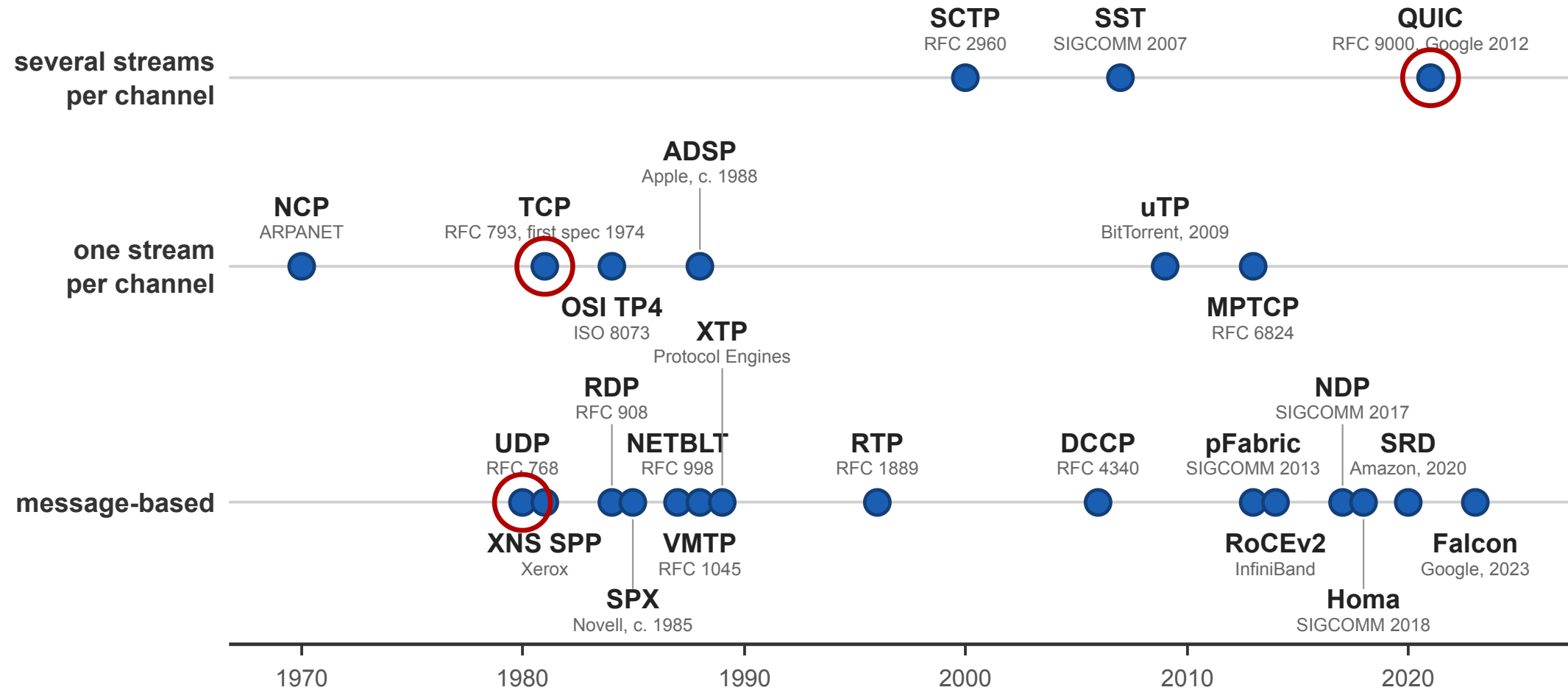
Transport protocols over the years



Transport protocols over the years



Transport protocols over the years



UDP: User Datagram Protocol

- 1. Get the data to the right application**
- 2. Segmentation and reassembly**
- 3. Deal with packet corruption, loss, and slowdown**

UDP: User Datagram Protocol

1. Get the data to the right application

~~2. Segmentation and reassembly~~ **nothing**

~~3. Deal with packet corruption, loss, and slowdown~~ **nothing**

UDP: User Datagram Protocol

1. Get the data to the right application

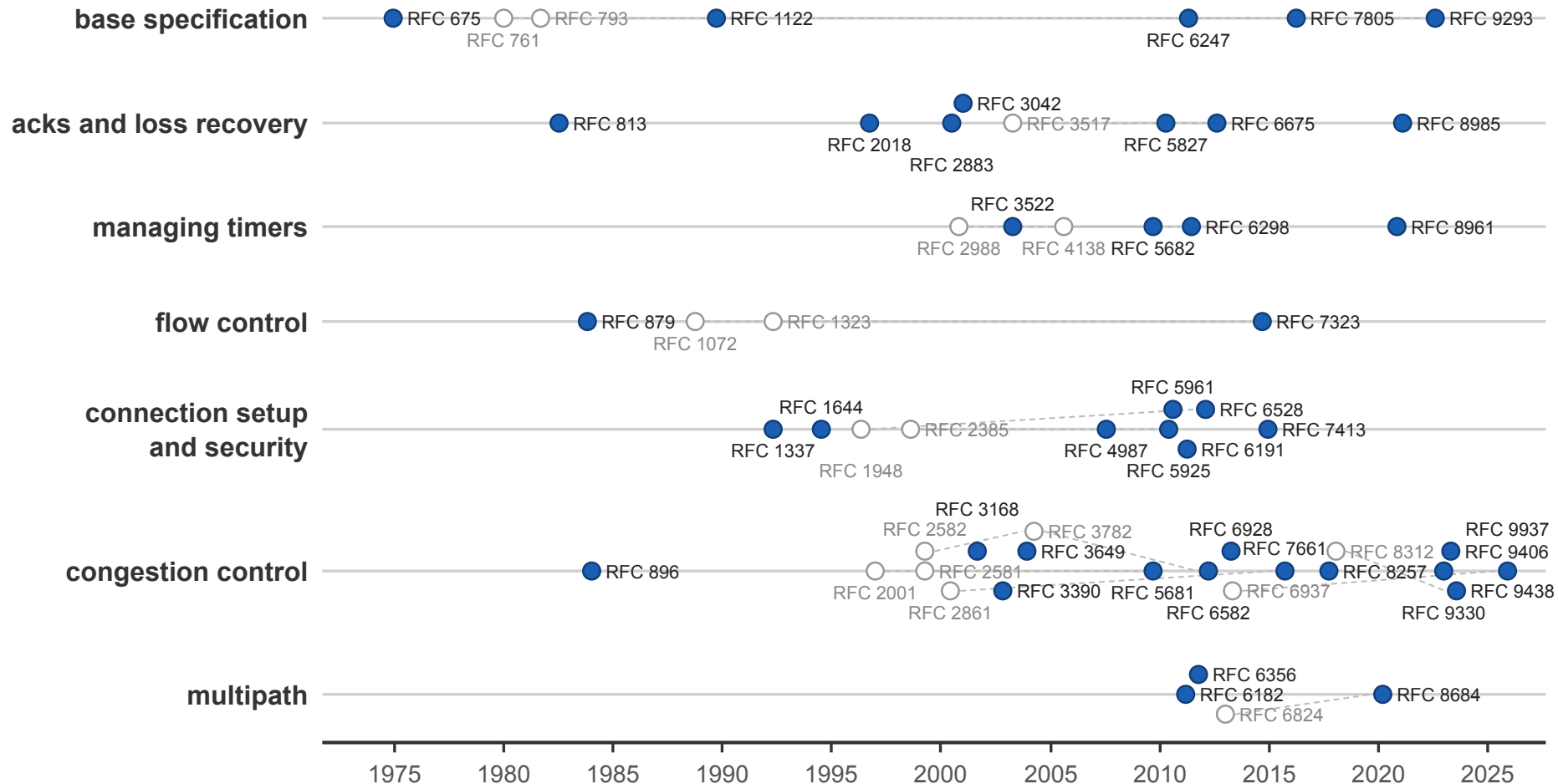
- App interface: independent messages, and each message should fit into a single packet
- Channel ID: uses port addresses; we'll discuss in TCP
- Establishing and closing channels: no guarantees, so no coordination needed

~~2. Segmentation and reassembly~~ **nothing**

~~3. Deal with packet corruption, loss, and slowdown~~ **nothing**

TCP: Transmission Control Protocol

One of the oldest and most widely deployed protocols in the Internet.



RFC: Request for Comments,
the Internet's standards documents

TCP: Transmission Control Protocol

1. Get the data to the right application

- App interface: one byte stream per channel, in each direction

2. Segmentation and reassembly

3. Deal with packet corruption, loss, and slowdown

TCP: Transmission Control Protocol

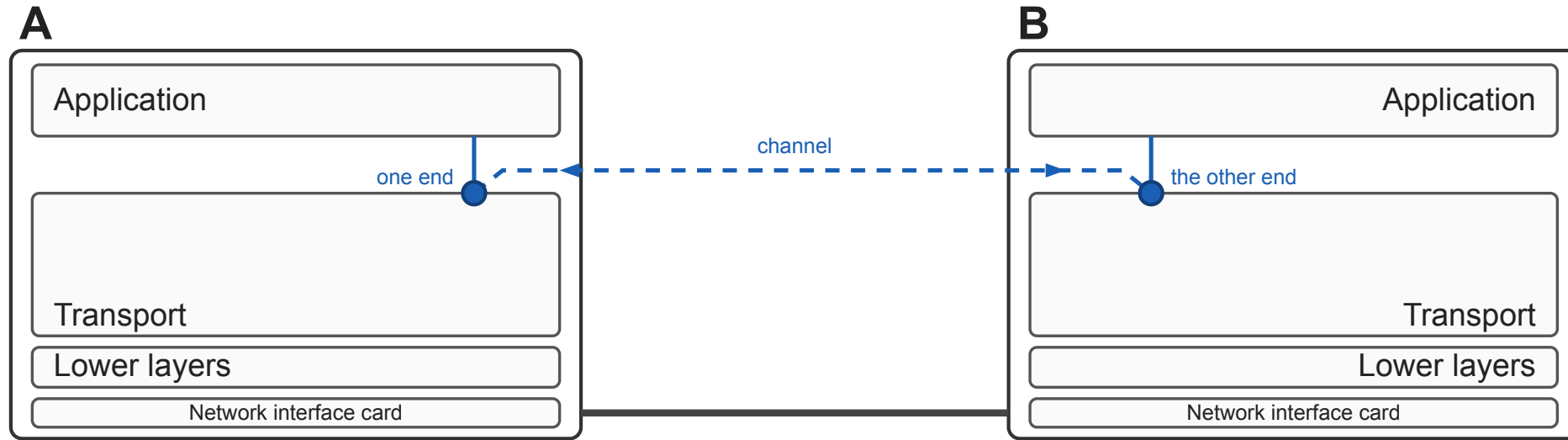
1. Get the data to the right application ✓

- App interface: one byte stream per channel, in each direction

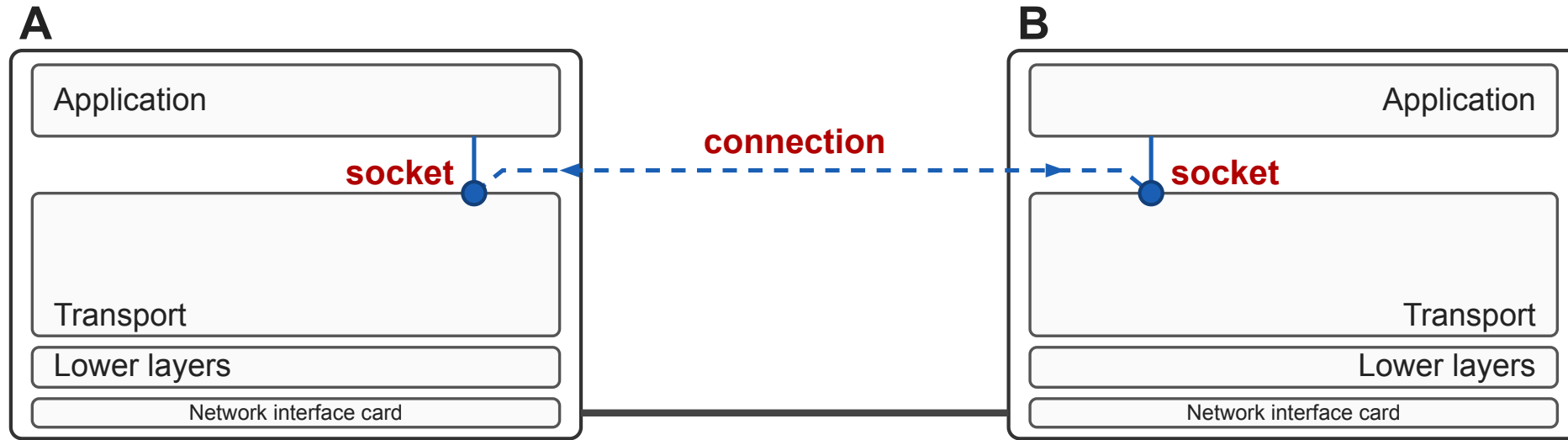
2. Segmentation and reassembly ✓

3. Deal with packet corruption, loss, and slowdown ✓

Channels, sockets and connections

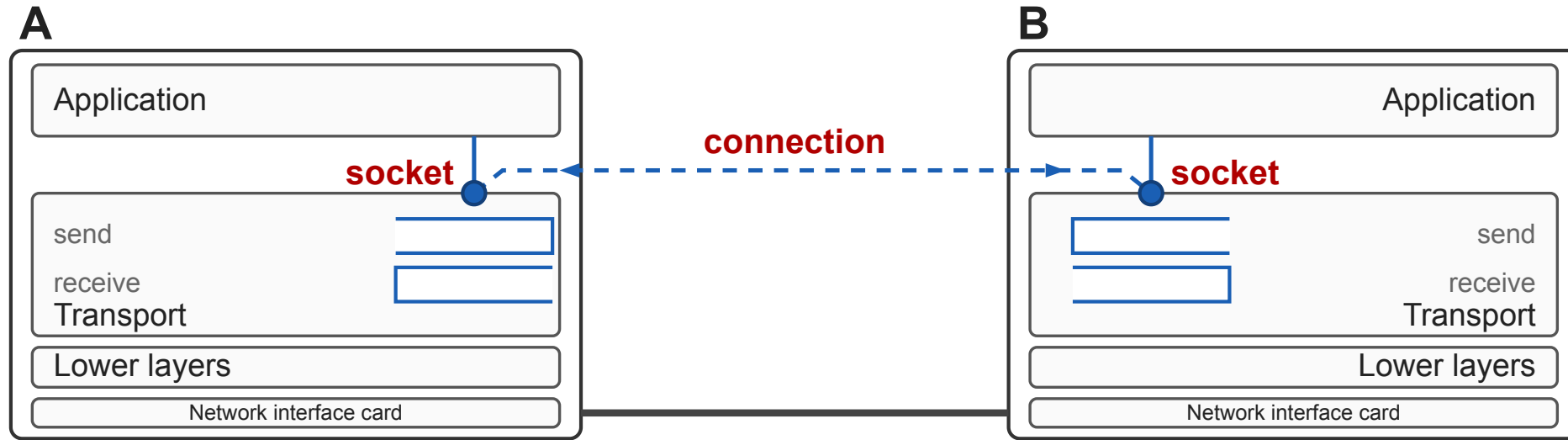


Channels, sockets and connections



Channel → Connection, Channel end → Socket

Channels, sockets and connections

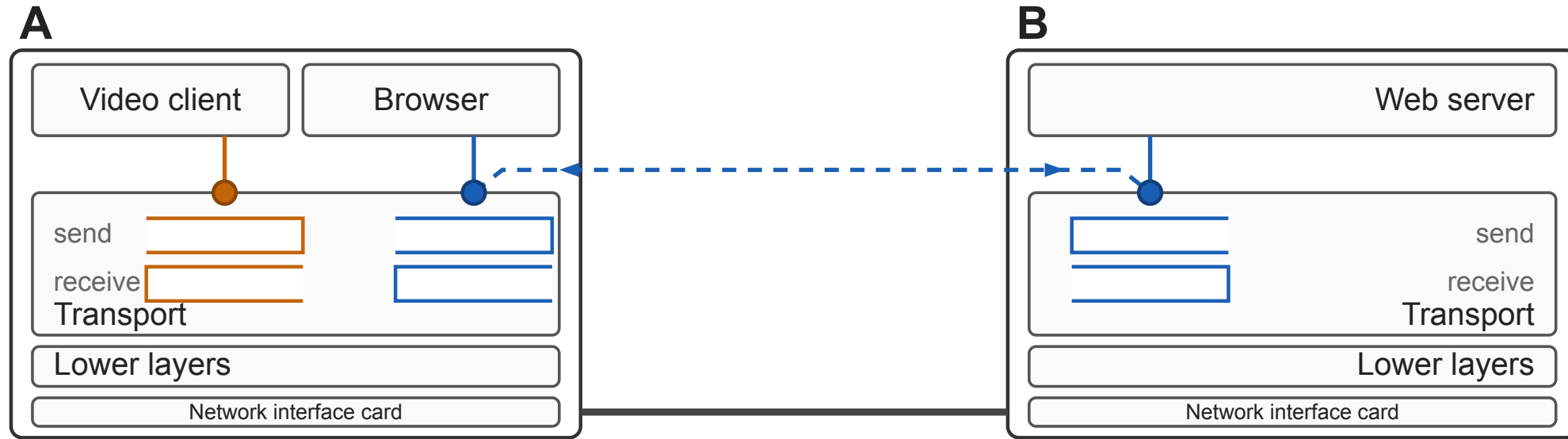


Channel → Connection, Channel end → Socket

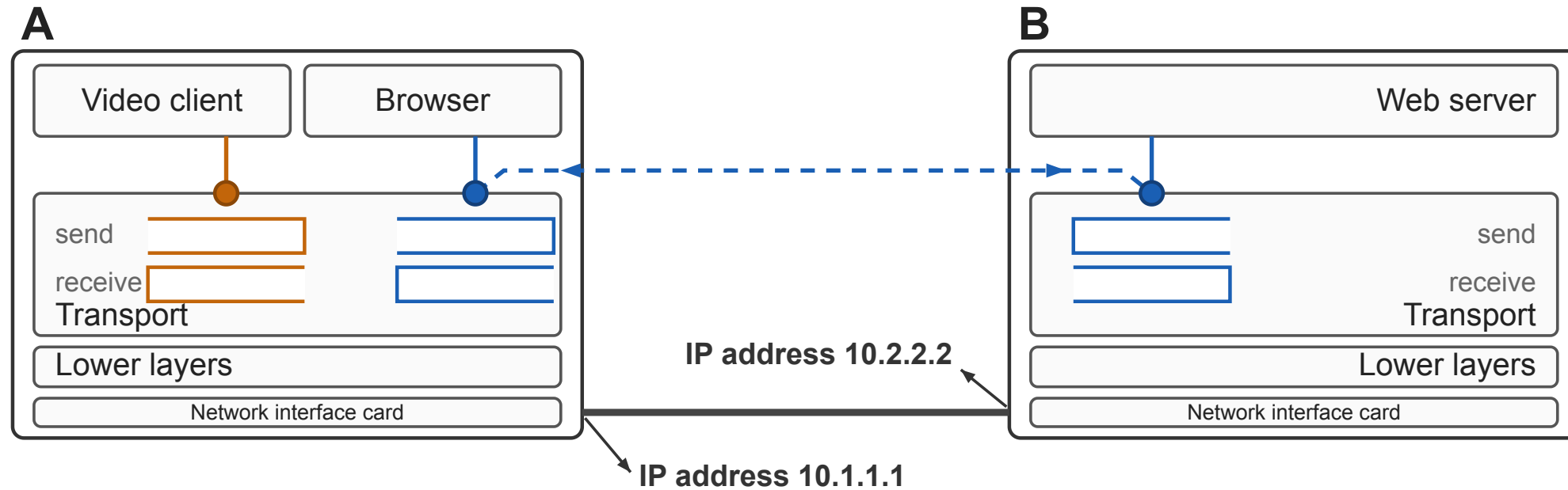
Ports and IP addresses

- **IP address:** identifies a computer on the network, or rather its network interface
 - not transport's business: it is a field in the header of the layer below (Part 2)
- **Port number:** a 16-bit number, local to one computer, that the transport layer uses to tell its connections apart
 - port 80 on B and port 80 on C have nothing to do with each other
- **TCP's “channel ID”:** the two port numbers in TCP's header, plus the two IP addresses in the header of the layer below, identify a connection

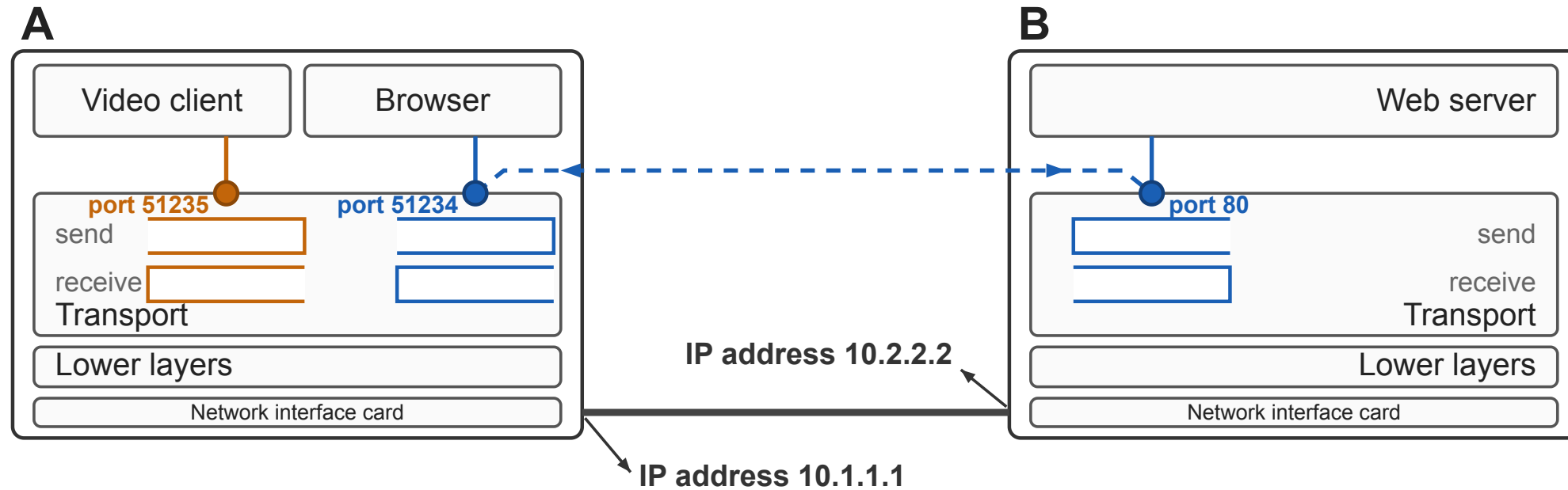
Ports and IP addresses: an example



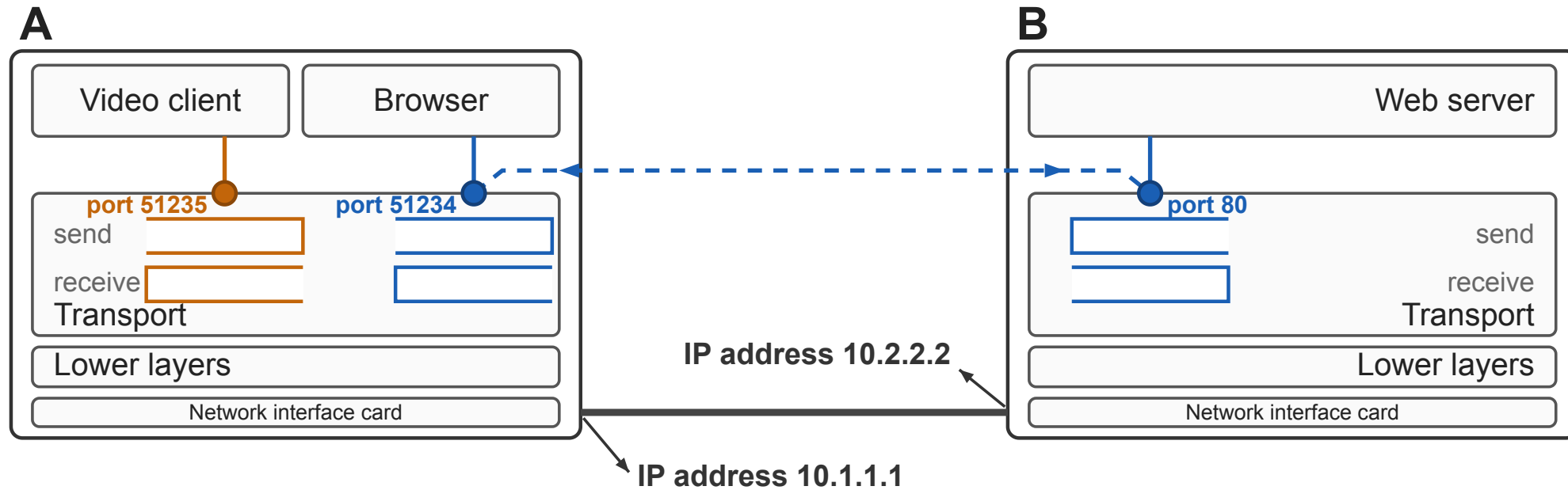
Ports and IP addresses: an example



Ports and IP addresses: an example



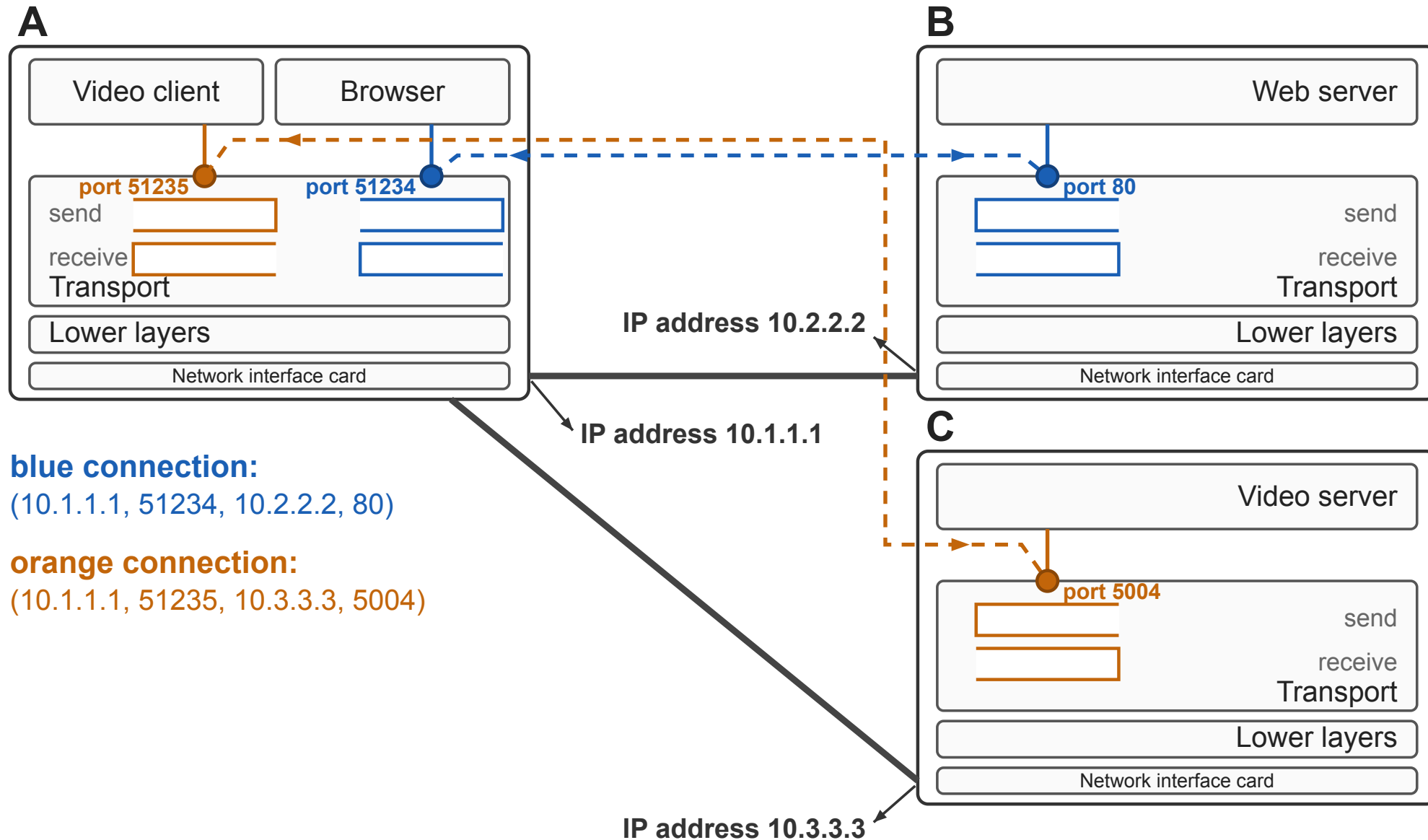
Ports and IP addresses: an example



blue connection:

(10.1.1.1, 51234, 10.2.2.2, 80)

Ports and IP addresses: an example



An example client-server application over TCP sockets

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

An example client-server application over TCP sockets

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

establishing a connection

An example client-server application over TCP sockets

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

establishing a connection

data transfer

An example client-server application over TCP sockets

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

establishing a connection

data transfer

closing the connection

An example client-server application over TCP sockets

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);
addr = make_address("10.2.2.2", 80);
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";
char reply[MAX];
send(s, request, strlen(request), 0);
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);
addr = make_address(my_ip, 80);
bind(s, &addr, sizeof addr);
listen(s, 10);
int c = accept(s, ...);
```

```
char request[MAX];
char reply[] = "HTTP/1.1 200 OK ...";
recv(c, request, MAX, 0);
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

establishing a connection

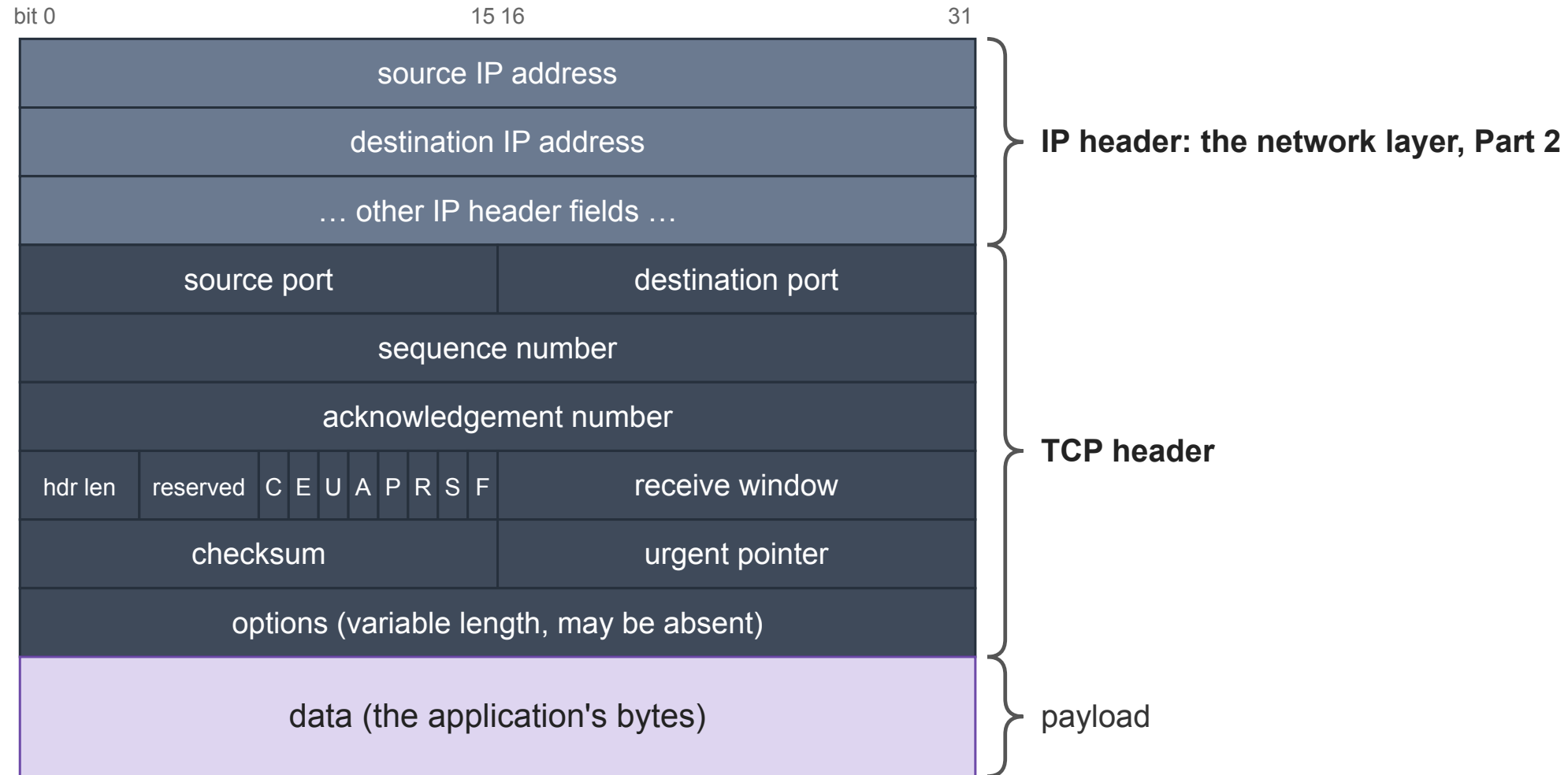
data transfer

closing the connection

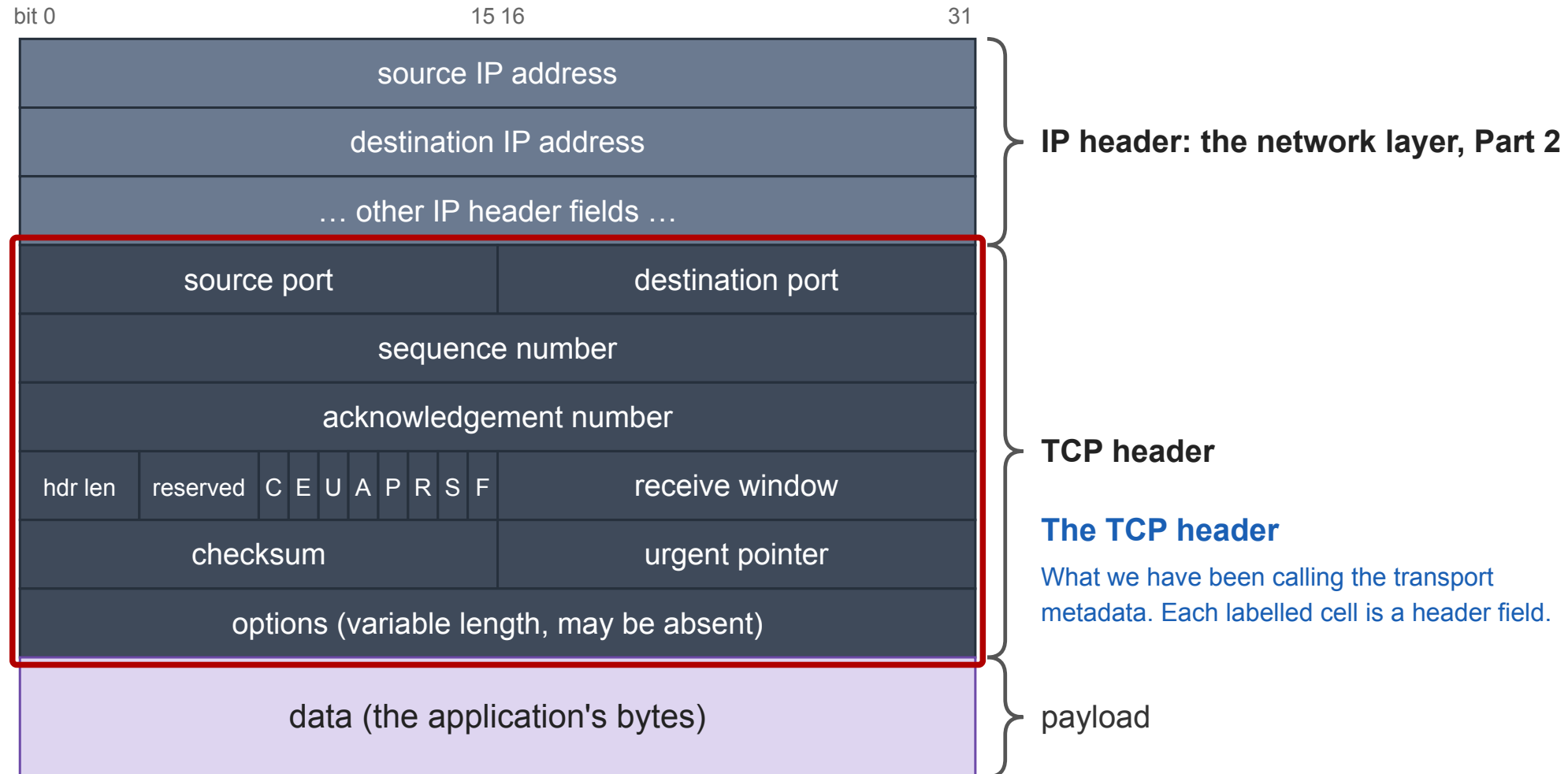
We look at the data transfer first, then come back to establishing and closing a connection.

TCP: data transfer

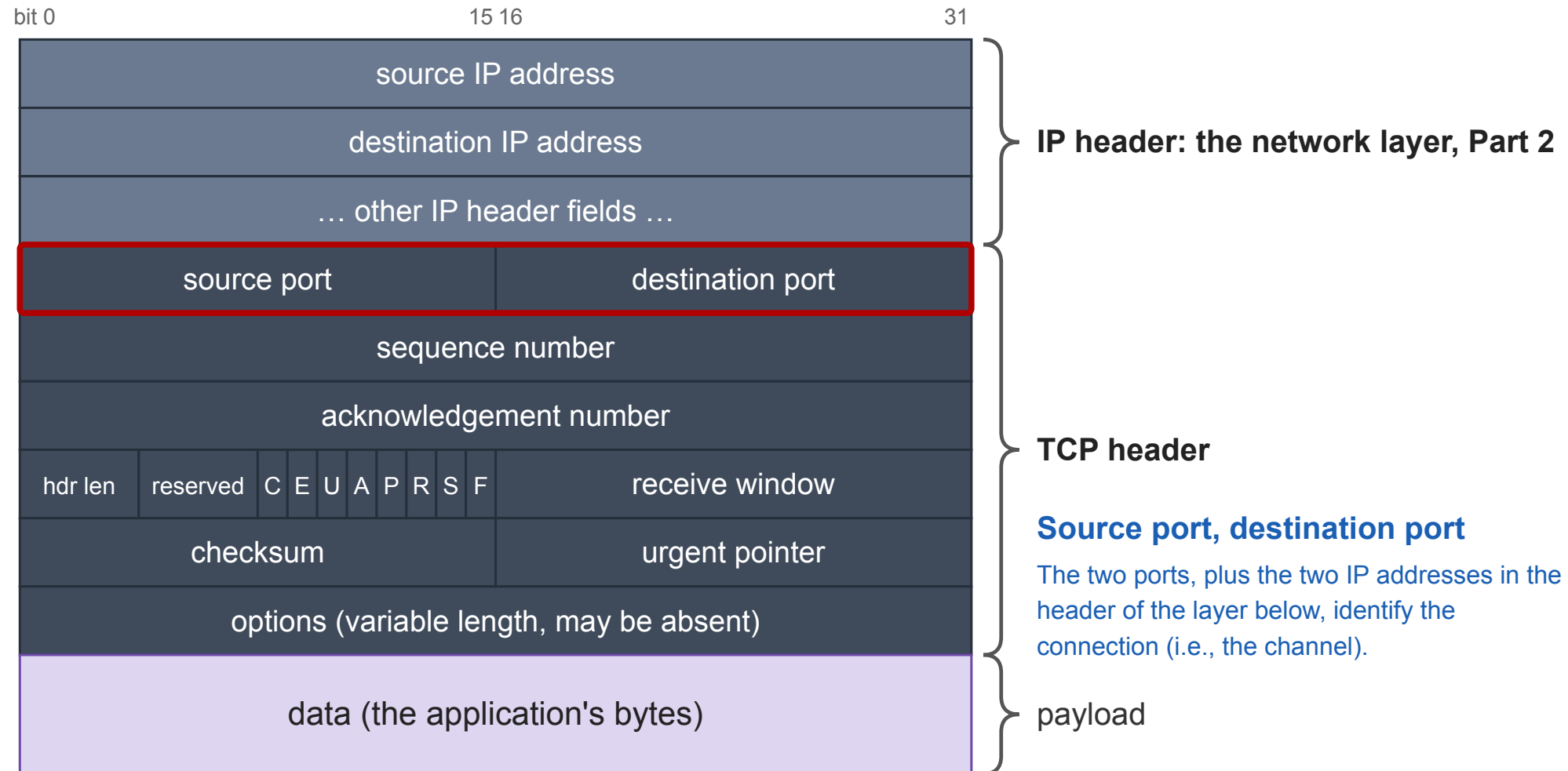
TCP header (i.e., metadata)



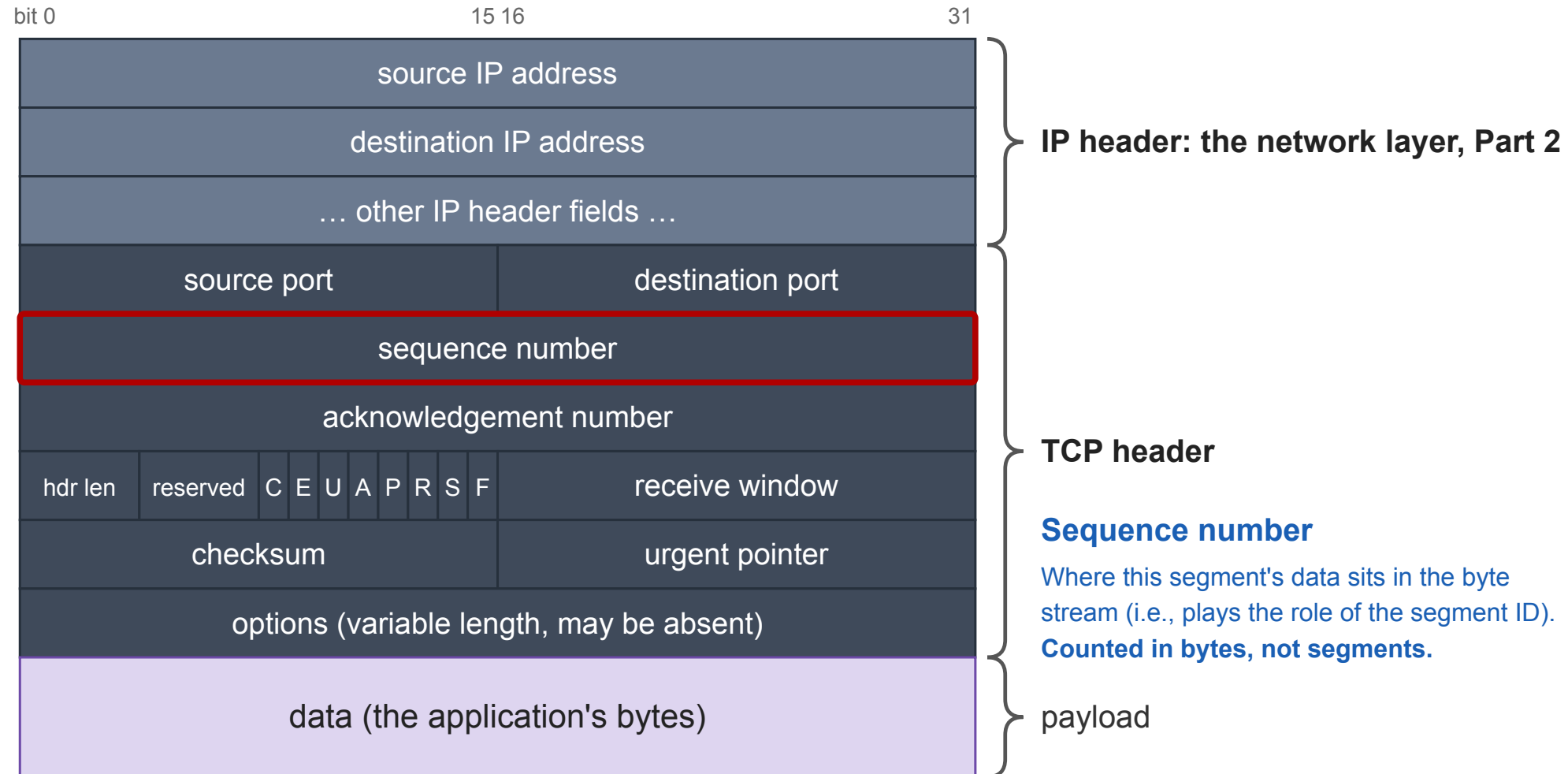
TCP header (i.e., metadata)



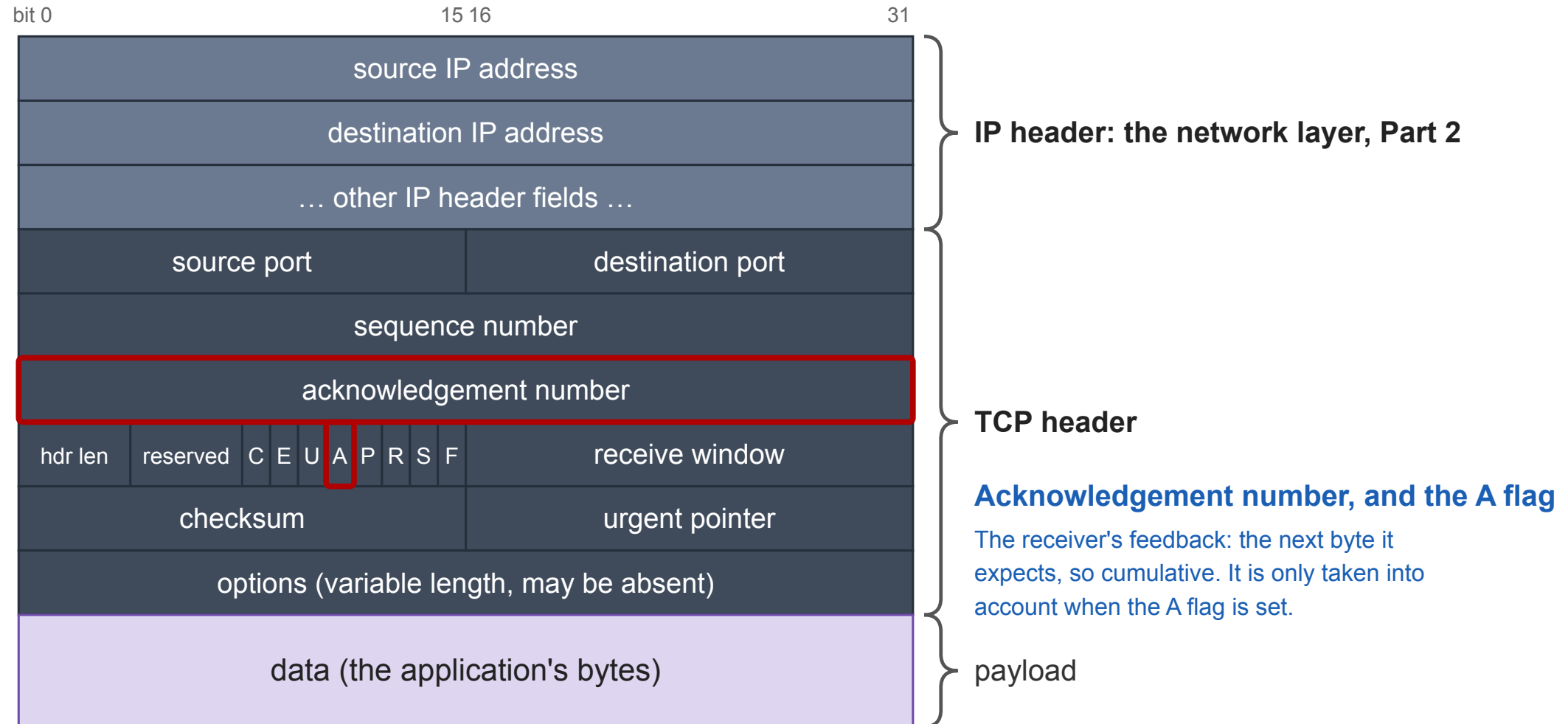
TCP header (i.e., metadata)



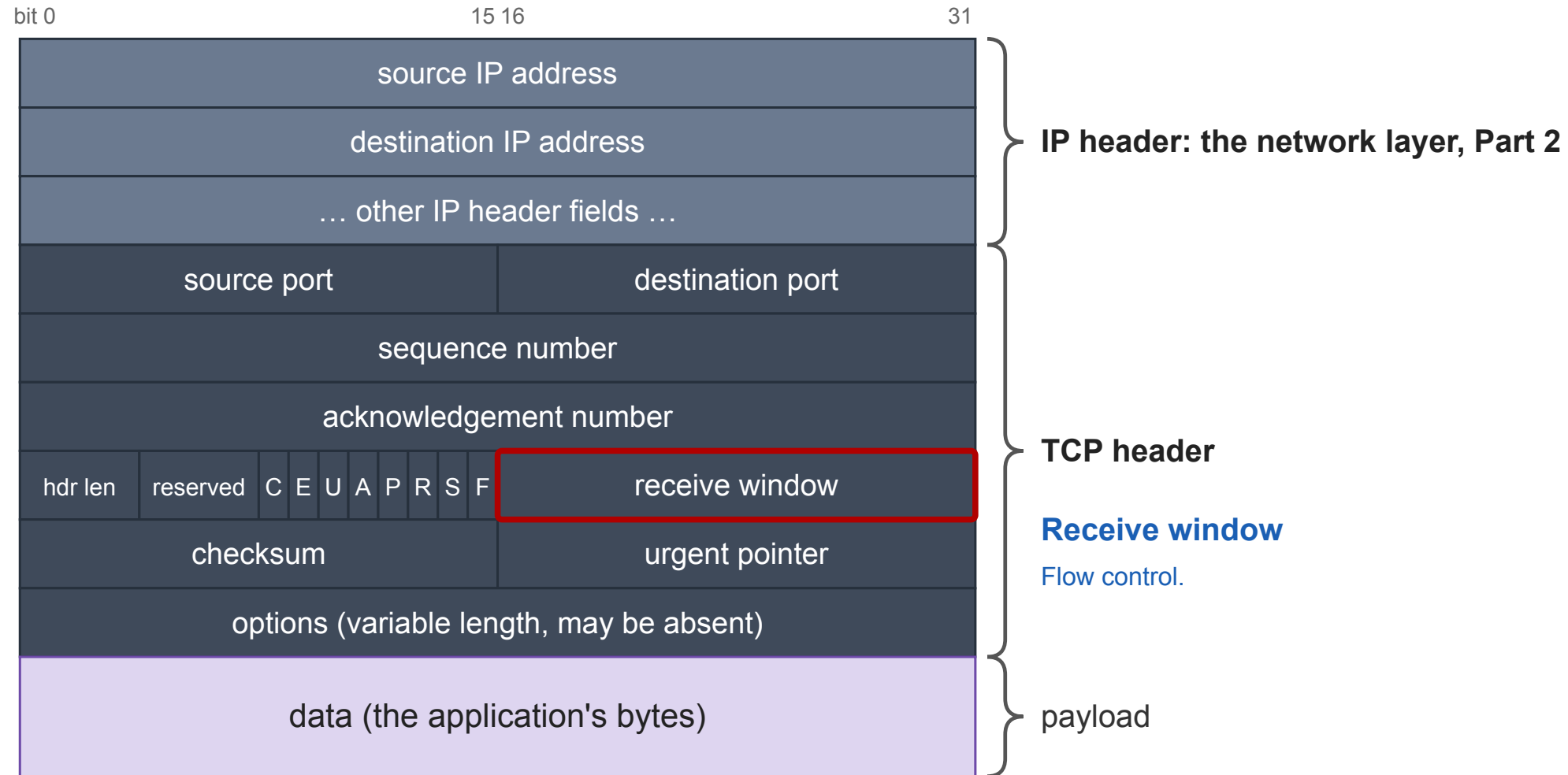
TCP header (i.e., metadata)



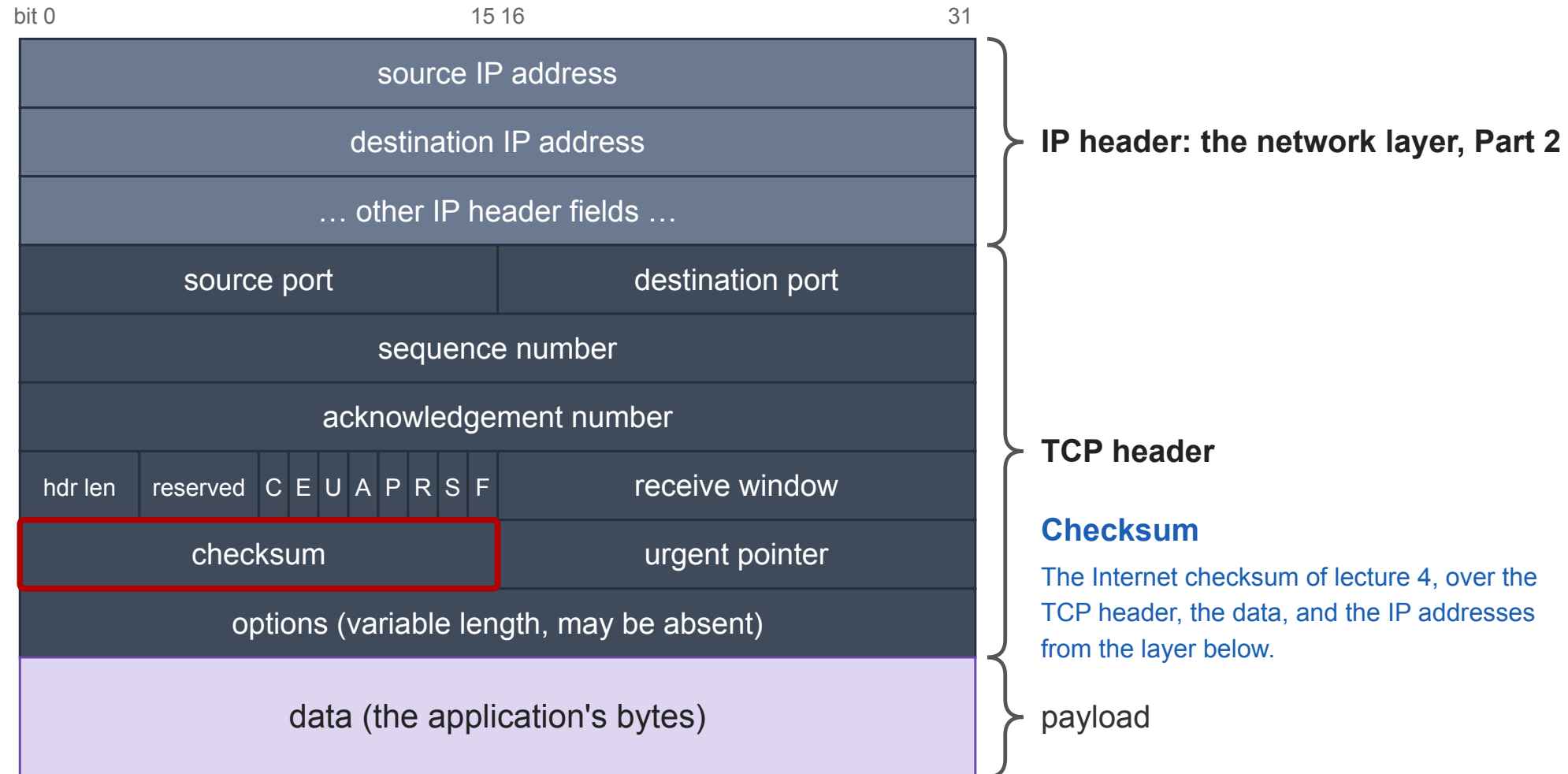
TCP header (i.e., metadata)



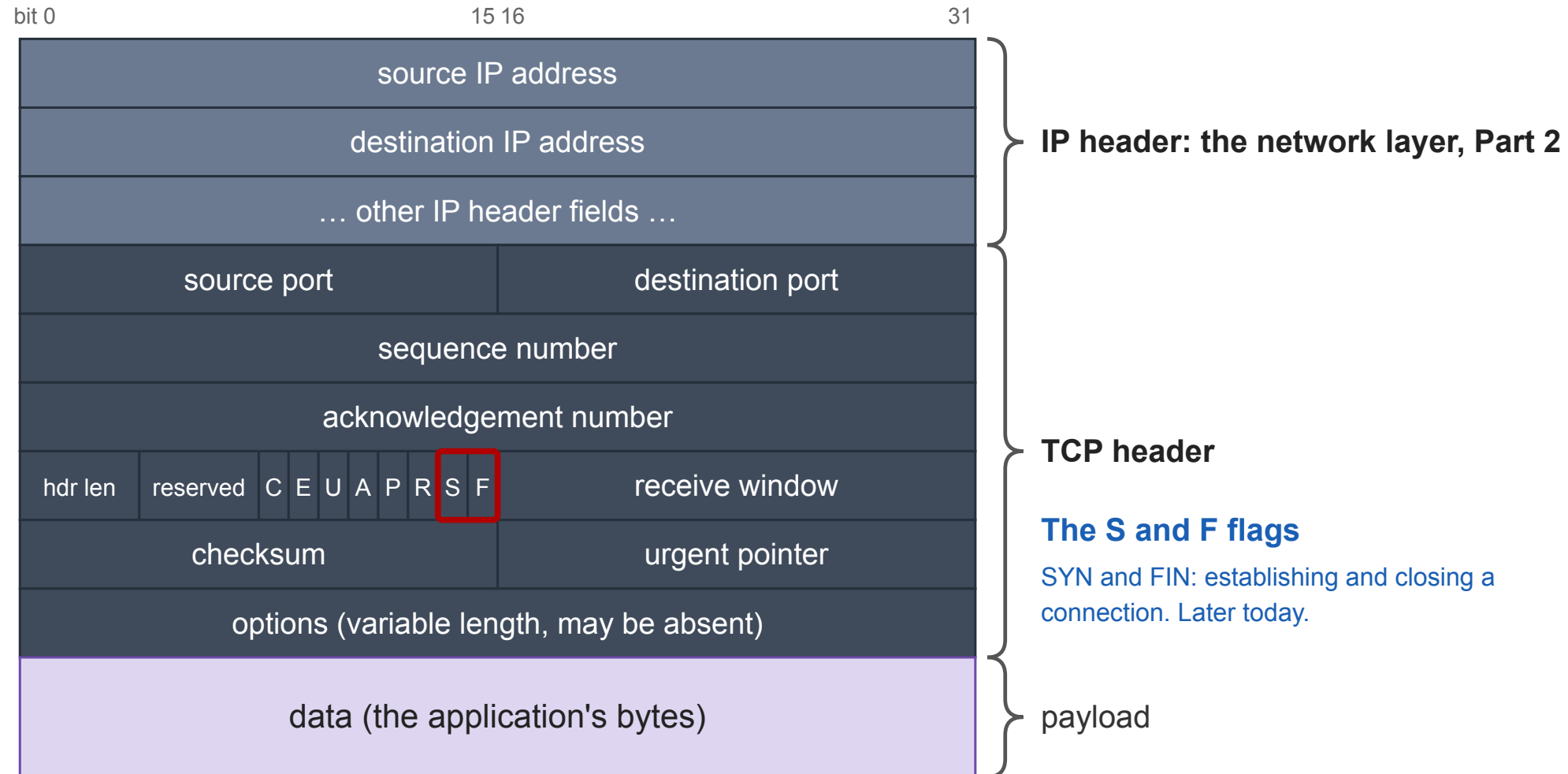
TCP header (i.e., metadata)



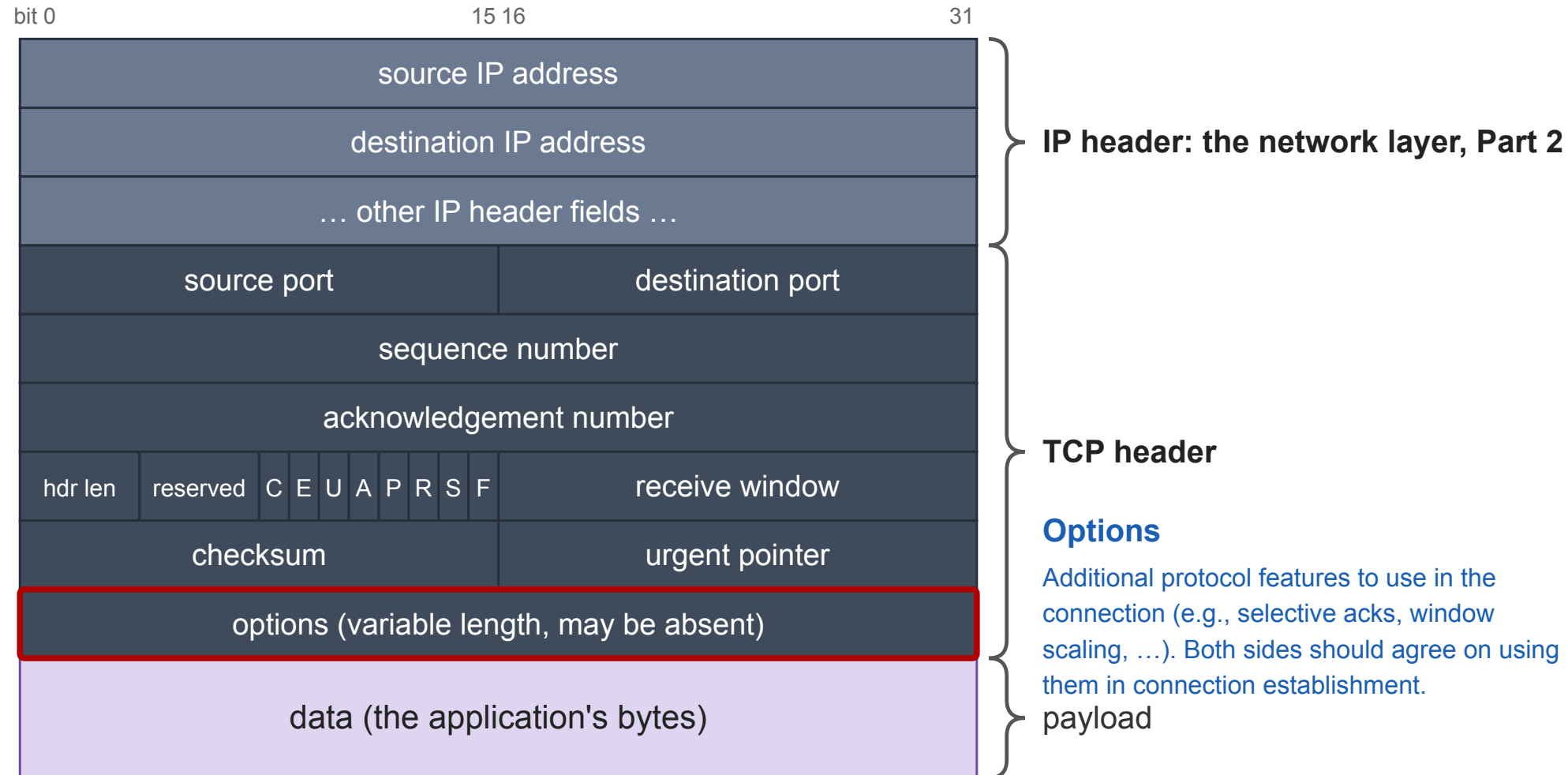
TCP header (i.e., metadata)



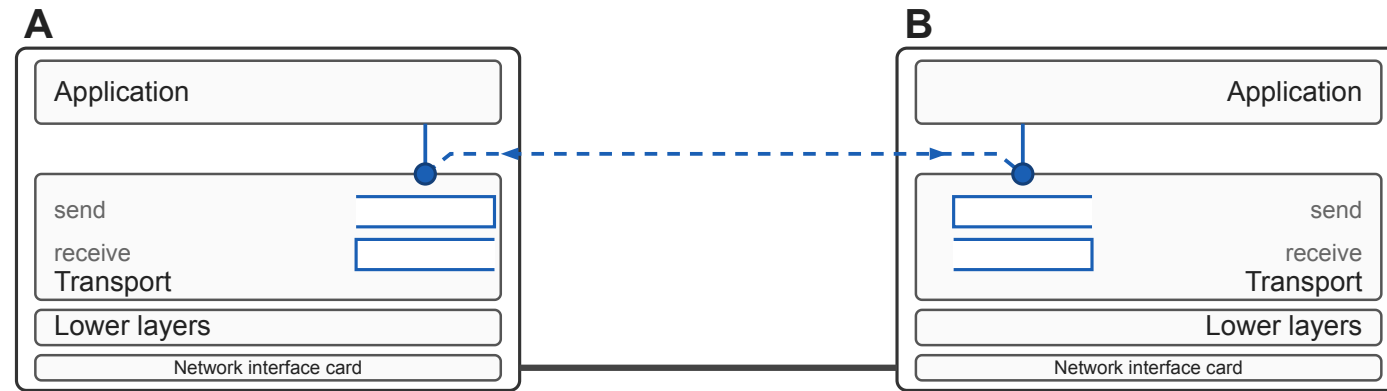
TCP header (i.e., metadata)



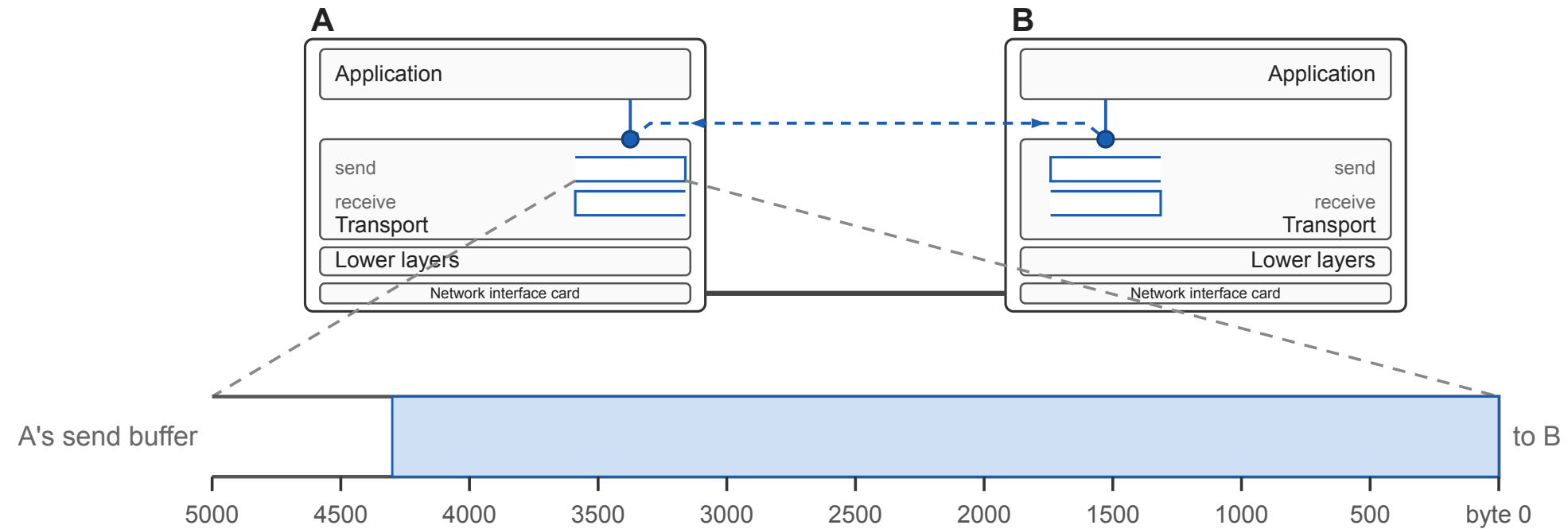
TCP header (i.e., metadata)



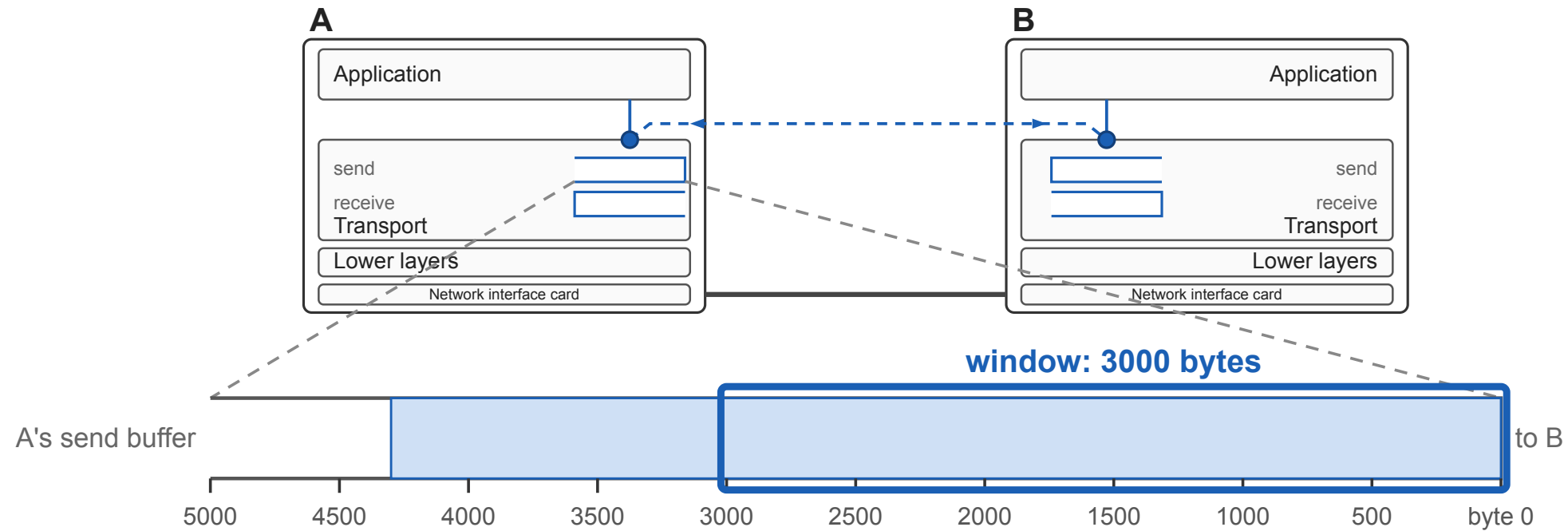
TCP sequence number and sliding window



TCP sequence number and sliding window

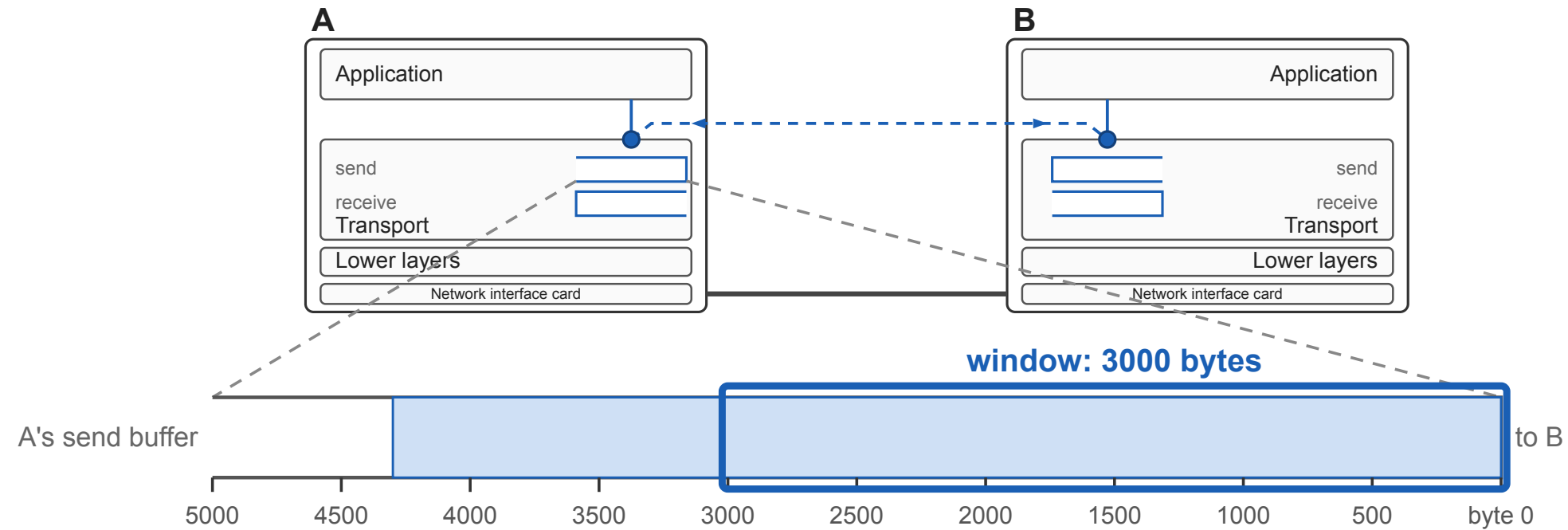


TCP sequence number and sliding window



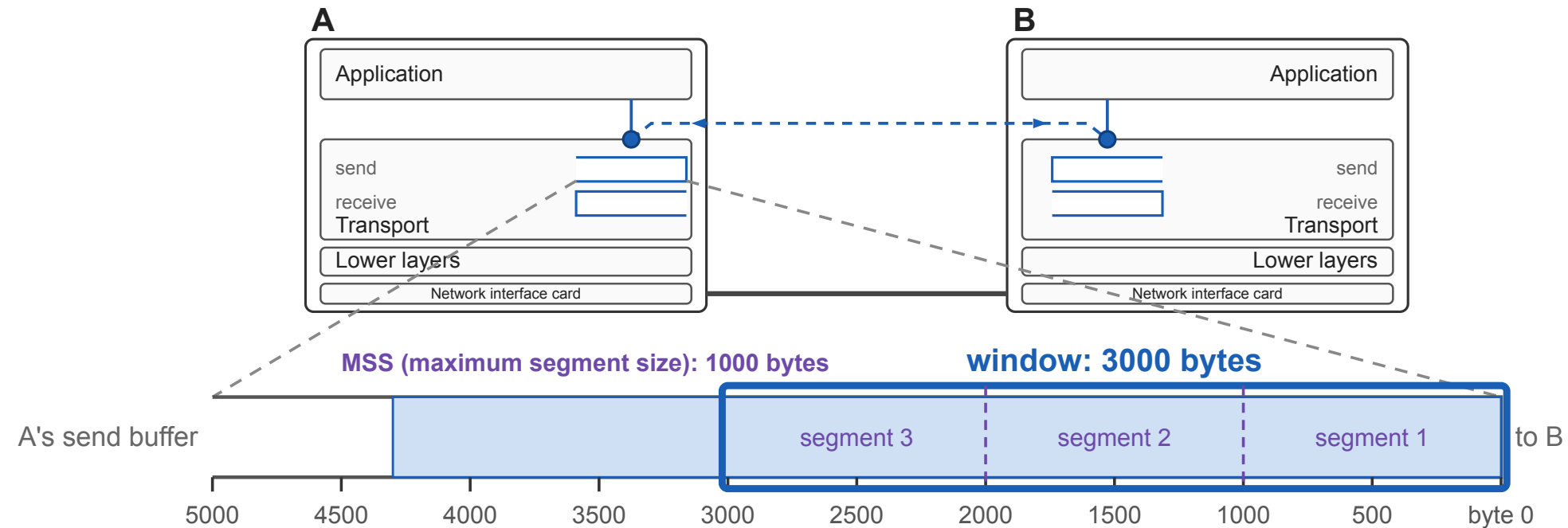
A sliding window over the send buffer (lecture 5), **counted in bytes**.

TCP sequence number and sliding window



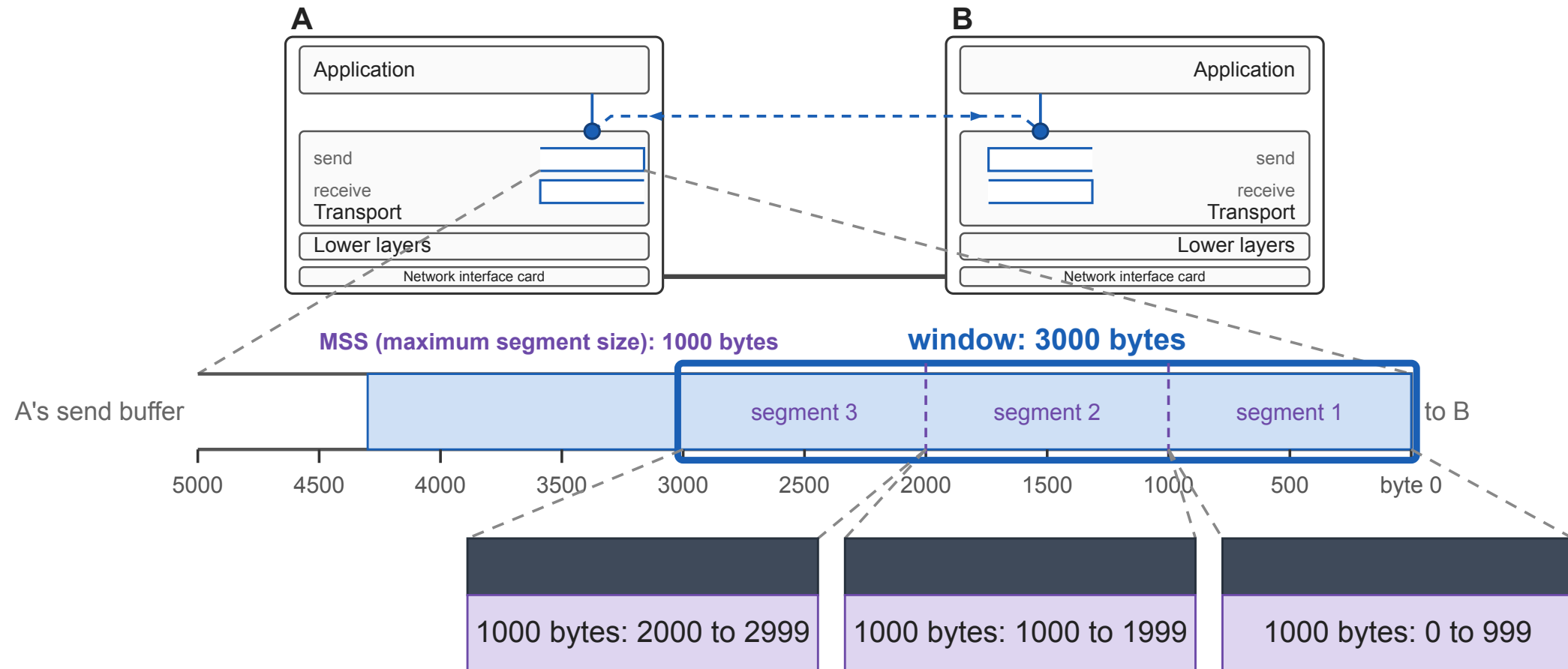
A sliding window over the send buffer (lecture 5), **counted in bytes**.
The receive window field in B's segments sets how big it can be (flow control).

TCP sequence number and sliding window



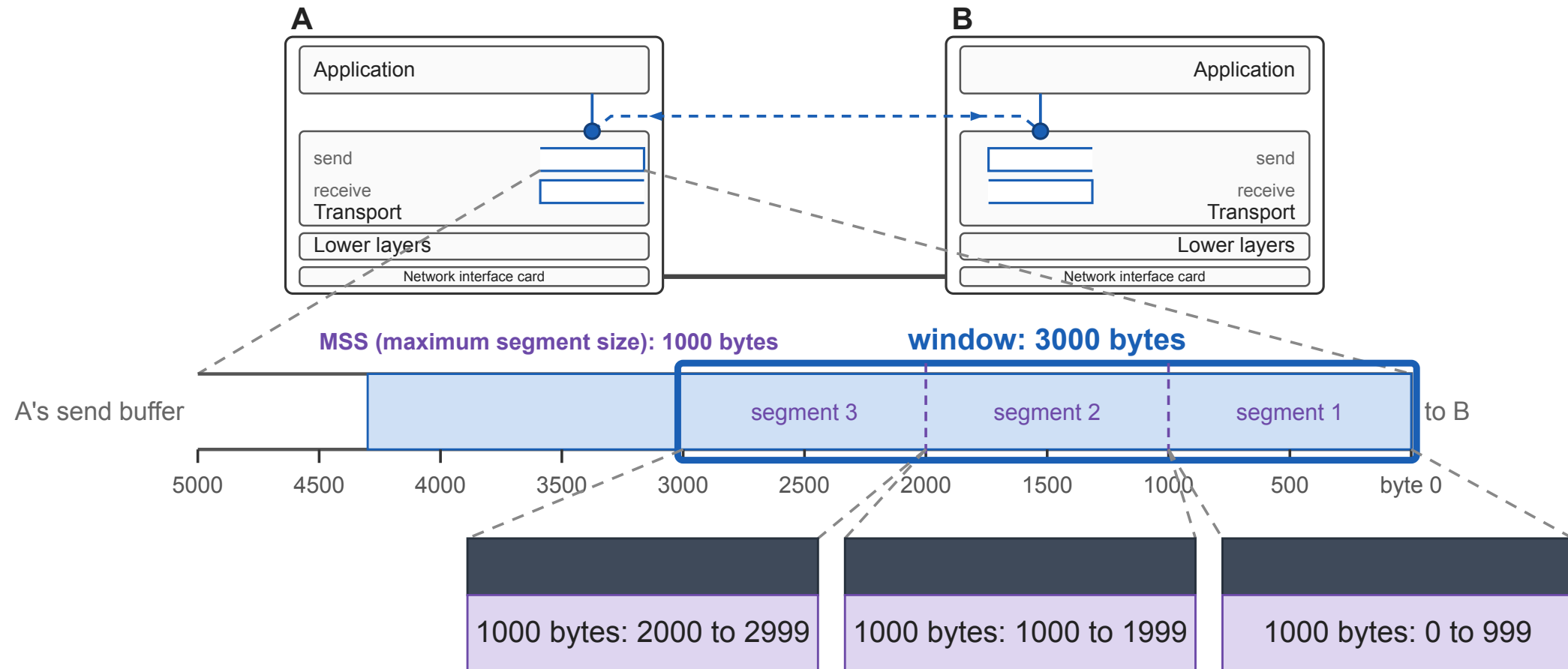
A sliding window over the send buffer (lecture 5), **counted in bytes**.
The receive window field in B's segments sets how big it can be (flow control).

TCP sequence number and sliding window



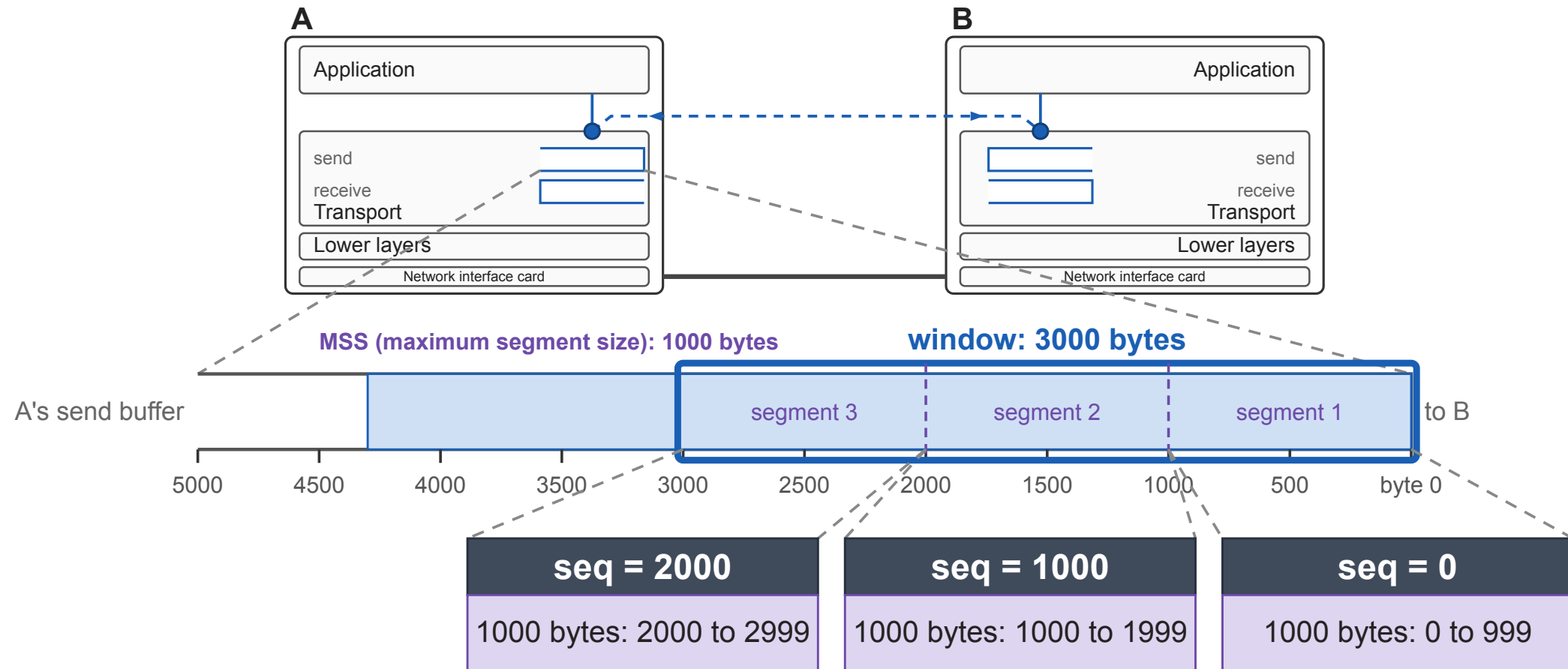
A sliding window over the send buffer (lecture 5), **counted in bytes**.
The receive window field in B's segments sets how big it can be (flow control).

TCP sequence number and sliding window



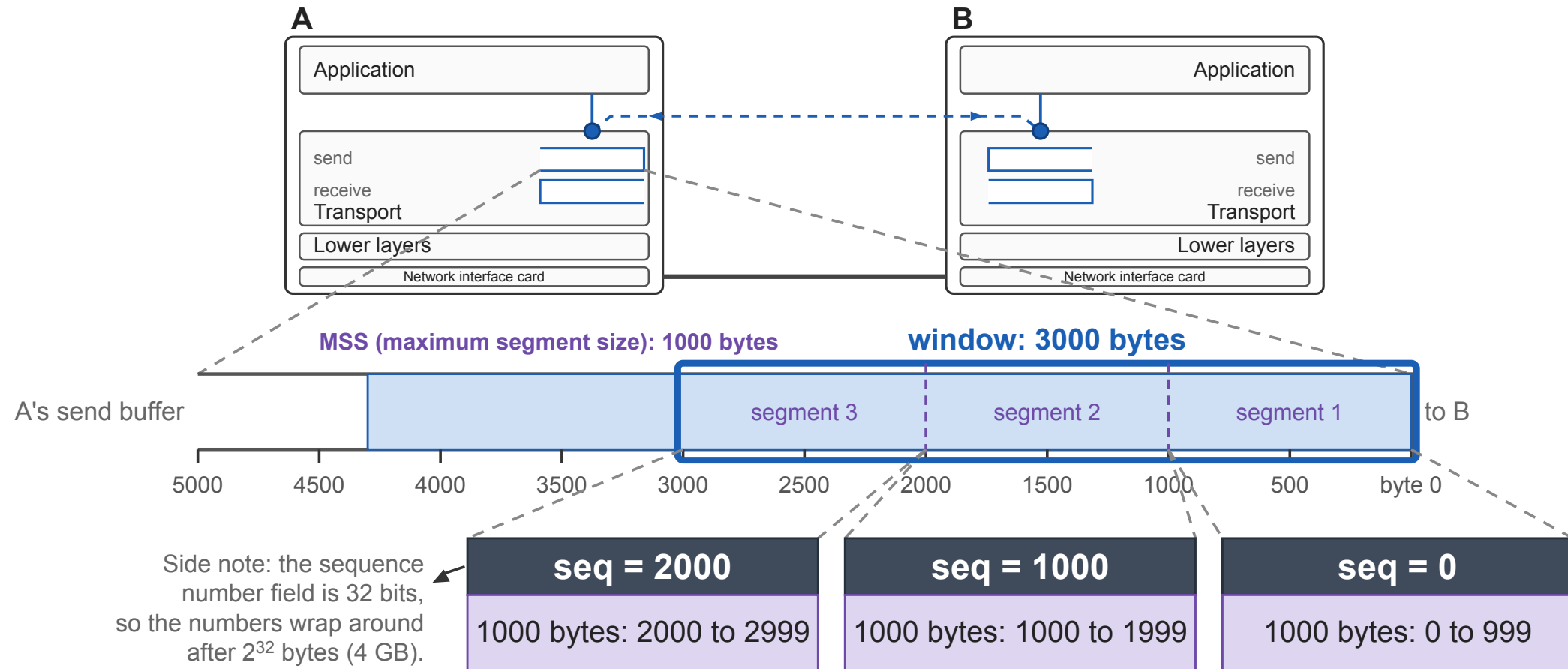
A sliding window over the send buffer (lecture 5), **counted in bytes**.
The receive window field in B's segments sets how big it can be (flow control).
In a TCP segment, the sequence number field is the number associated with the first byte in the payload, counted from the start of the stream.

TCP sequence number and sliding window



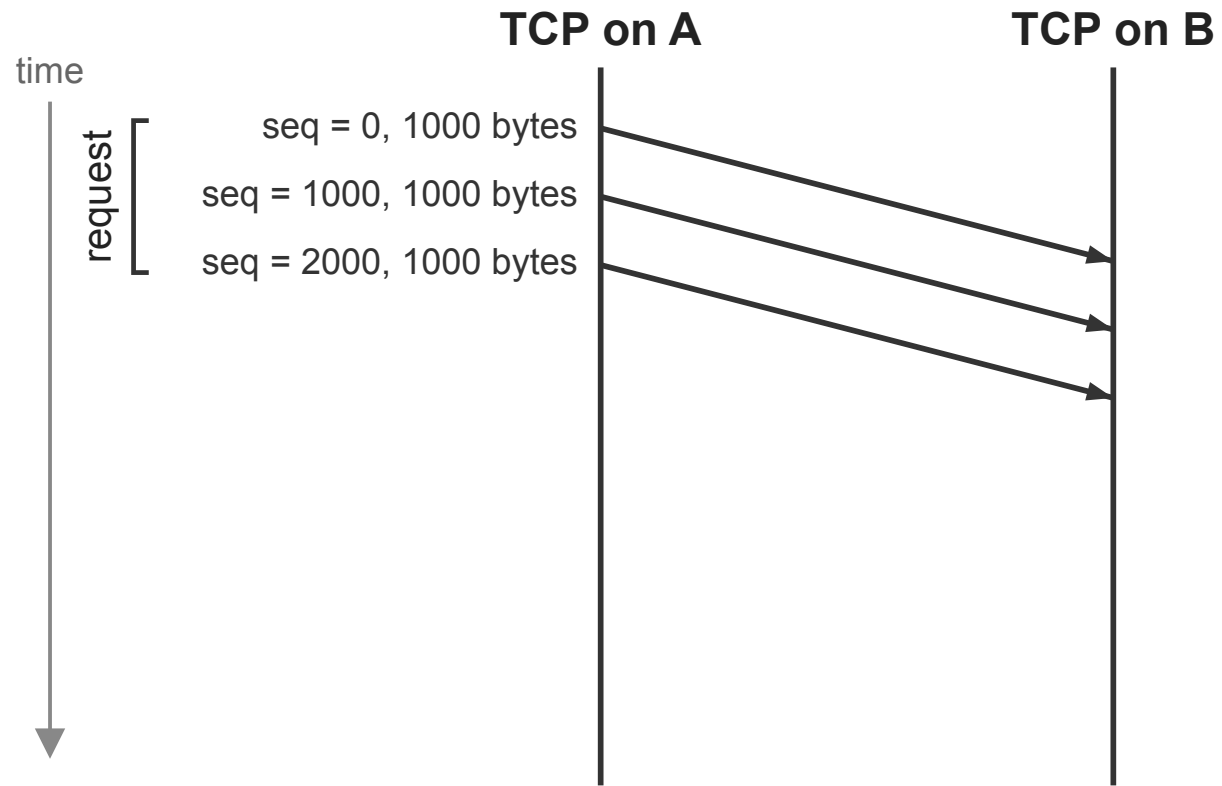
A sliding window over the send buffer (lecture 5), **counted in bytes**.
The receive window field in B's segments sets how big it can be (flow control).
In a TCP segment, the sequence number field is the number associated with the first byte in the payload, counted from the start of the stream.

TCP sequence number and sliding window

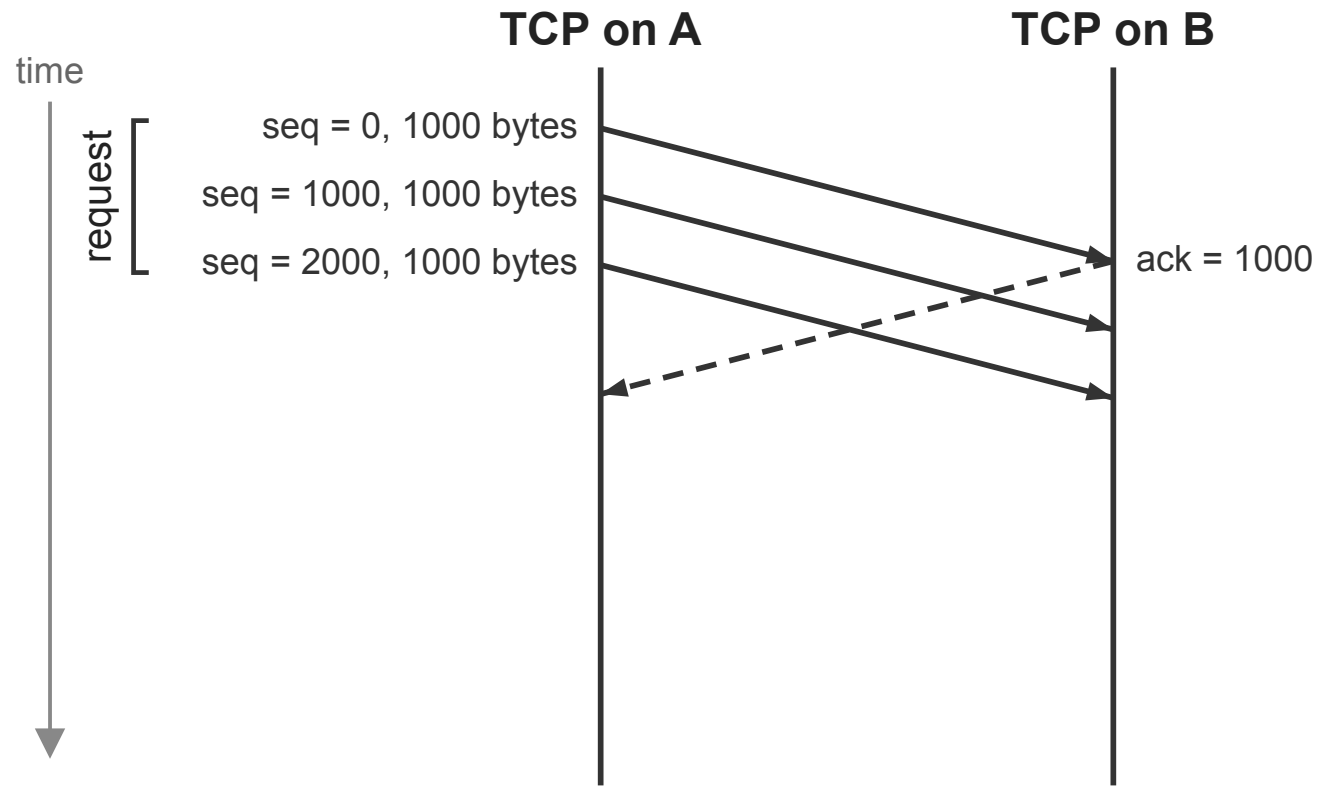


A sliding window over the send buffer (lecture 5), **counted in bytes**.
The receive window field in B's segments sets how big it can be (flow control).
In a TCP segment, the sequence number field is the number associated with the first byte in the payload, counted from the start of the stream.

A TCP data transfer timeline

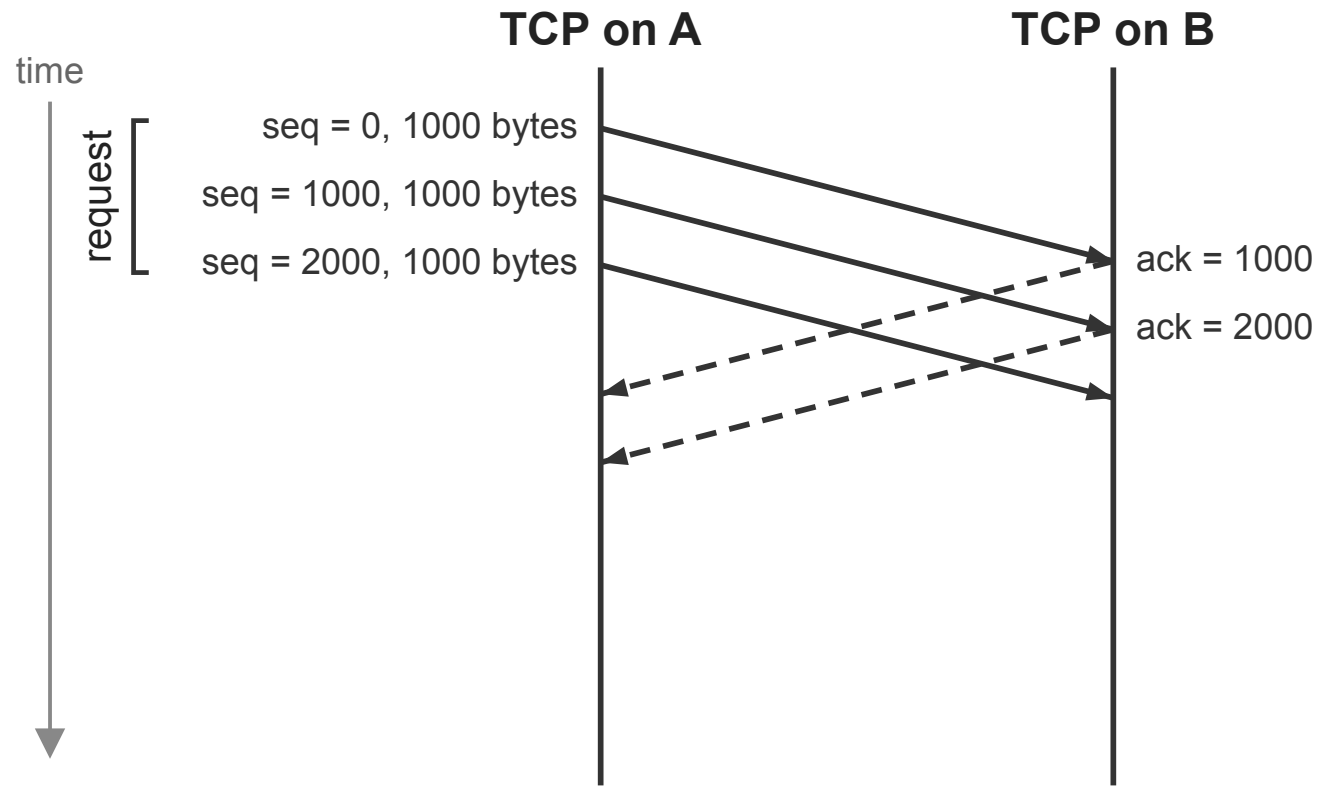


A TCP data transfer timeline



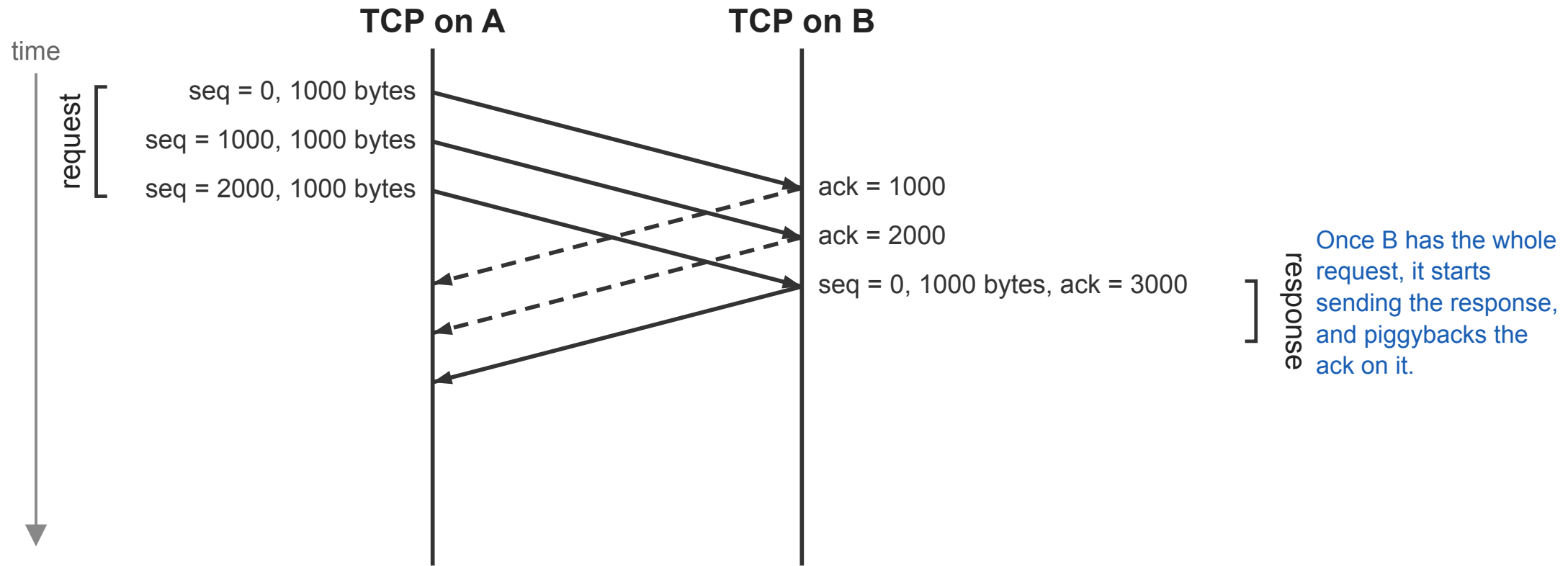
TCP traditionally uses cumulative acks. When the ack number in the header is X, it means the receiver has received all the bytes before **byte number X**.

A TCP data transfer timeline



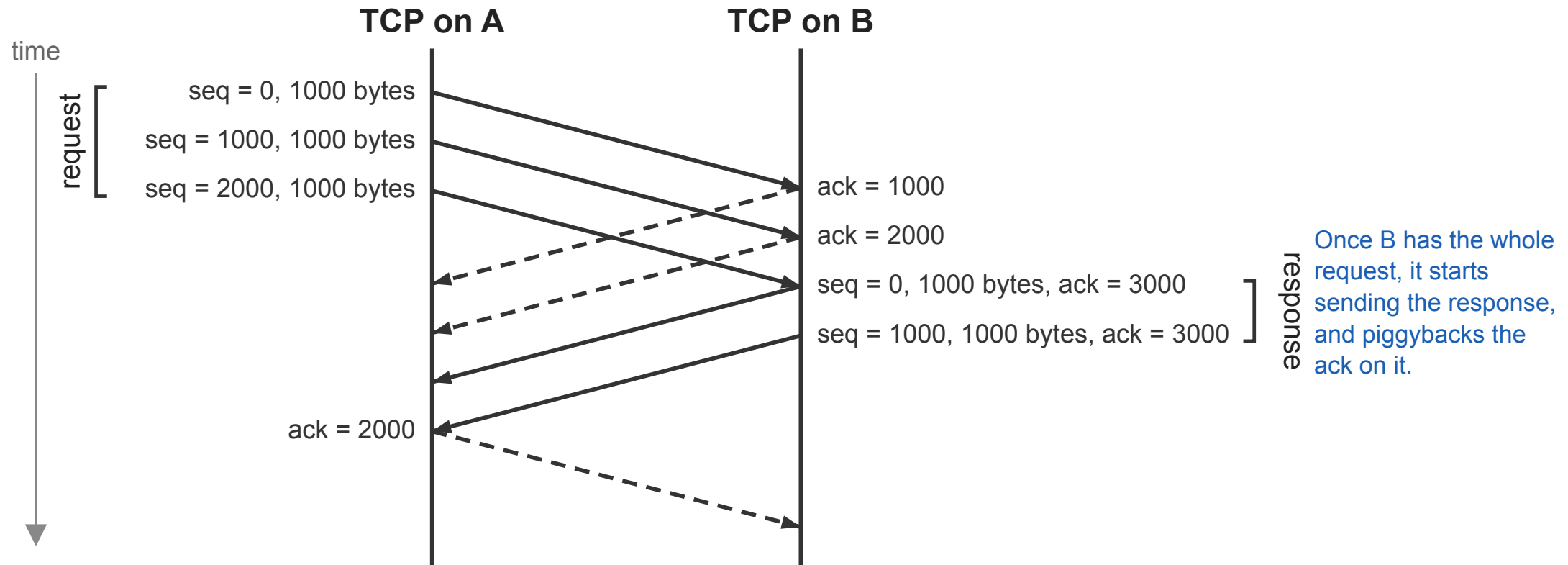
TCP traditionally uses cumulative acks. When the ack number in the header is X, it means the receiver has received all the bytes before **byte number X**.

A TCP data transfer timeline



TCP traditionally uses cumulative acks. When the ack number in the header is X, it means the receiver has received all the bytes before **byte number X**.

A TCP data transfer timeline



TCP traditionally uses cumulative acks. When the ack number in the header is X, it means the receiver has received all the bytes before **byte number X**.

Each direction uses its own sequence number space.

More details on TCP acknowledgements

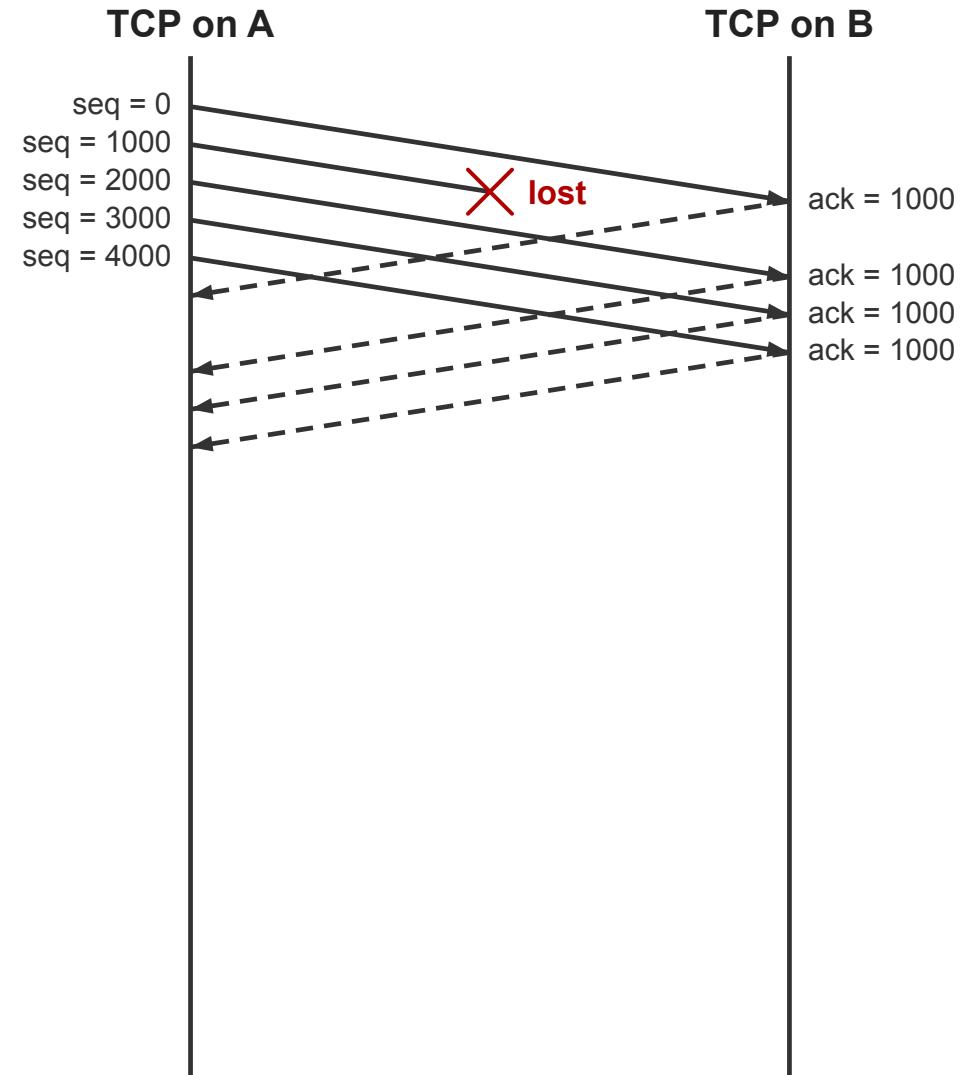
- **Delayed ack:**

- The TCP receiver may not always send one ack per data segment
- When everything arrives in order (no loss), it may wait a configurable time first
- It then sends one ack to acknowledge data from several data segments
- Why is this a good idea? What are the pros and cons?

- **Selective ack (SACK):**

- It is provided as an option: used only if both sides agree on it at the beginning
- Its metadata goes in the option part of the TCP header

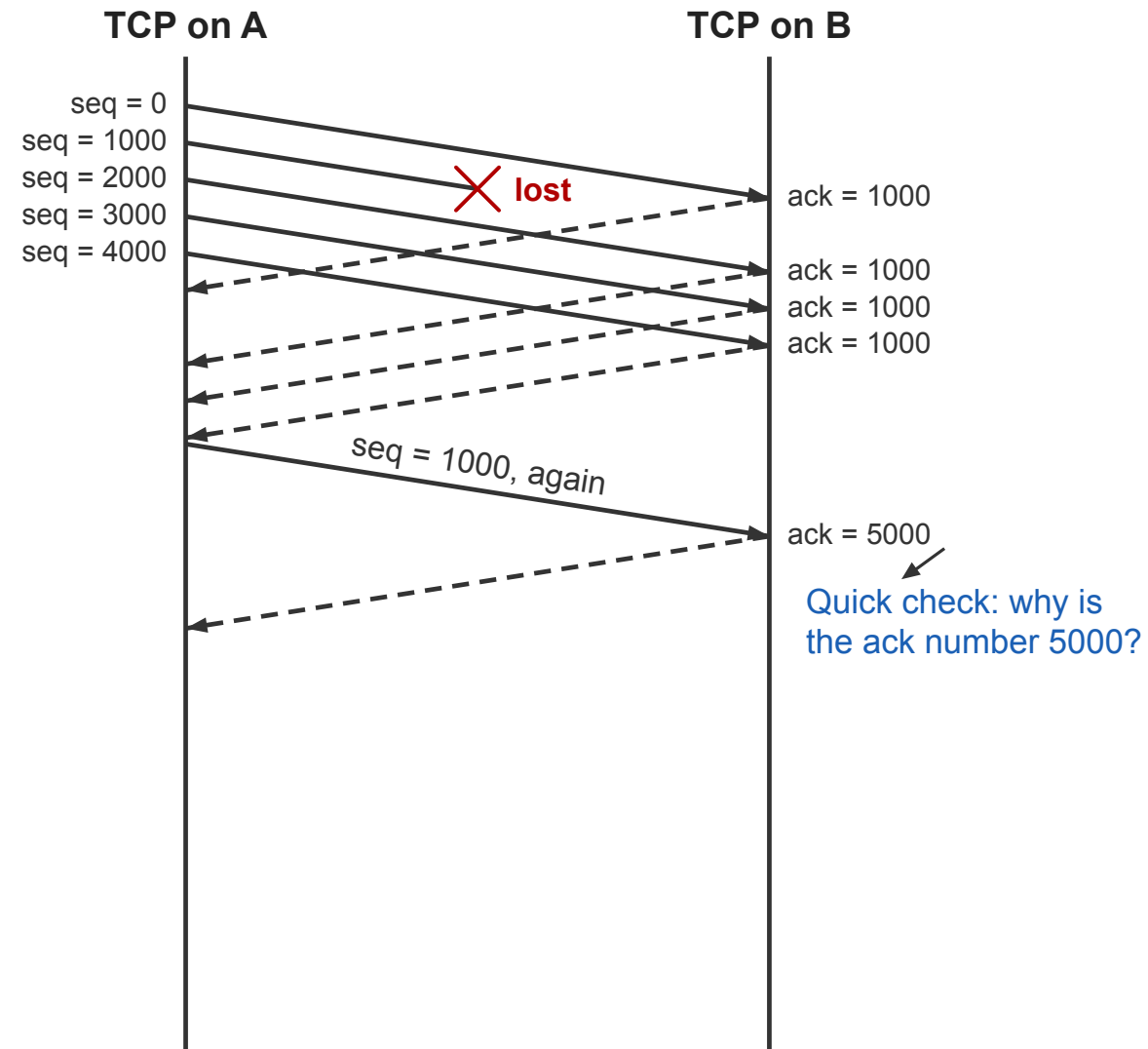
Loss signals: three duplicate acks, and the timer



Loss signals: three duplicate acks, and the timer

ack = 1000 arrives four times, so A has received three duplicate acks.

A resends an MSS-sized segment starting at sequence number 1000.

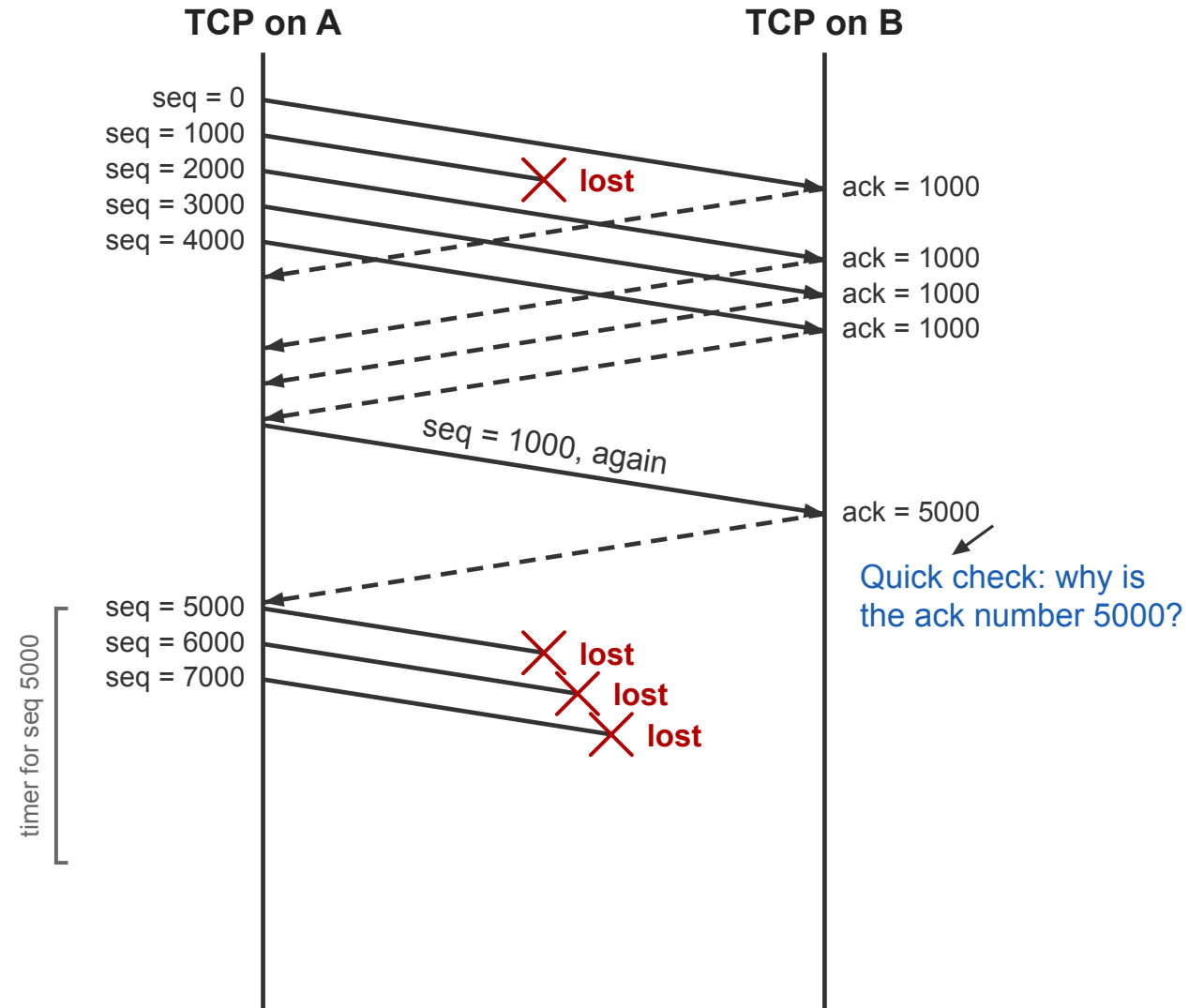


Loss signals: three duplicate acks, and the timer

ack = 1000 arrives four times, so A has received three duplicate acks.

A resends an MSS-sized segment starting at sequence number 1000.

Three more segments, but all are lost.



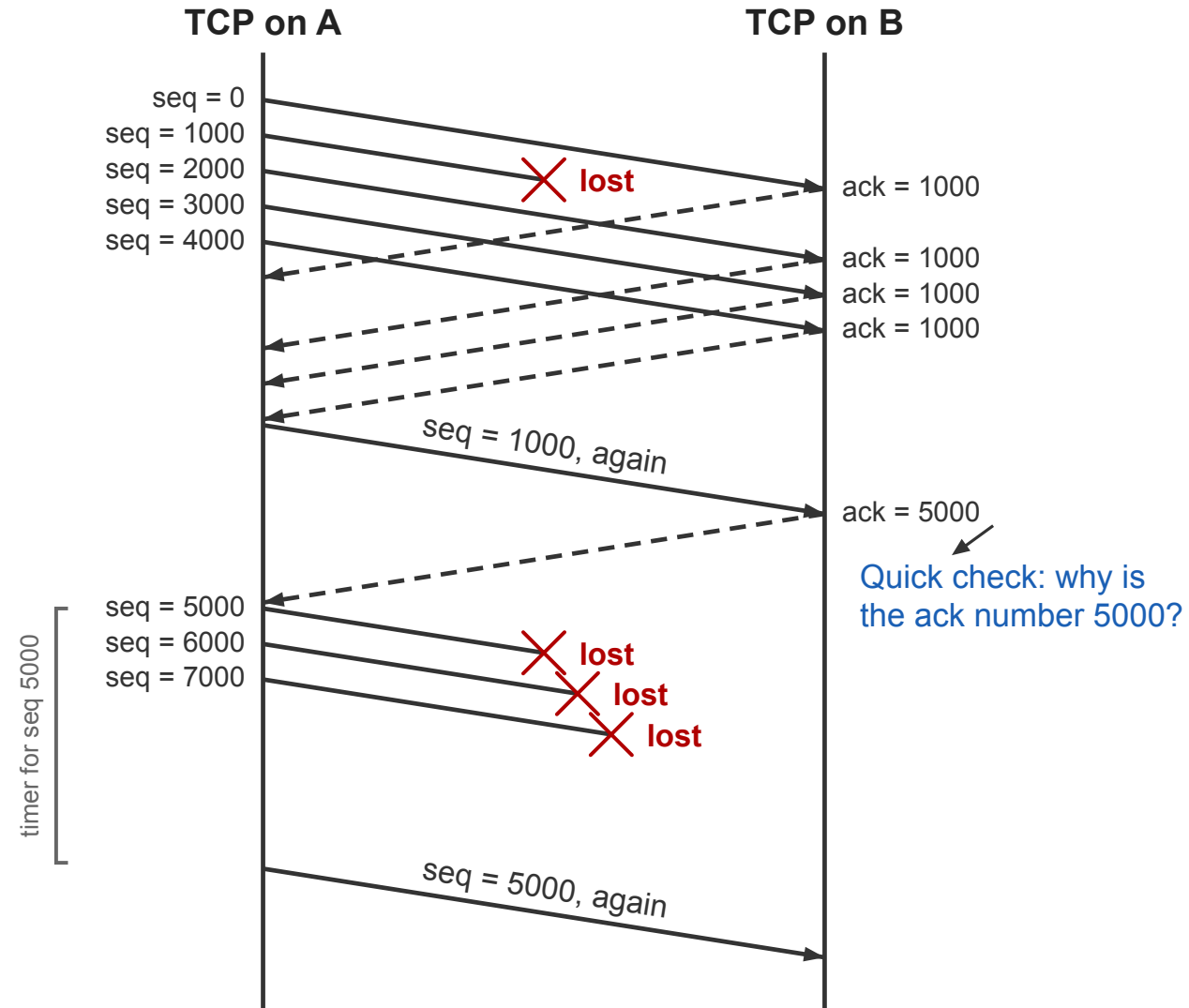
Loss signals: three duplicate acks, and the timer

ack = 1000 arrives four times, so A has received three duplicate acks.

A resends an MSS-sized segment starting at sequence number 1000.

Three more segments, but all are lost.

A resends an MSS-sized segment starting from seq 5000, and doubles the timer duration.



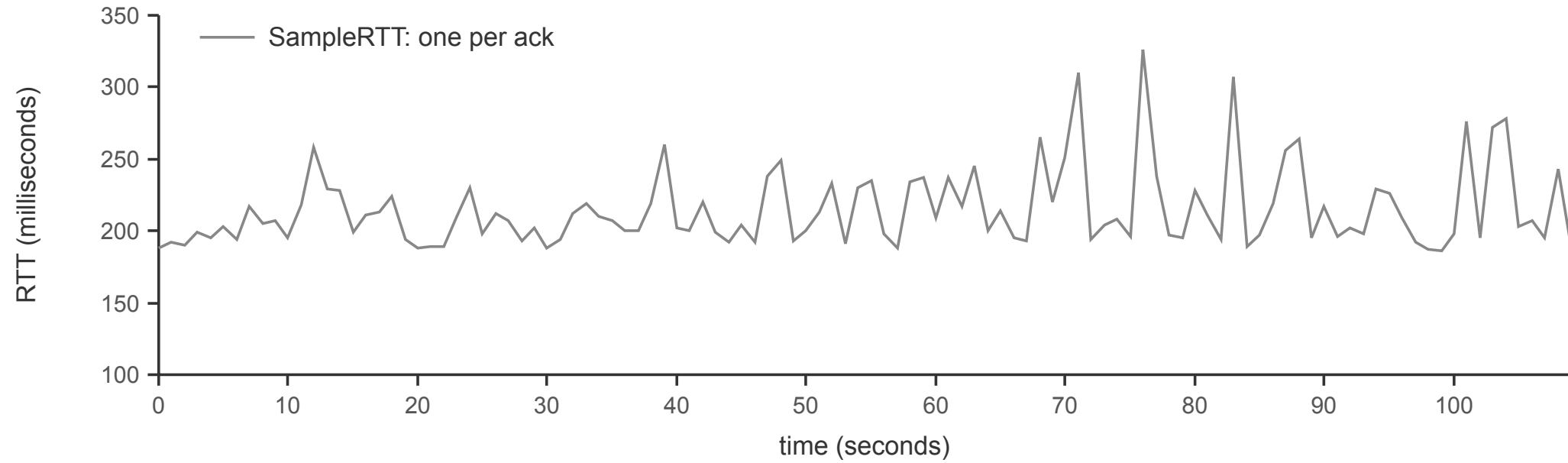
TCP timers: how long to wait?

- TCP keeps one timer during data transfer, for the oldest unacked sequence number
- To set the timer value, it measures the RTT using acks
- Every ack provides a sample: $R = \text{now} - \text{when the corresponding segment left}$

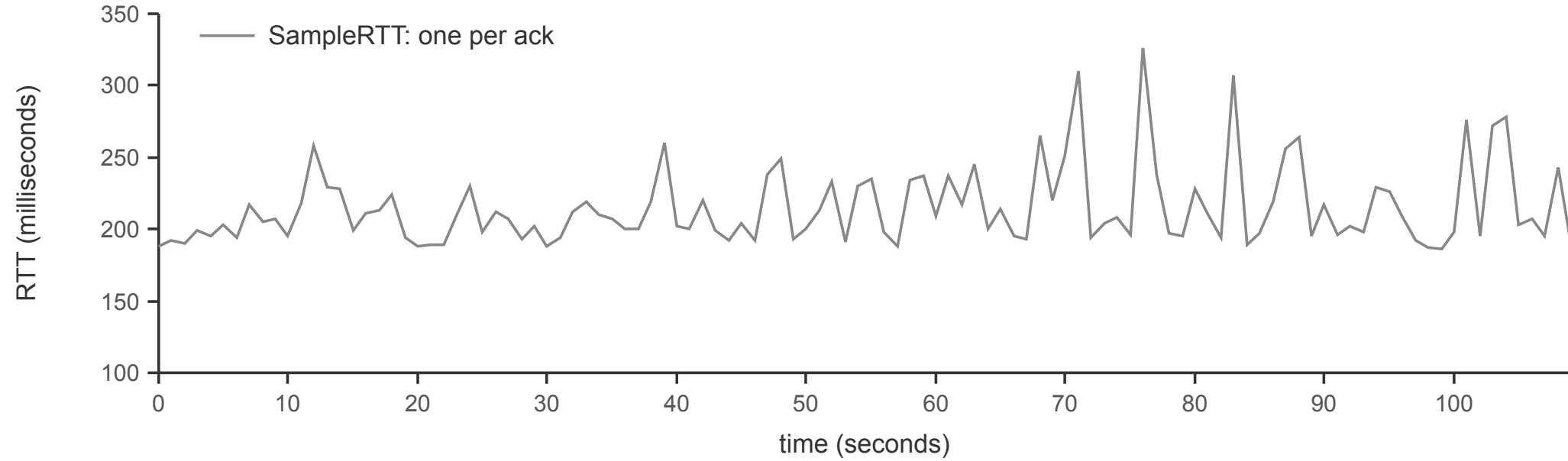
TCP timers: how long to wait?

- TCP keeps one timer during data transfer, for the oldest unacked sequence number
- To set the timer value, it measures the RTT using acks
- Every ack provides a sample: $R = \text{now} - \text{when the corresponding segment left}$
 - No samples from a retransmitted segment: the ack could be for either copy

TCP timers: how long to wait?



TCP timers: how long to wait?



Can't keep changing the timer value per each sample (Why?)

Back to the example application

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

Back to the example application

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

data transfer ✓

Back to the example application

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

establishing a connection

data transfer ✓

closing the connection

Next: establishing a connection, then closing it.

Establishing a connection

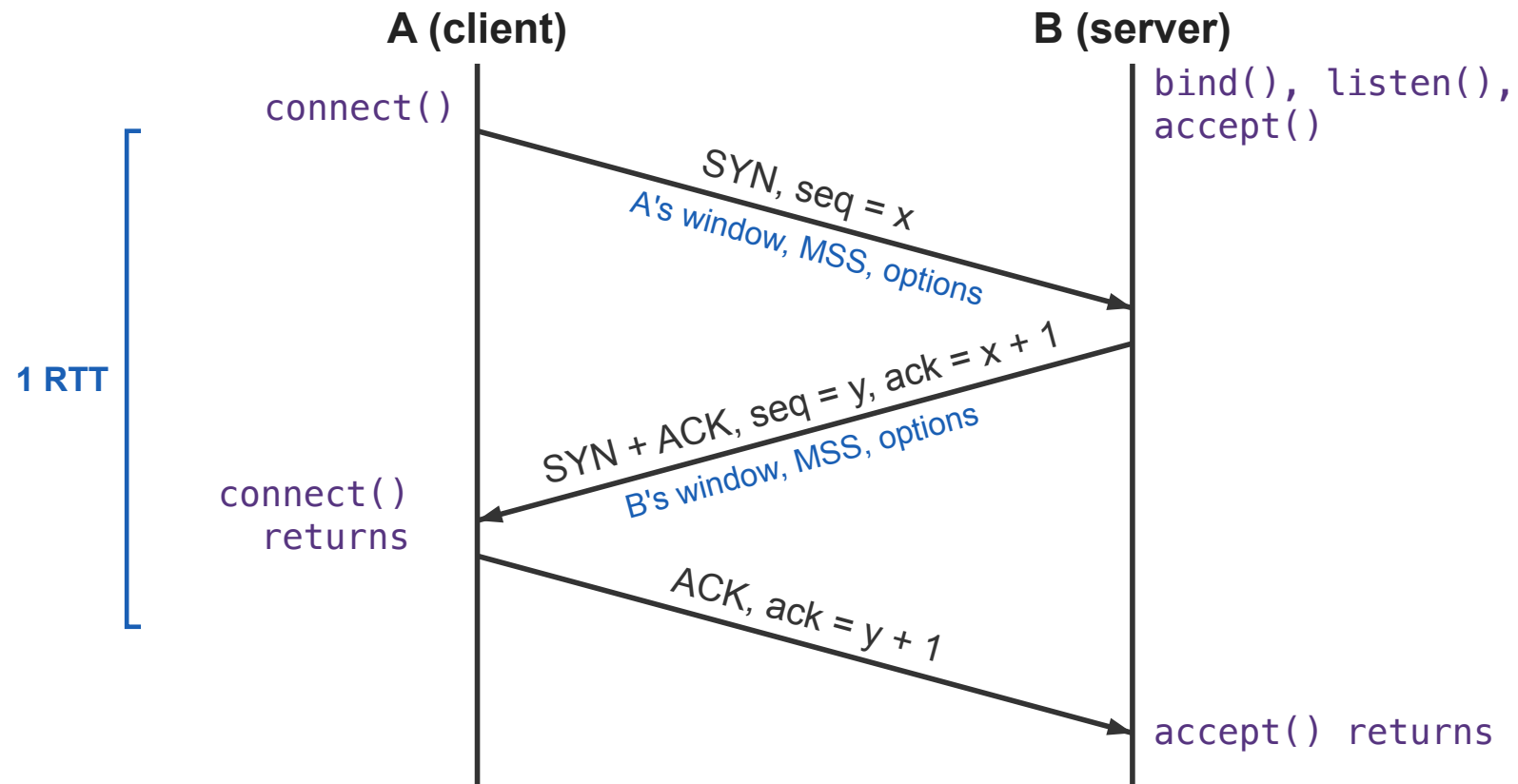
- **Purpose:** check the other side is there, and agree on how to talk
- **For example, agree on:**
 - the first sequence number in each direction
 - each side's receive window and MSS
 - the options both support

Establishing a connection

- **Purpose:** check the other side is there, and agree on how to talk
- **For example, agree on:**
 - the first sequence number in each direction
 - each side's receive window and MSS
 - the options both support

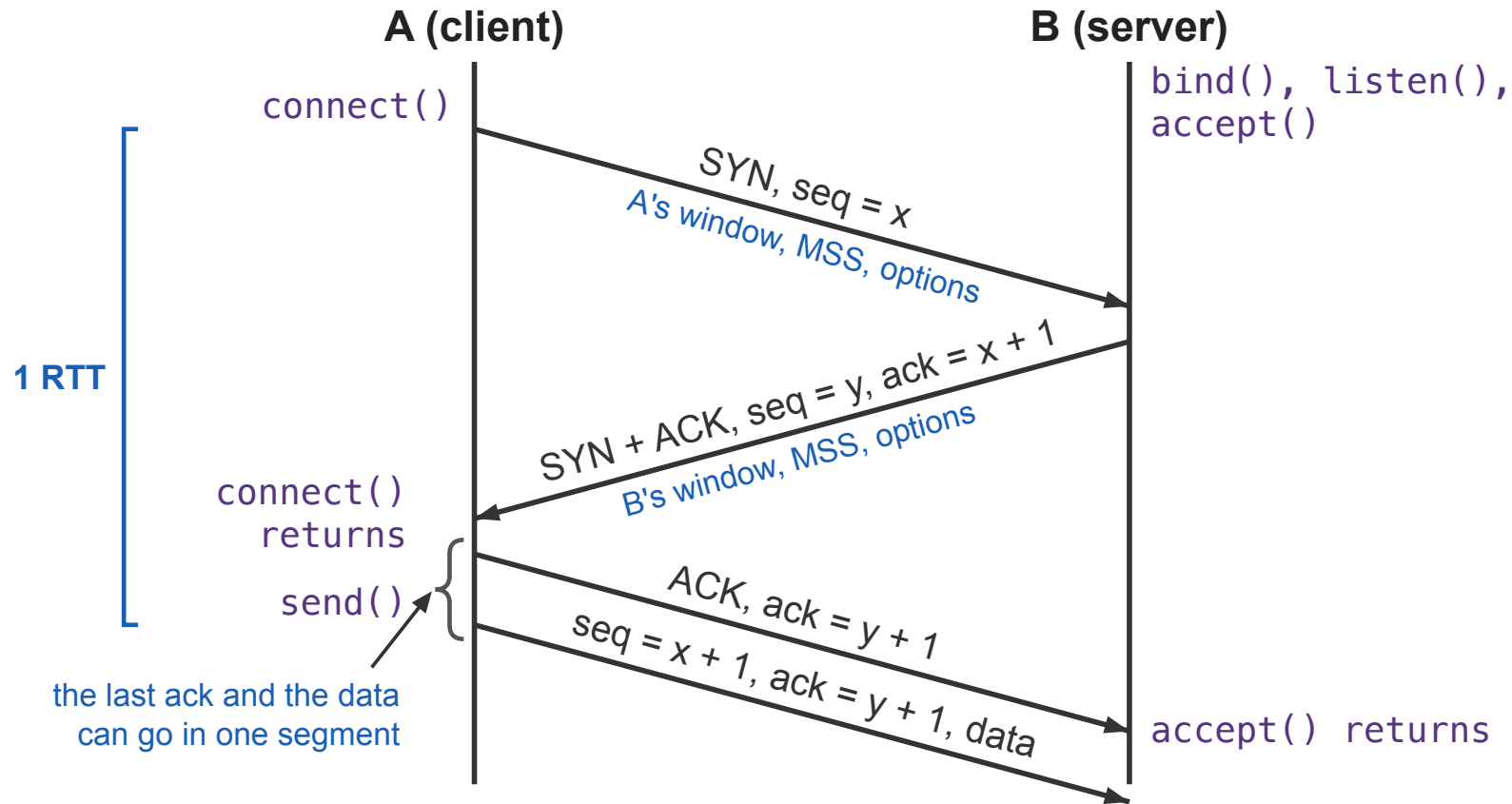
- So far, we assumed both sides start numbering bytes at sequence number 0.
- In TCP, the first sequence number should start from a random number in each connection.
- Why?

Establishing a connection: 3-way handshake



- **SYN:** A's first sequence number x , with its receive window, MSS and the options it supports
- **SYN + ACK:** the same for B: y , its window, MSS and options. $ack = x + 1$ tells A the SYN arrived
- **ACK:** $ack = y + 1$ tells B the SYN + ACK arrived. A can start sending data

Establishing a connection: 3-way handshake



- **SYN:** A's first sequence number x , with its receive window, MSS and the options it supports
- **SYN + ACK:** the same for B: y , its window, MSS and options. $\text{ack} = x + 1$ tells A the SYN arrived
- **ACK:** $\text{ack} = y + 1$ tells B the SYN + ACK arrived. A can start sending data

Back to the example application

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

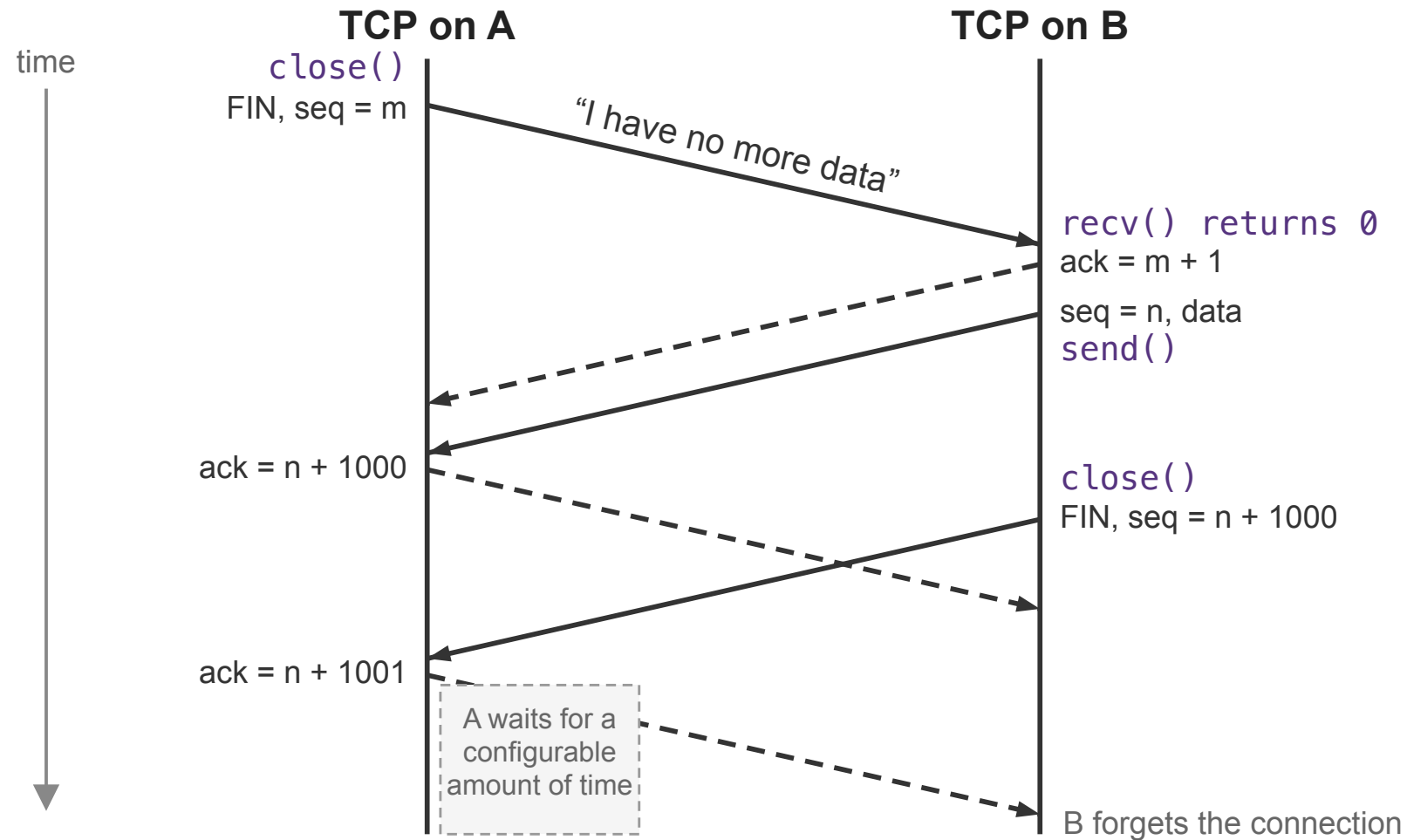
establishing a connection ✓

data transfer ✓

closing the connection

Next: closing the connection.

Closing a connection

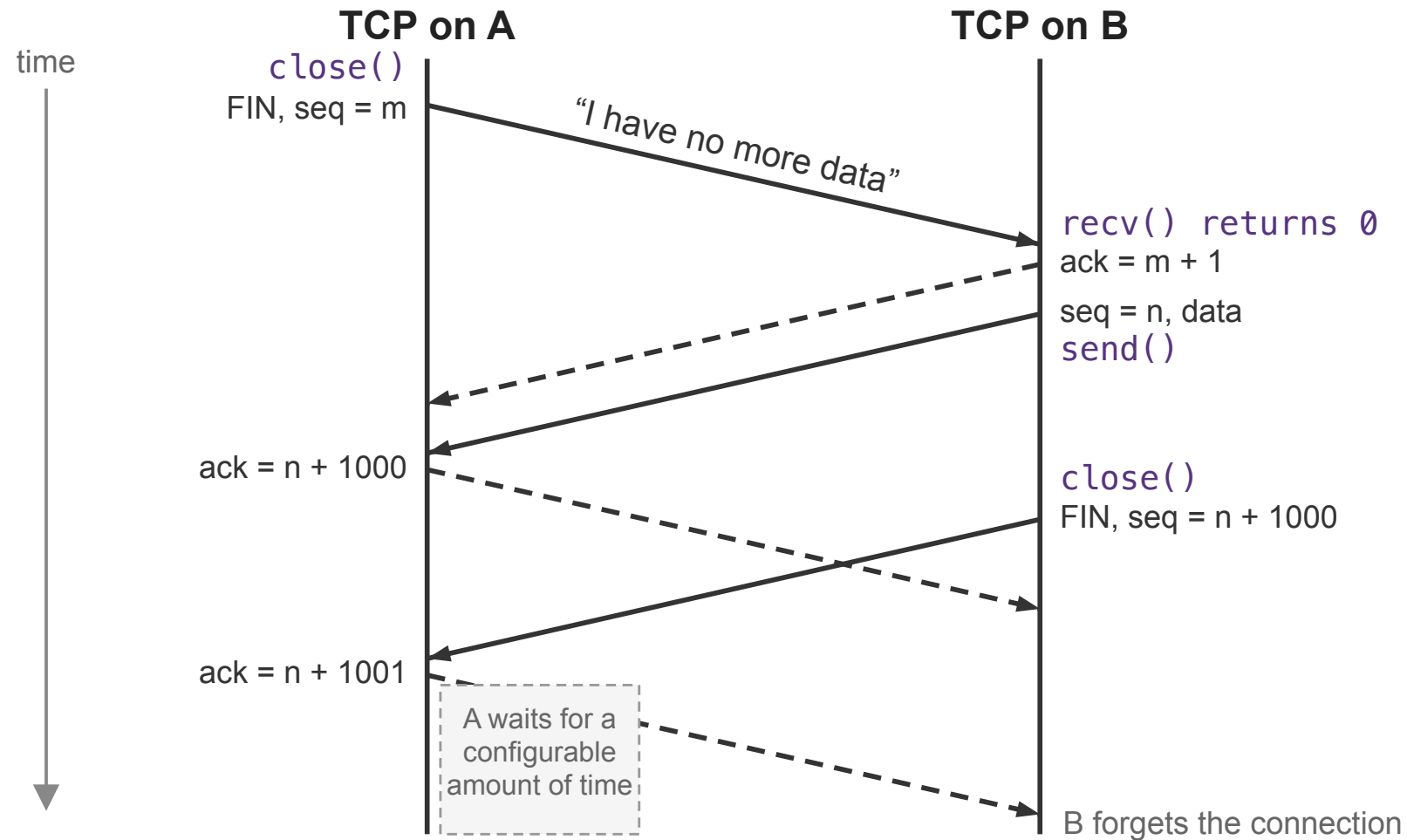


TCP is bidirectional:

- two independent data streams, one each way
- each side closes its own stream independently
- so B can keep sending after A's FIN

Why does A wait?

Closing a connection



TCP is bidirectional:

- two independent data streams, one each way
- each side closes its own stream independently
- so B can keep sending after A's FIN

Why does A wait?

How long should it wait?

Back to the example application

Client (A, the browser)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address("10.2.2.2", 80);  
connect(s, &addr, sizeof addr);
```

```
char request[] = "GET /index.html ...";  
char reply[MAX];  
send(s, request, strlen(request), 0);  
recv(s, reply, MAX, 0);
```

```
close(s);
```

Server (B, the web server)

```
int s = socket(AF_INET, SOCK_STREAM, 0);  
addr = make_address(my_ip, 80);  
bind(s, &addr, sizeof addr);  
listen(s, 10);  
int c = accept(s, ...);
```

```
char request[MAX];  
char reply[] = "HTTP/1.1 200 OK ...";  
recv(c, request, MAX, 0);  
send(c, reply, strlen(reply), 0);
```

```
close(c);
```

establishing a connection ✓

data transfer ✓

closing the connection ✓

QUIC: Quick UDP Internet Connections

- Google, 2012 → IETF standard, 2021 (RFC 9000)
- Some conceptual differences from TCP:

QUIC: Quick UDP Internet Connections

- Google, 2012 → IETF standard, 2021 (RFC 9000)
- Some conceptual differences from TCP:
 - **Multiple streams per connection:** to avoid head-of-line blocking (lecture 1)

QUIC: Quick UDP Internet Connections

- Google, 2012 → IETF standard, 2021 (RFC 9000)
- Some conceptual differences from TCP:
 - **Multiple streams per connection:** to avoid head-of-line blocking (lecture 1)
 - **Fast and secure connection establishment:**
 - TCP + a secure protocol (TLS) on top takes 2 or 3 RTTs to establish a connection.
 - QUIC has encryption built in: a secure connection in 1 RTT, and potentially 0 RTTs when reconnecting

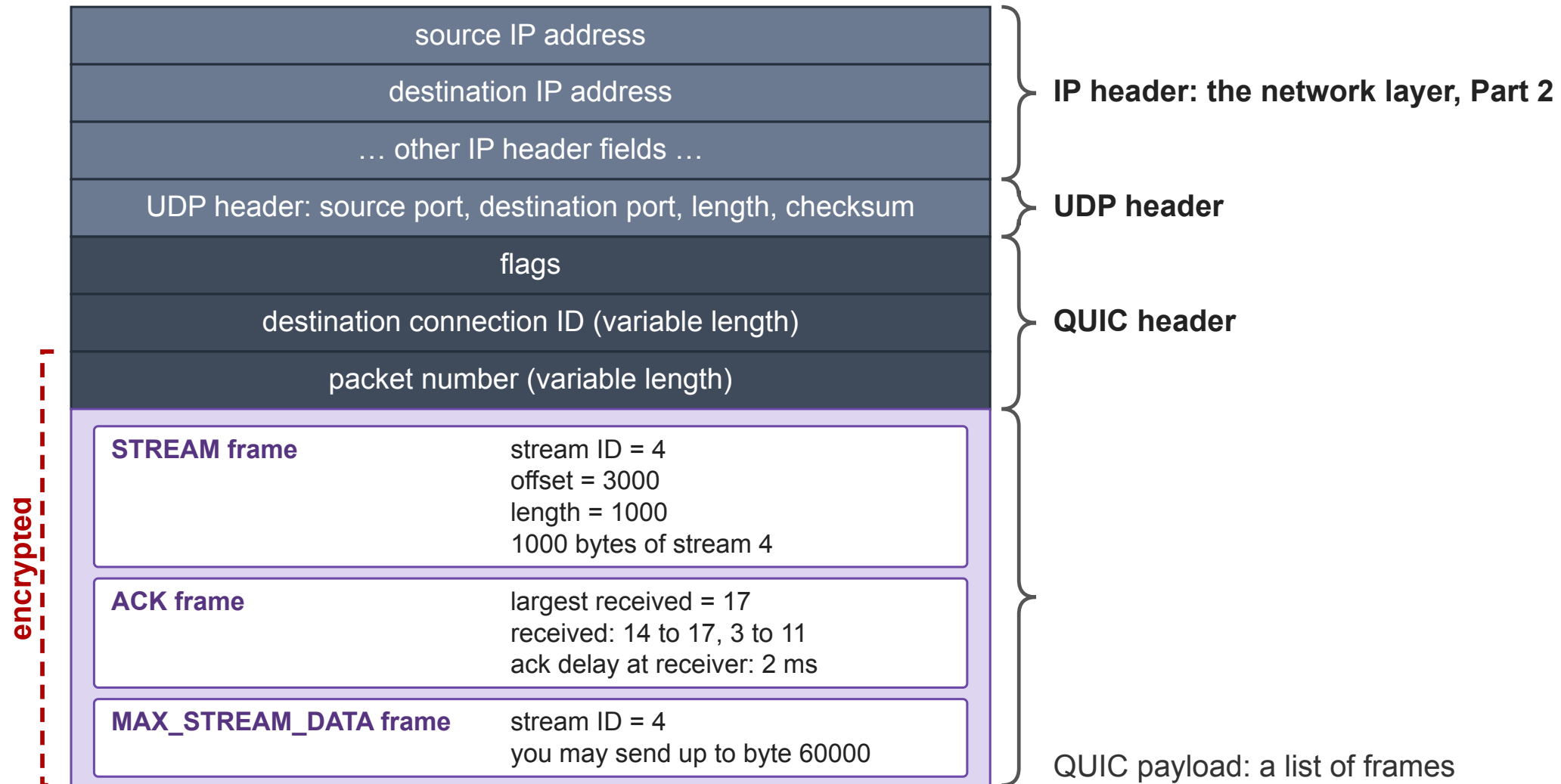
QUIC: Quick UDP Internet Connections

- Google, 2012 → IETF standard, 2021 (RFC 9000)
- Some conceptual differences from TCP:
 - **Multiple streams per connection:** to avoid head-of-line blocking (lecture 1)
 - **Fast and secure connection establishment:**
 - TCP + a secure protocol (TLS) on top takes 2 or 3 RTTs to establish a connection.
 - QUIC has encryption built in: a secure connection in 1 RTT, and potentially 0 RTTs when reconnecting
 - **Connection ID decoupled from IP addresses:** The connection can keep going if a device's IP address changes (e.g., due to mobility)

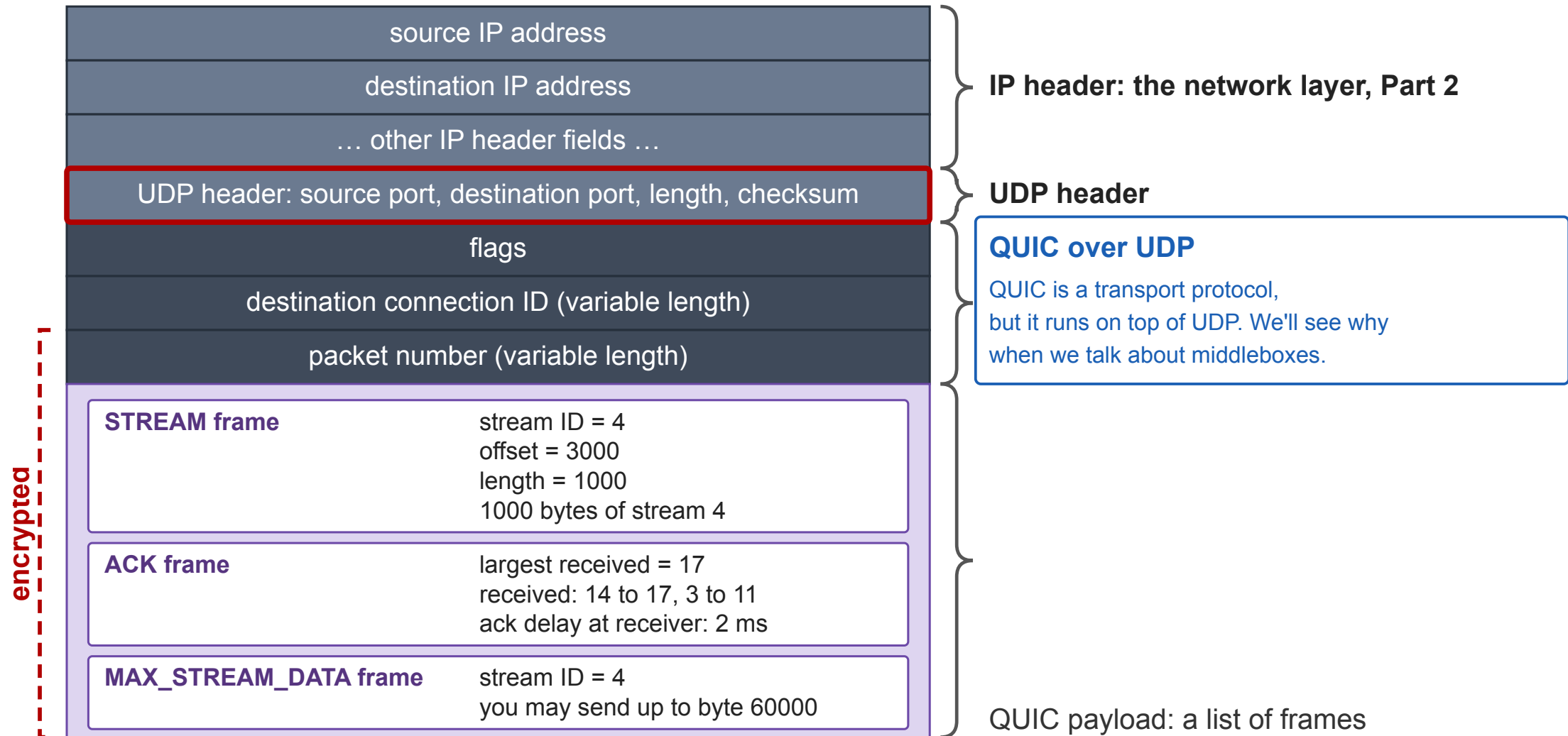
QUIC: Quick UDP Internet Connections

- Google, 2012 → IETF standard, 2021 (RFC 9000)
- Some conceptual differences from TCP:
 - **Multiple streams per connection:** to avoid head-of-line blocking (lecture 1)
 - **Fast and secure connection establishment:**
 - TCP + a secure protocol (TLS) on top takes 2 or 3 RTTs to establish a connection.
 - QUIC has encryption built in: a secure connection in 1 RTT, and potentially 0 RTTs when reconnecting
 - **Connection ID decoupled from IP addresses:** The connection can keep going if a device's IP address changes (e.g., due to mobility)
 - **Decoupling packet IDs from byte offsets:** Packets have unique, increasing IDs. Even a retransmitted packet has its own new unique ID. No ambiguity in matching acks to packets.

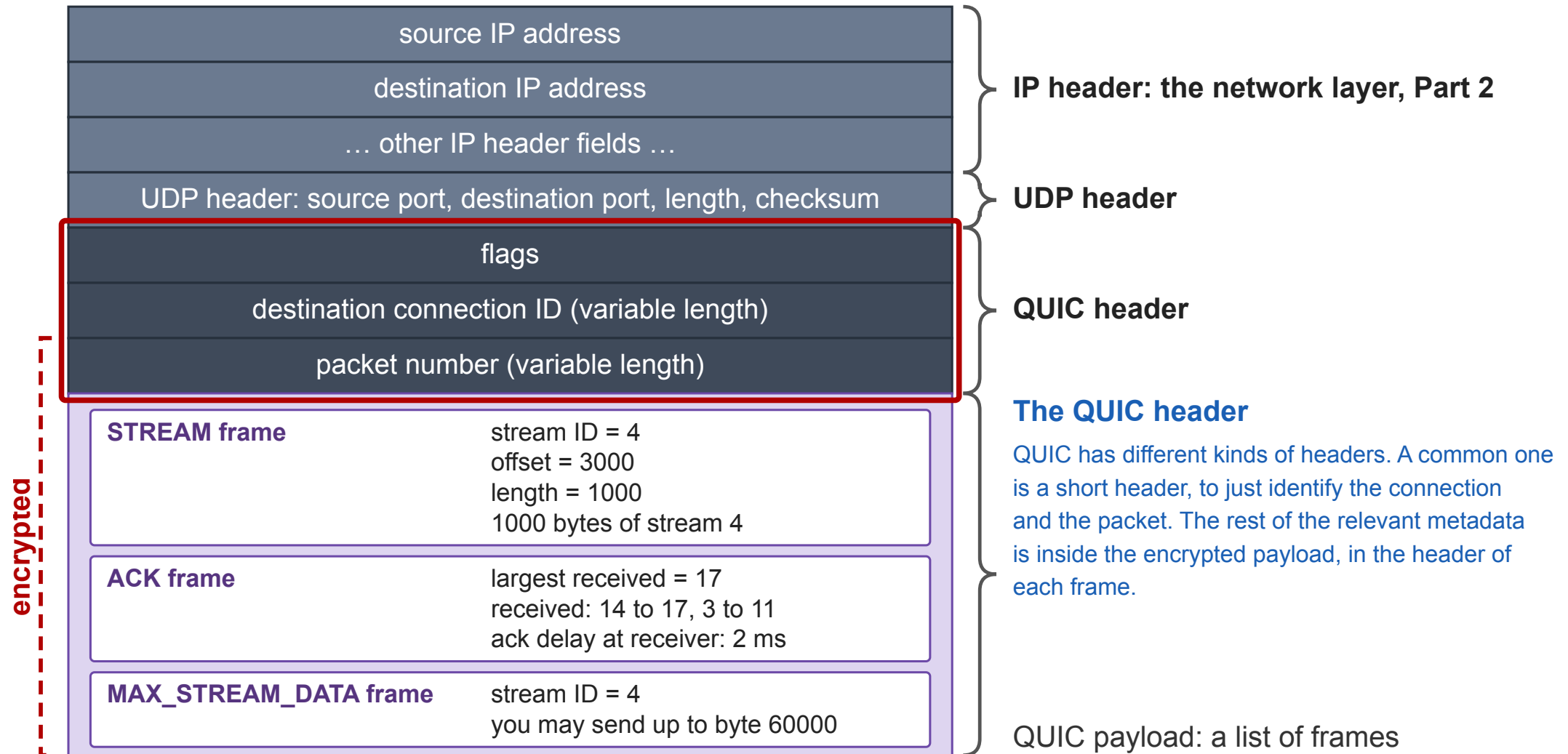
QUIC packet (i.e., metadata)



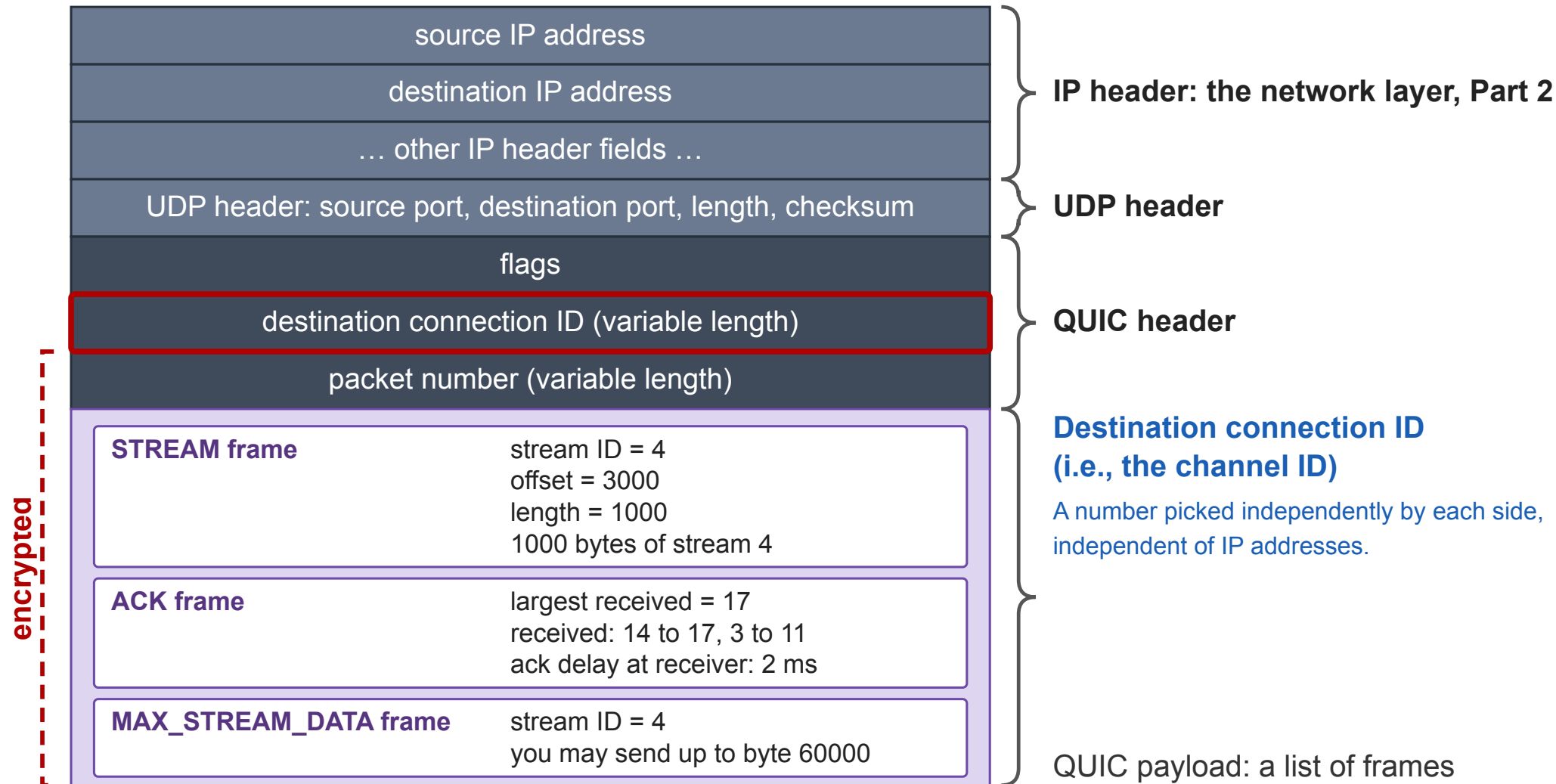
QUIC packet (i.e., metadata)



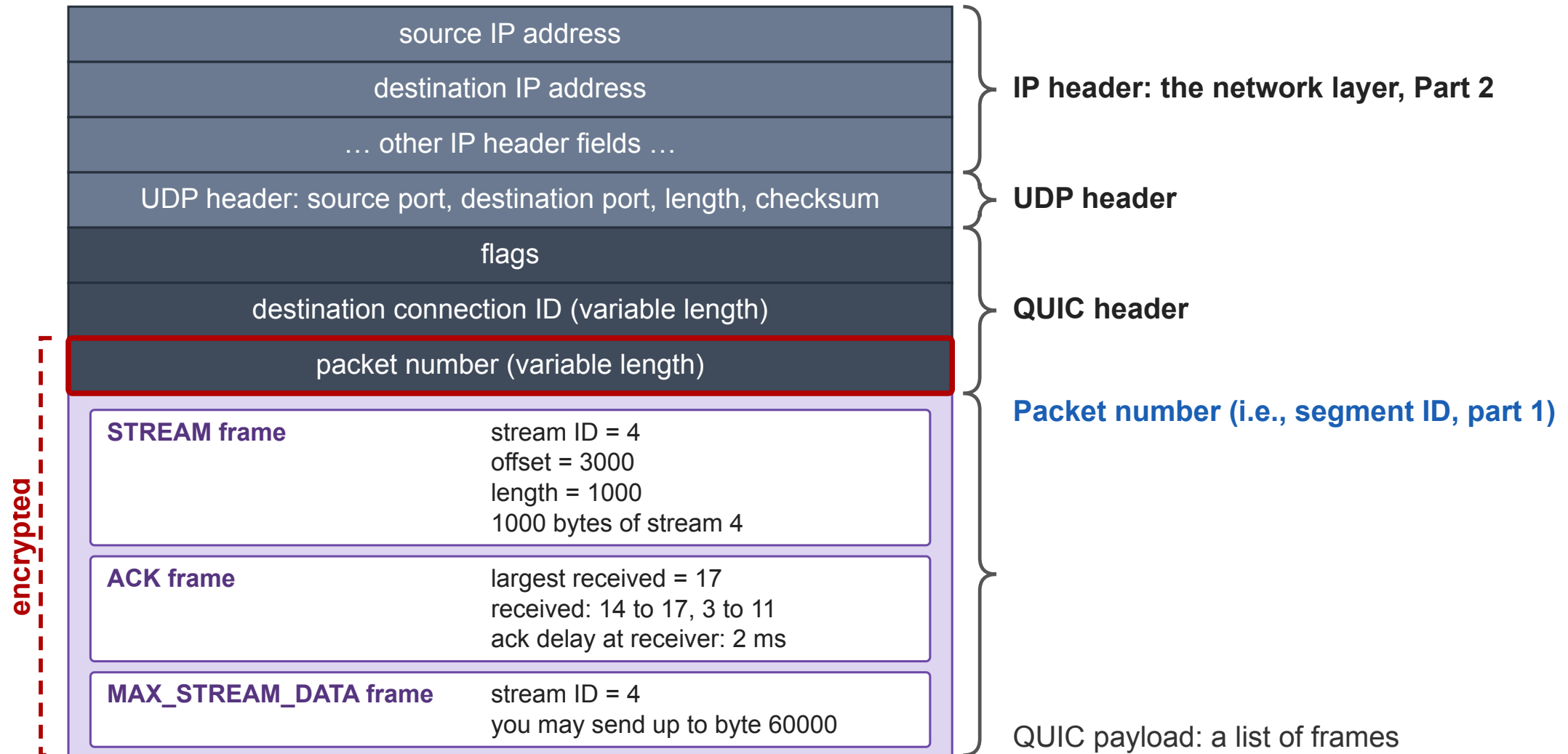
QUIC packet (i.e., metadata)



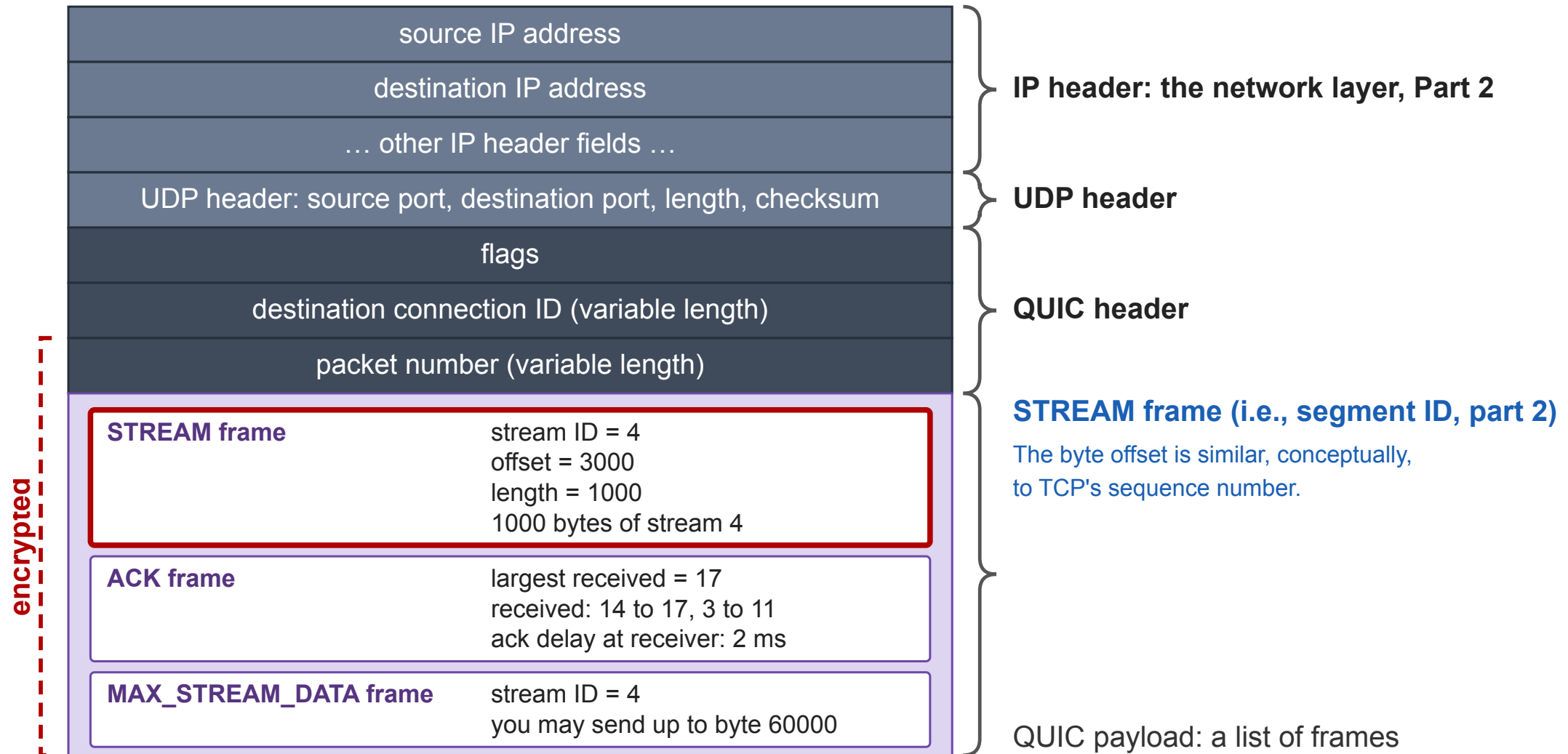
QUIC packet (i.e., metadata)



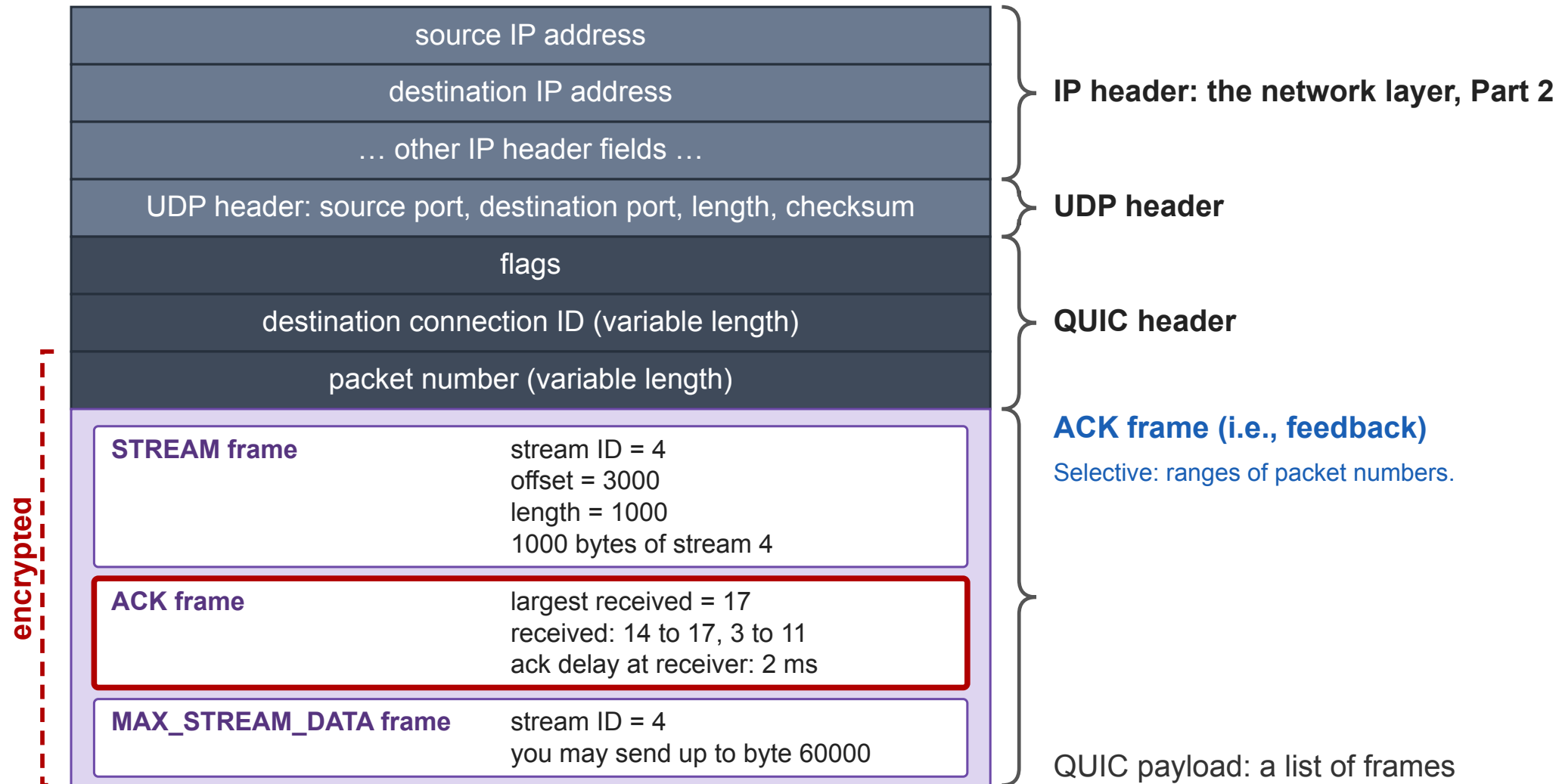
QUIC packet (i.e., metadata)



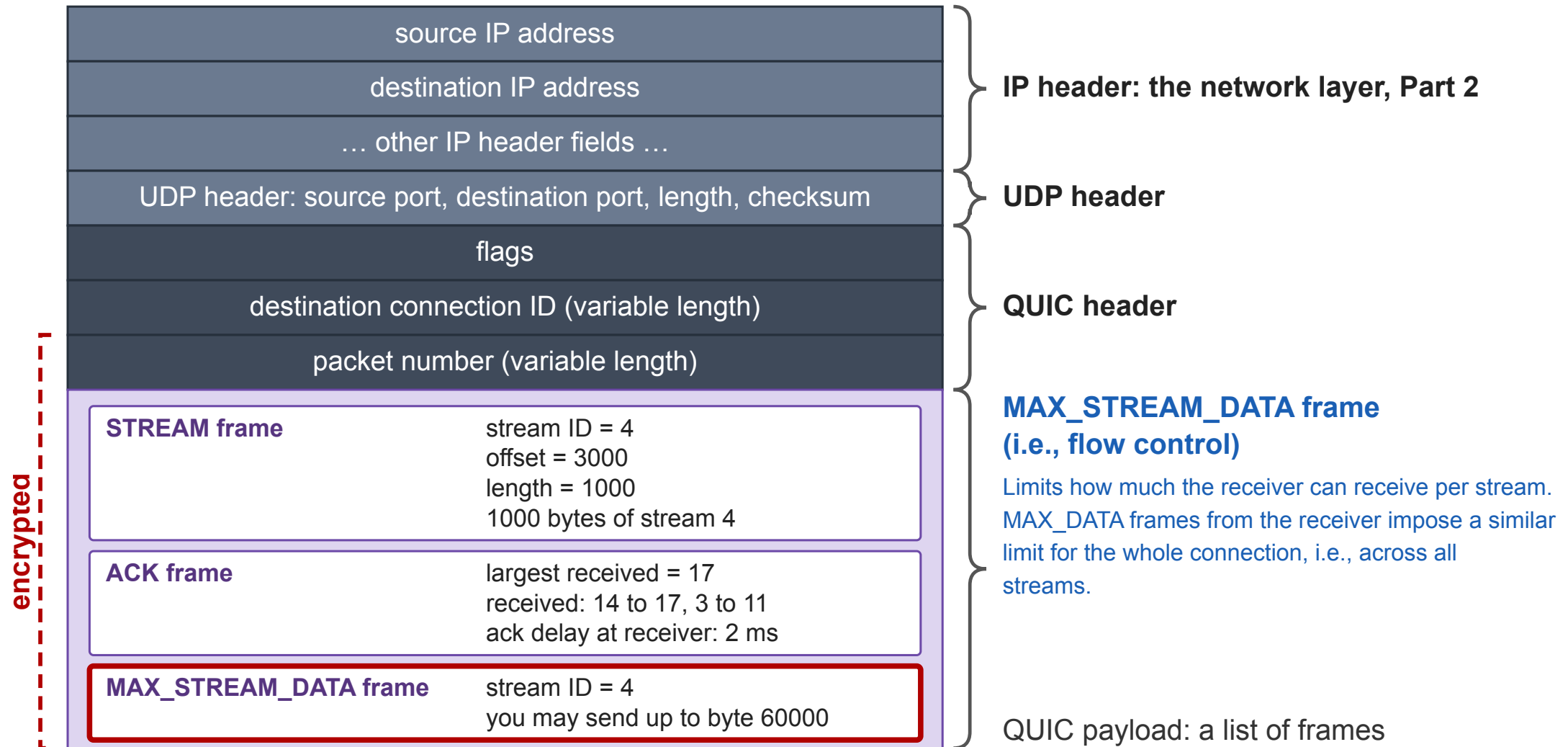
QUIC packet (i.e., metadata)



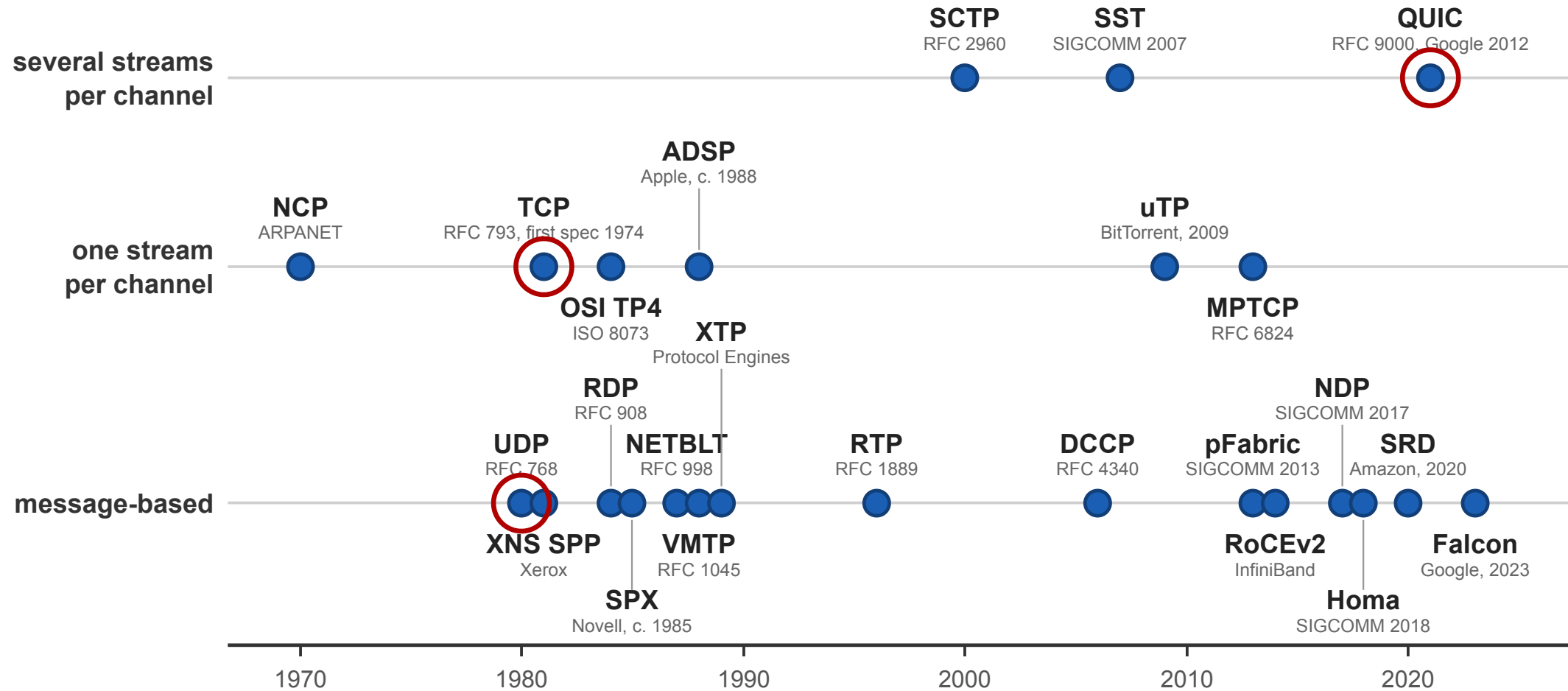
QUIC packet (i.e., metadata)



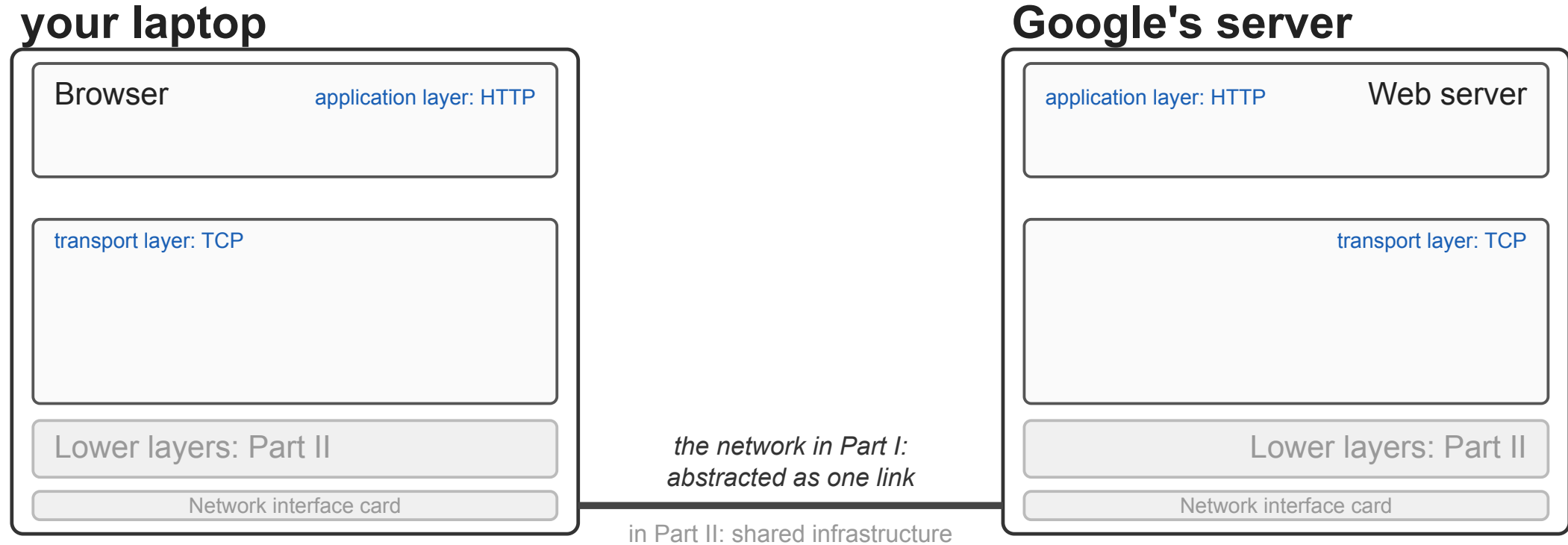
QUIC packet (i.e., metadata)



Transport protocols over the years

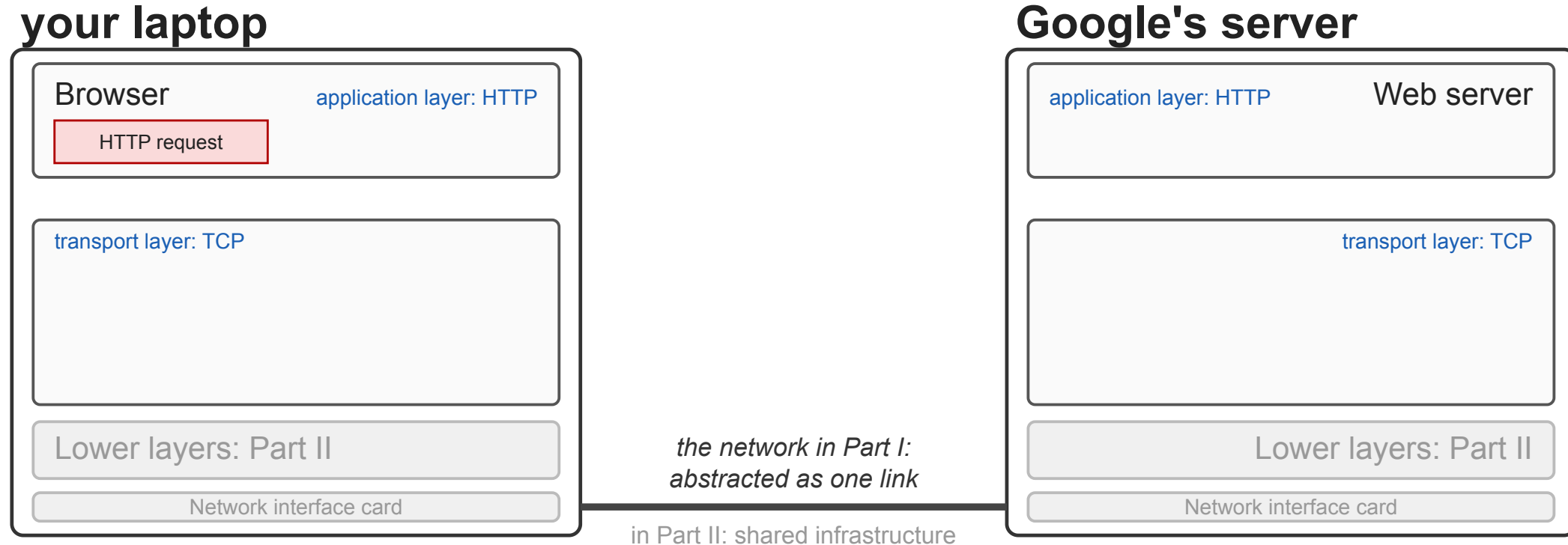


Part I, all together: typing www.google.com



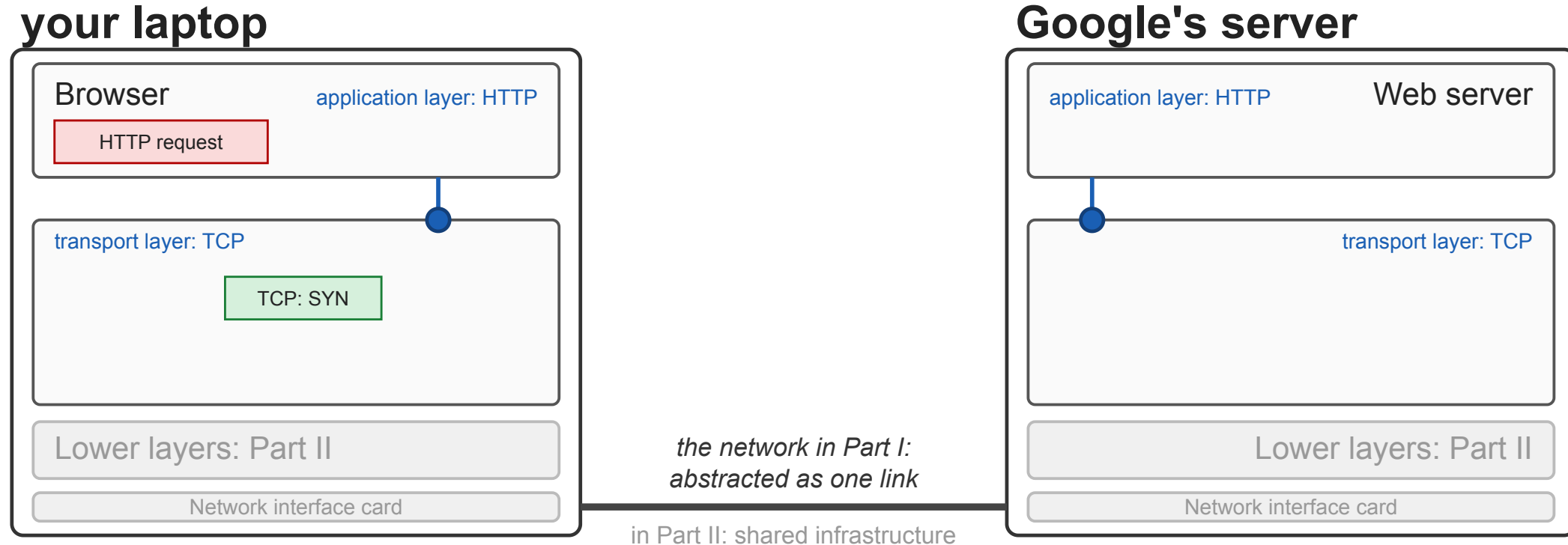
You come to school, open your laptop, and type www.google.com in the browser.

Part I, all together: typing www.google.com



The application, the browser, wants a page from the application on the other side, the web server. It writes a request in the application protocol, HTTP: GET /index.html, Host: www.google.com.

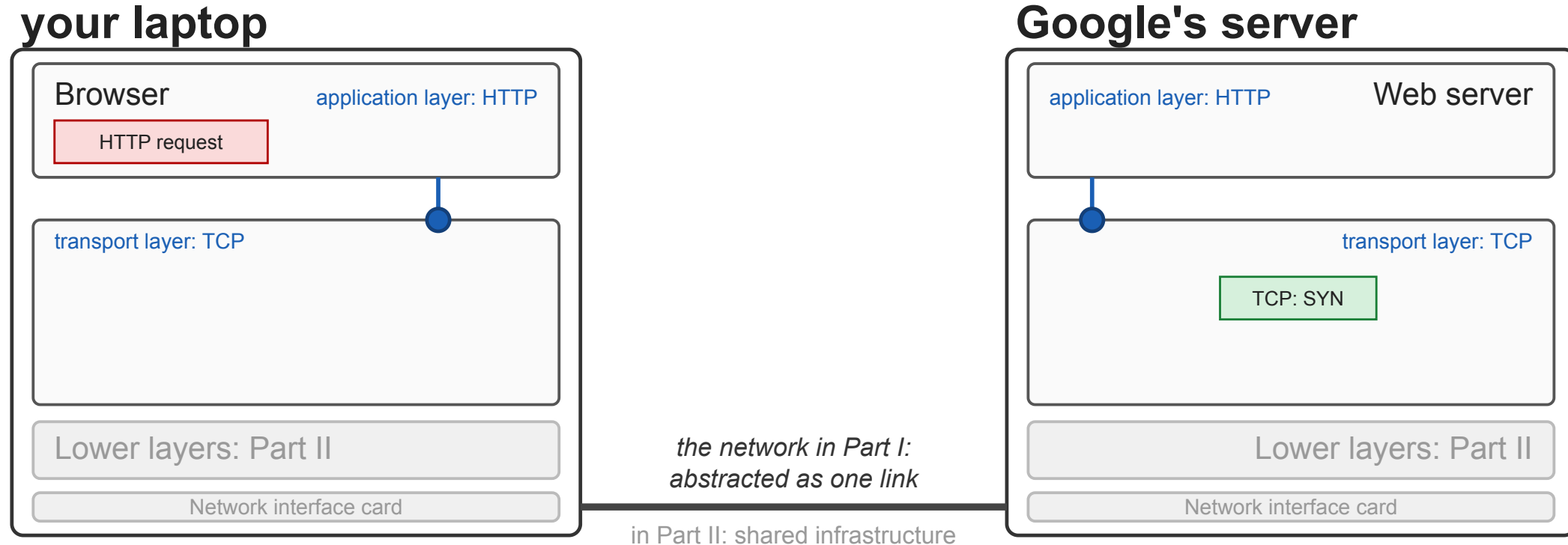
Part I, all together: typing www.google.com



It asks the transport layer for a channel to the server's application: `socket()`, `connect()`.

The transport protocol, TCP, starts the handshake: a SYN segment with its first sequence number.

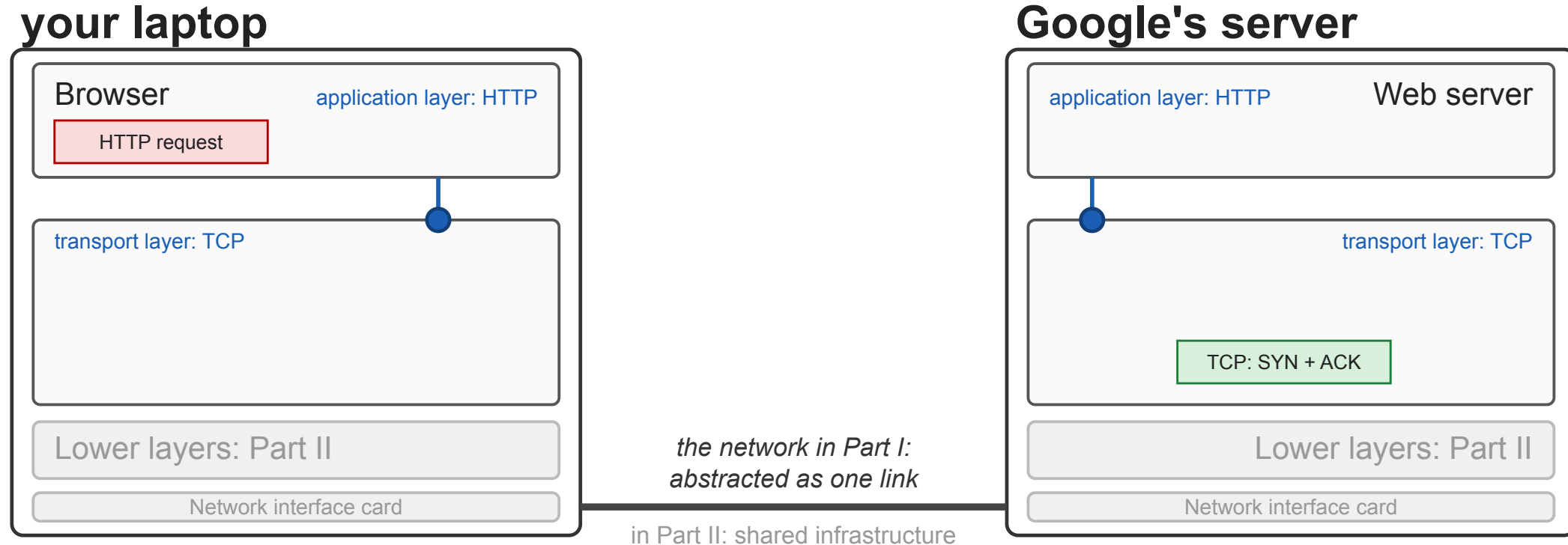
Part I, all together: typing www.google.com



It asks the transport layer for a channel to the server's application: `socket()`, `connect()`.

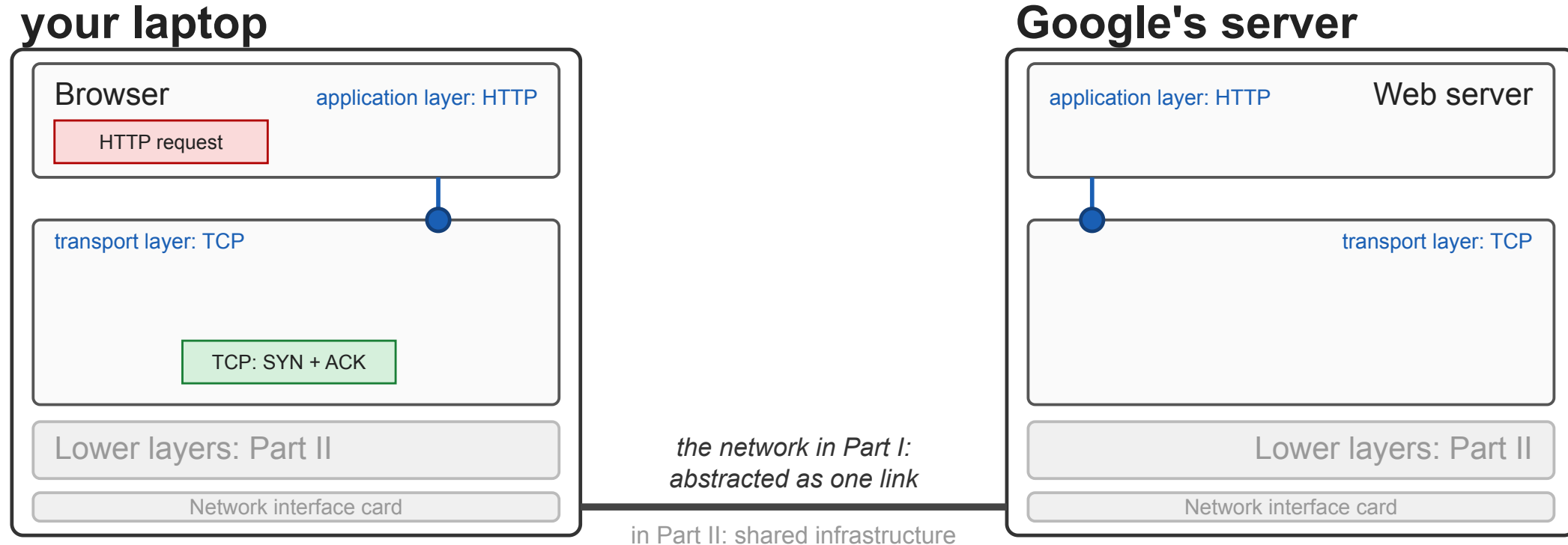
The transport protocol, TCP, starts the handshake: a SYN segment with its first sequence number.

Part I, all together: typing www.google.com



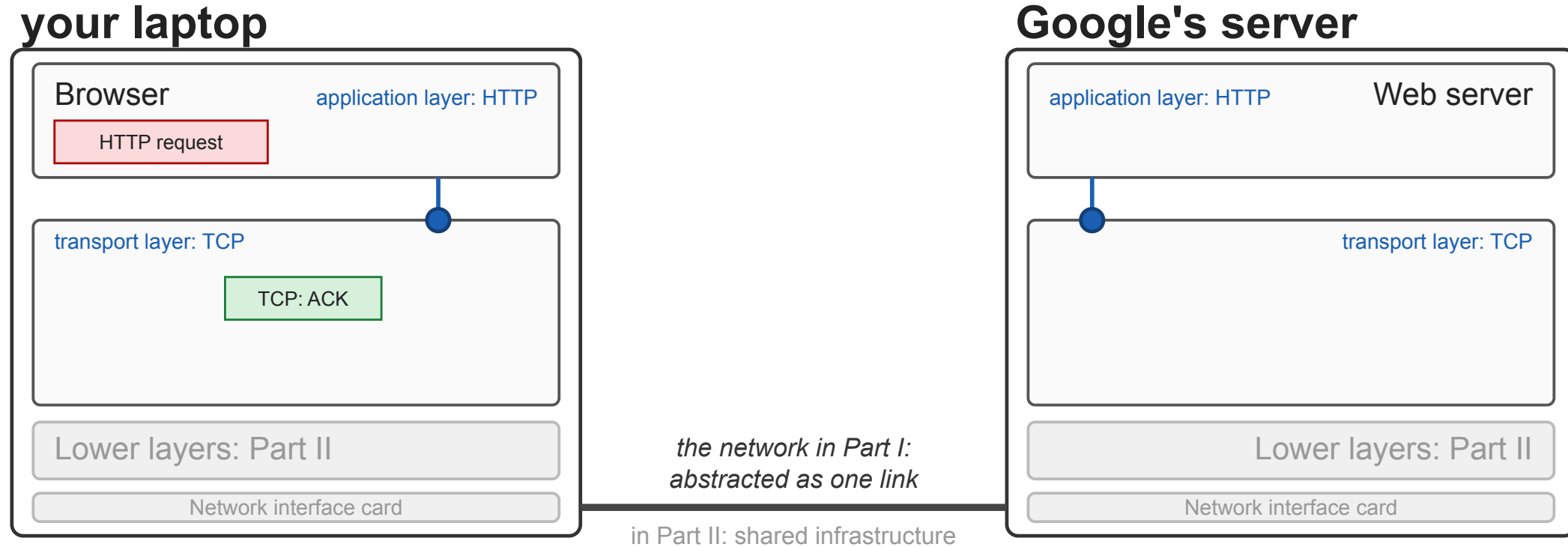
The server's transport layer, listening on port 80 (socket(), bind(), listen(), accept()), answers: SYN + ACK, with its own first sequence number, window and MSS.

Part I, all together: typing www.google.com



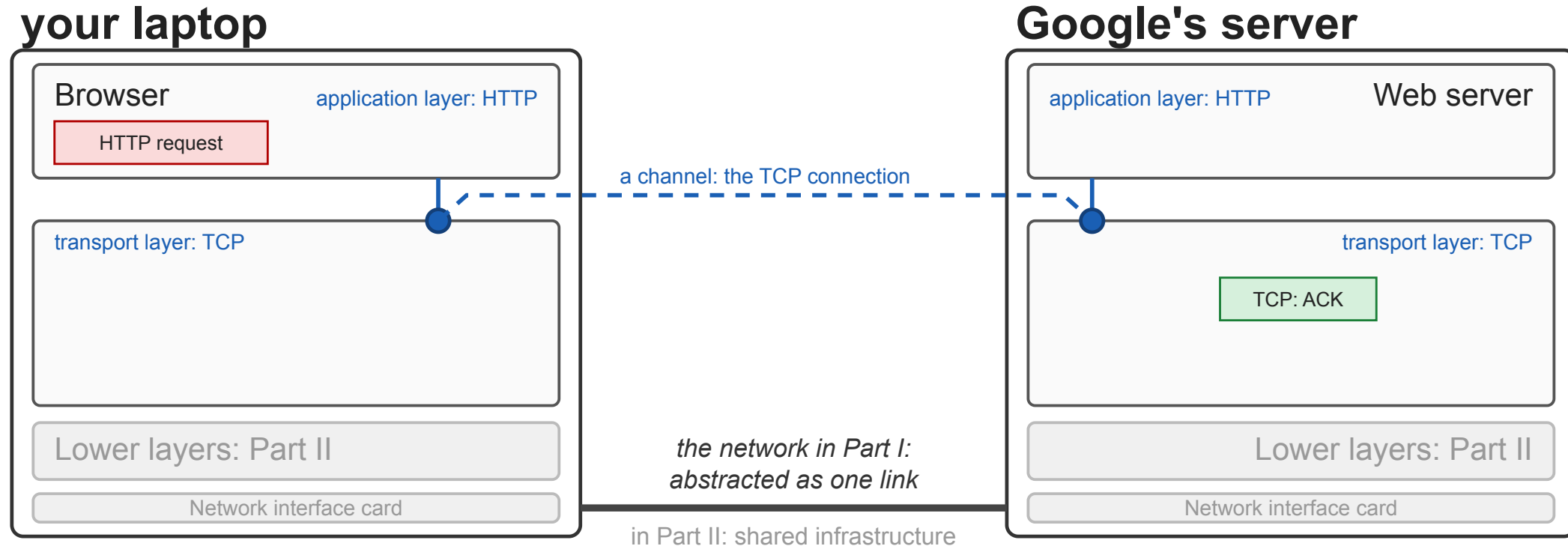
The server's transport layer, listening on port 80 (socket(), bind(), listen(), accept()), answers: SYN + ACK, with its own first sequence number, window and MSS.

Part I, all together: typing www.google.com



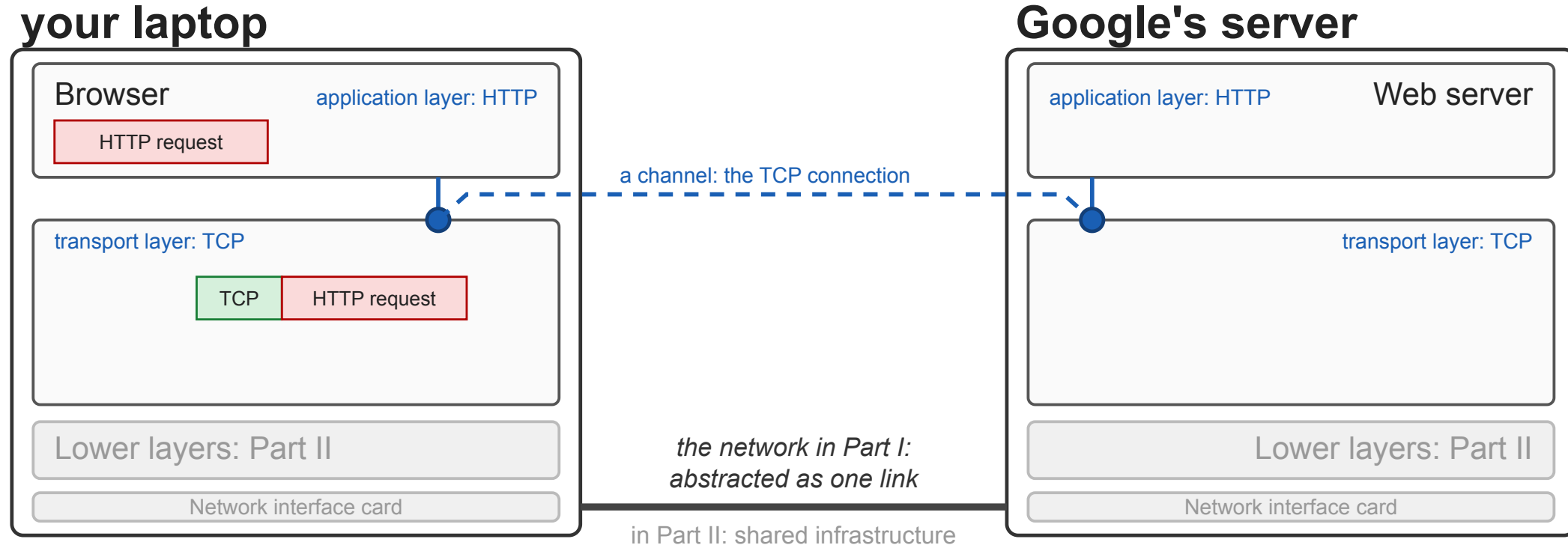
The laptop's transport layer answers with an ACK.

Part I, all together: typing www.google.com



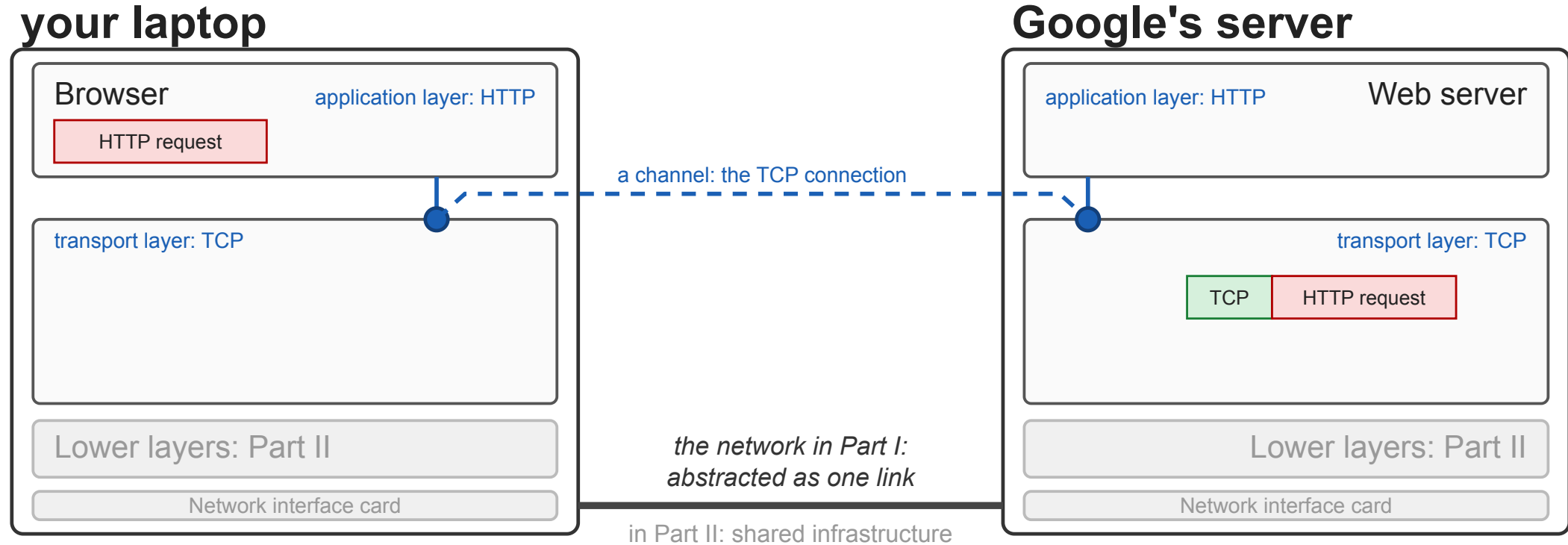
The handshake is complete: the connection (channel) is established and can be used for data transfer.

Part I, all together: typing www.google.com



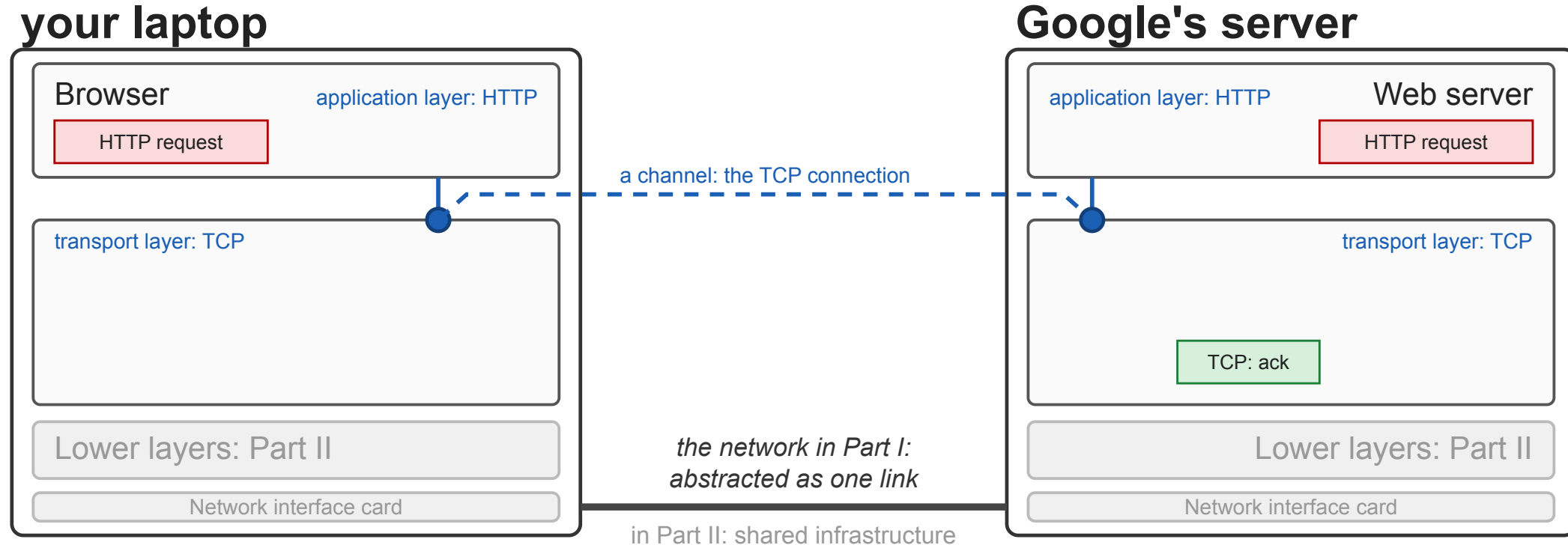
The browser writes its request into the channel: `send()`. TCP segments the request, adding its own metadata, and ensures it is transferred reliably to the server (the request fits in one segment in this example).

Part I, all together: typing www.google.com



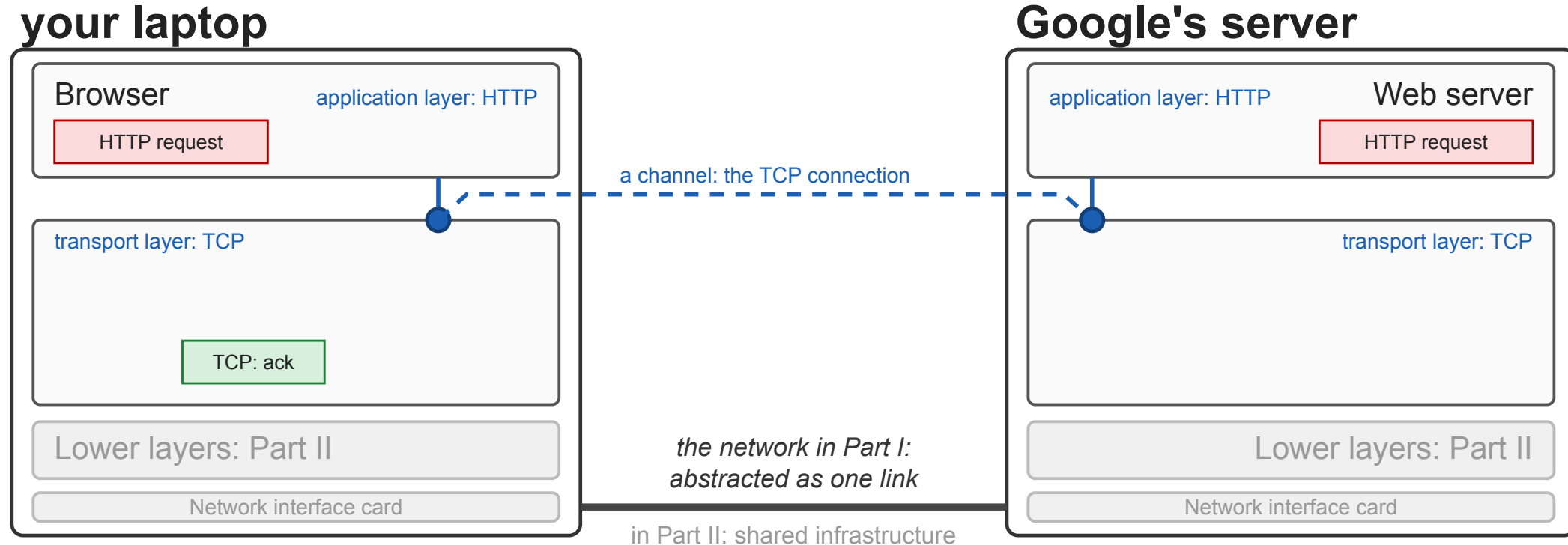
The browser writes its request into the channel: `send()`. TCP segments the request, adding its own metadata, and ensures it is transferred reliably to the server (the request fits in one segment in this example).

Part I, all together: typing www.google.com



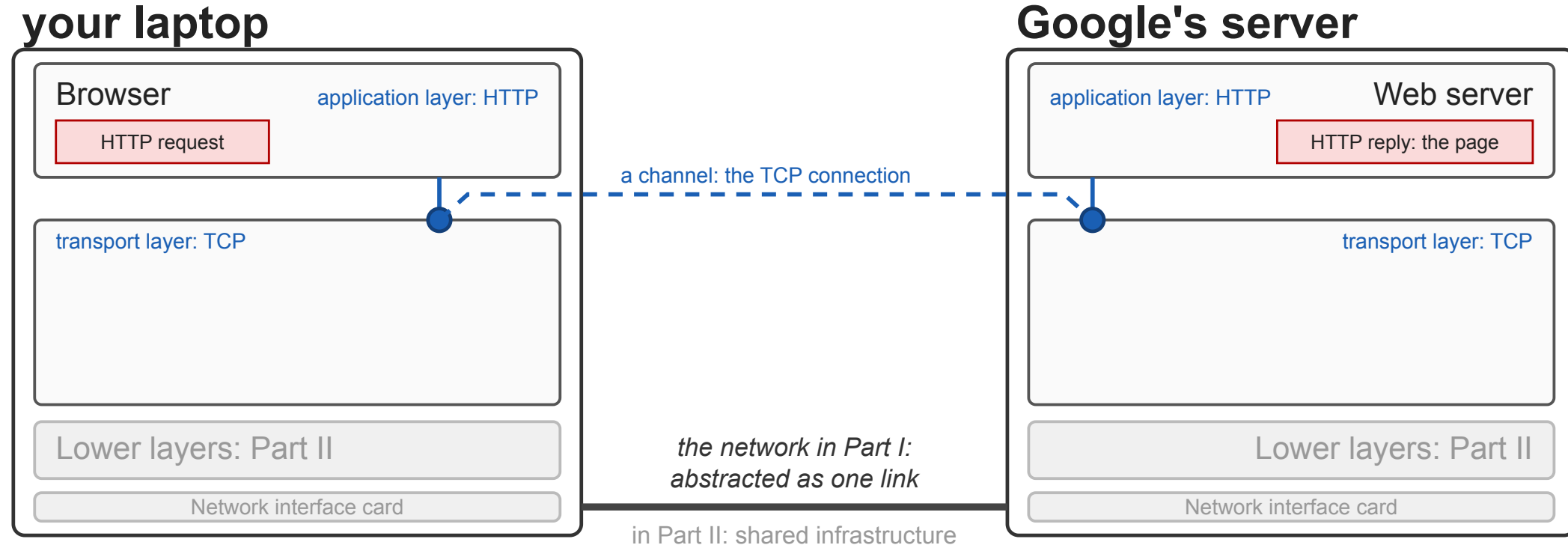
The server's transport layer checks the segment, hands the bytes up (recv() returns, and the web server reads the request), and acknowledges it.

Part I, all together: typing www.google.com



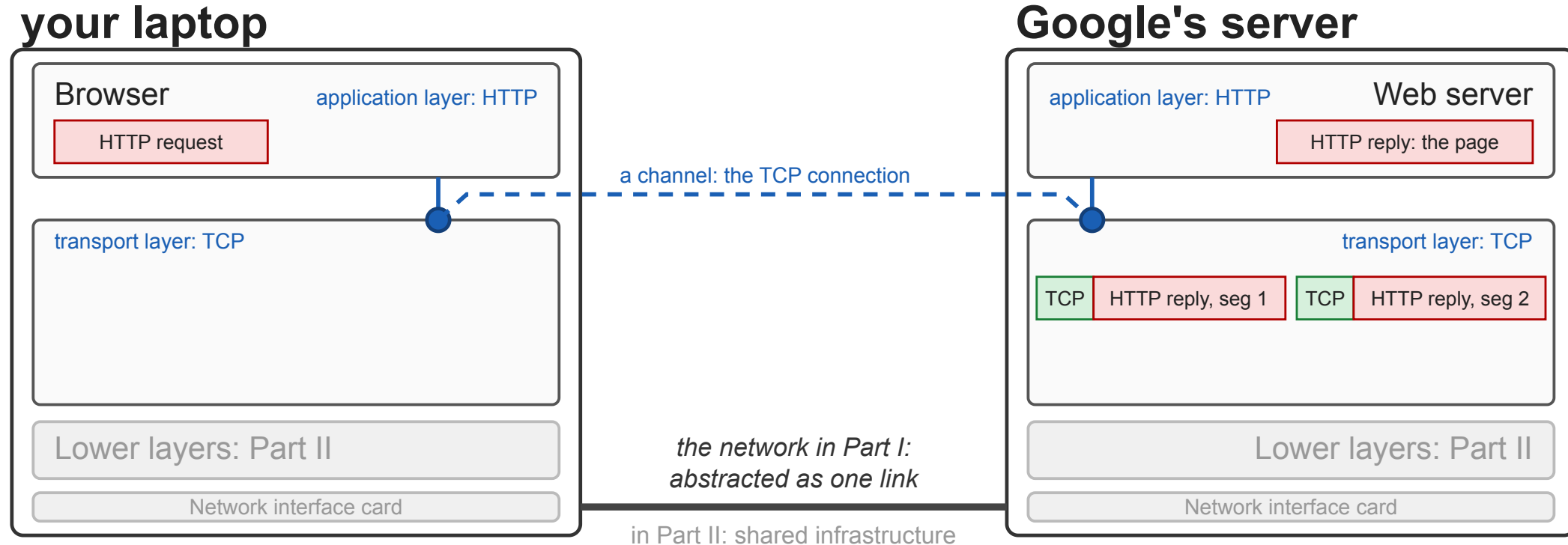
The server's transport layer checks the segment, hands the bytes up (recv() returns, and the web server reads the request), and acknowledges it.

Part I, all together: typing www.google.com



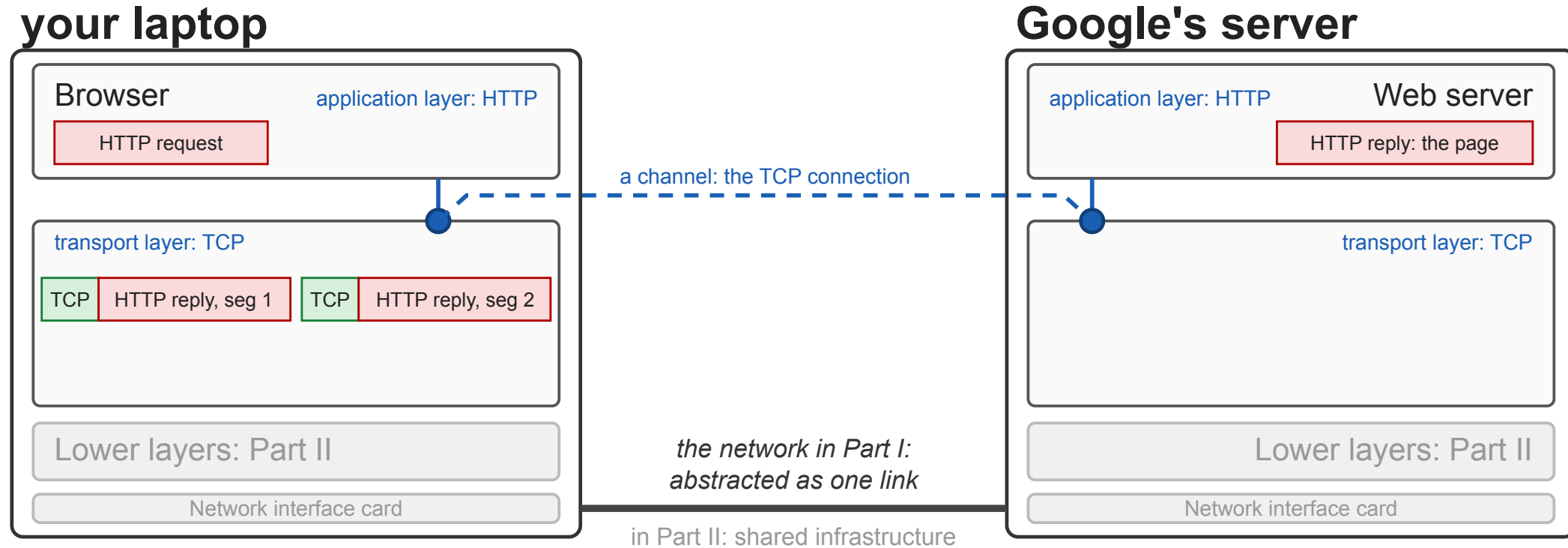
The web server writes the reply in HTTP, 200 OK and the page, and writes it into the channel: send().

Part I, all together: typing www.google.com



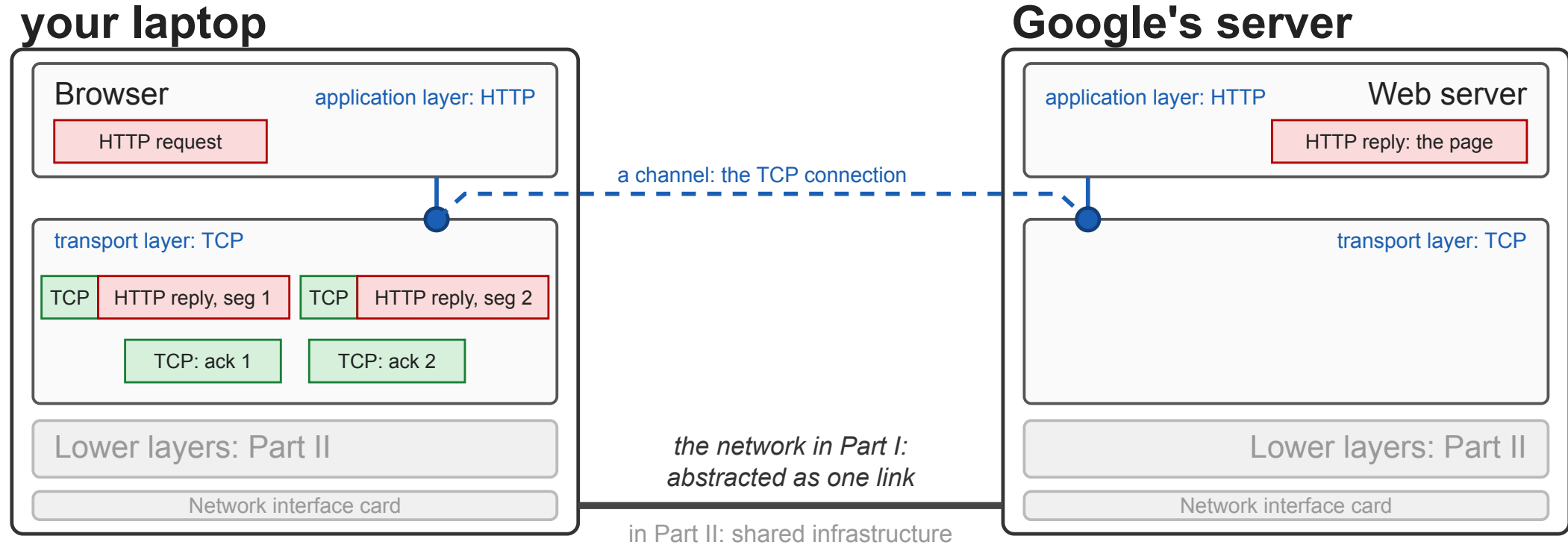
Its transport layer cuts the bytes into segments of at most one MSS and sends them out.

Part I, all together: typing www.google.com



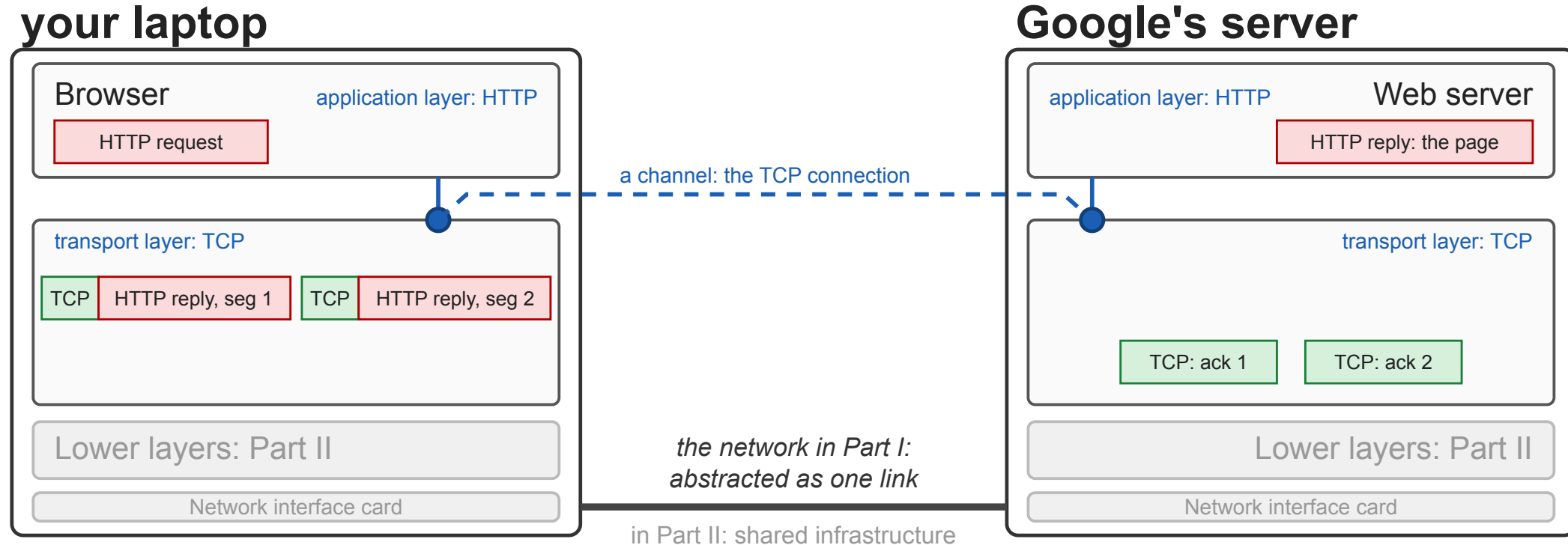
Its transport layer cuts the bytes into segments of at most one MSS and sends them out.

Part I, all together: typing www.google.com



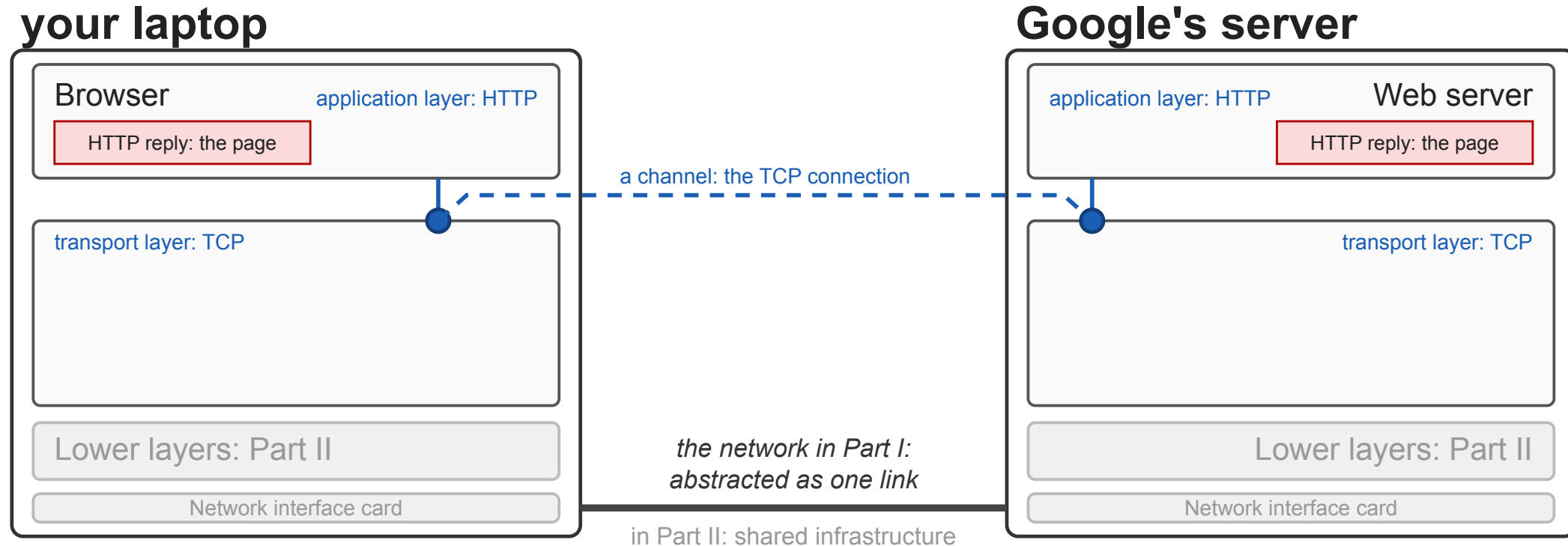
The laptop's transport layer (TCP) checks each segment, puts them in order, and acknowledges each one, also with its receive window. A lost segment is sent again when the timer goes off, or after three duplicate acks.

Part I, all together: typing www.google.com



With the acks, the server's transport layer slides its window; here, there is nothing more to send.

Part I, all together: typing www.google.com



recv() returns the bytes in order: the browser renders the page. When both sides are done, each closes its direction of the channel.

Part I, all together: typing www.google.com

