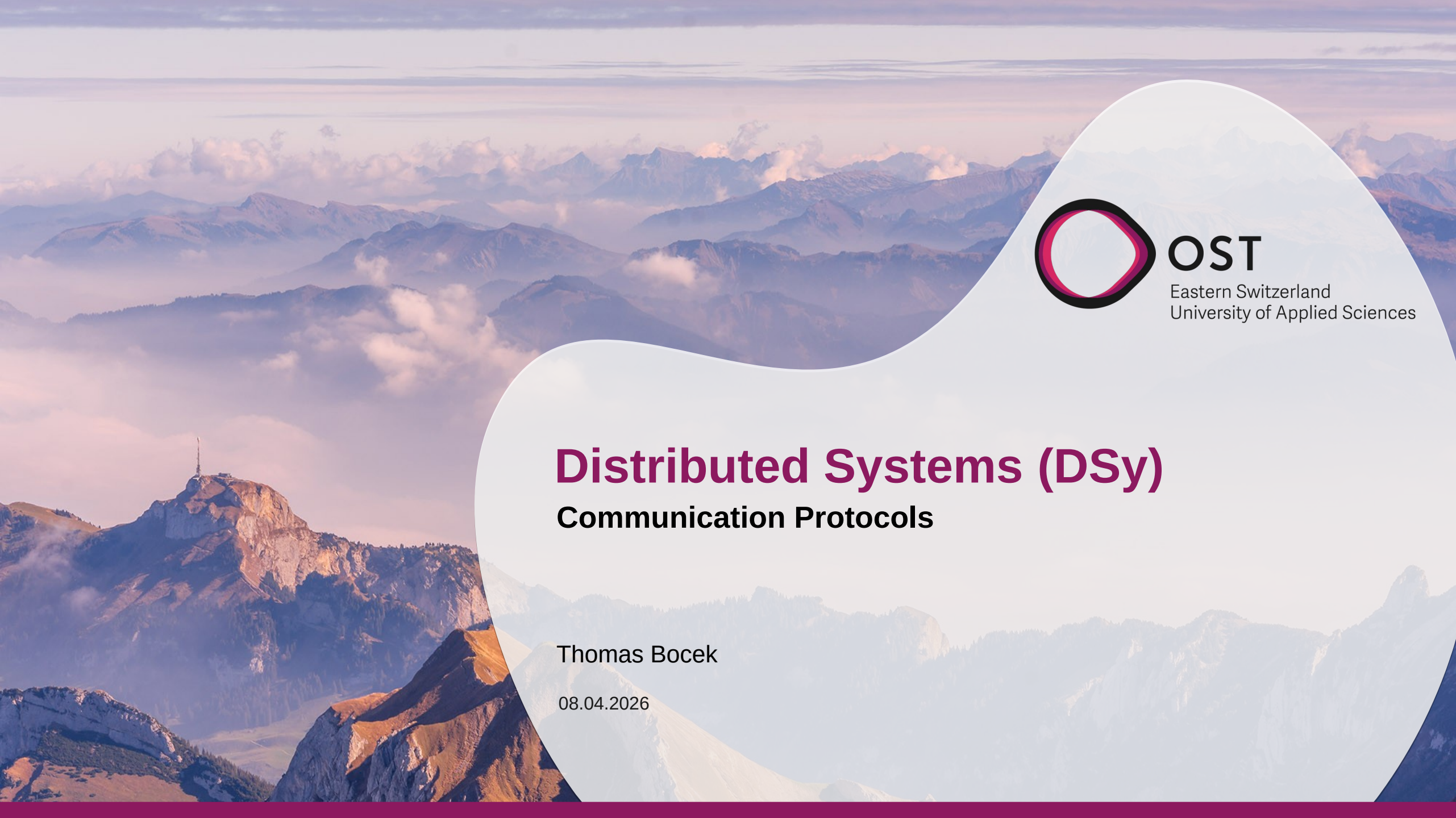


OST
Eastern Switzerland
University of Applied Sciences

Distributed Systems (DSy)

Communication Protocols

Thomas Bocek

08.04.2026

Learning Goals

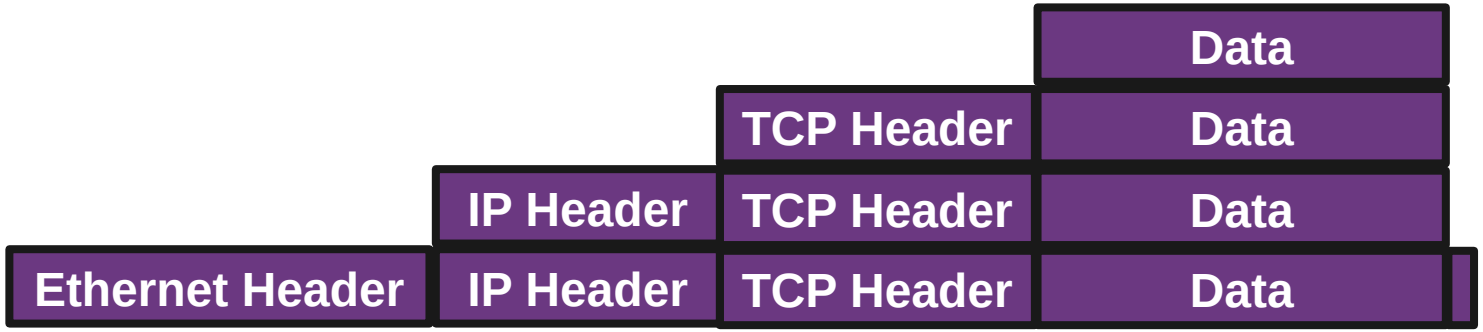
- Lecture 8 (Protocols)
  - How do network layers work?
  - What are the TCP mechanisms?
  - What are the problems of TCP, and how other protocols (HTTP/3) can improve that

Networking: Layers

- Networking: Each vendor had its own proprietary solution - not compatible with another solution
  - IPX/SPX – 1983, AppleTalk 1985, DECnet 1975, XNS 1977
- Nowadays most vendors build compatible networks hardware/software
  - Cisco, Dell, HP, Huawei, Juniper, Lenovo, Netgear, MicroTik, Ubiquiti, etc.

- Goal of layers: interoperability
  - 1984: ISO 7498 - The Basic Reference Model for Open Systems Interconnection

OSI model	"Internet model"
Application	Application
Presentation	
Session	
Transport	Transport
Network	Internet
Data link	Link
Pysical	



OST

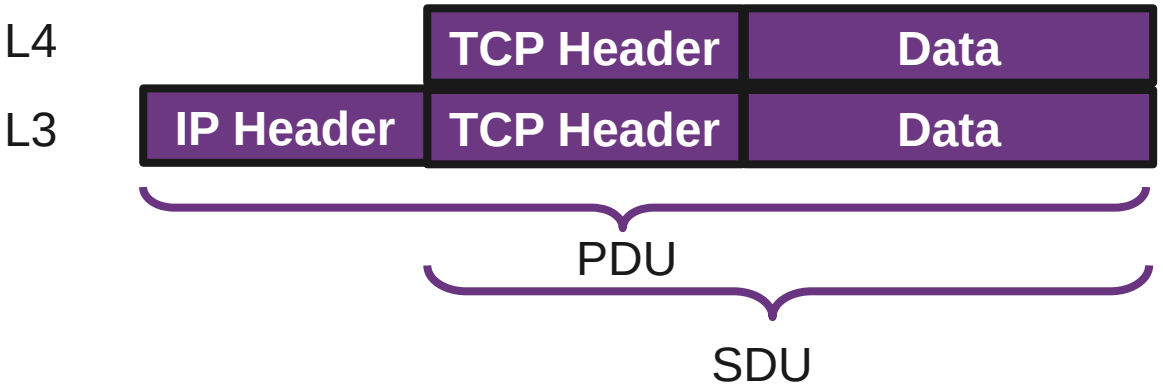
Networking: Definitions

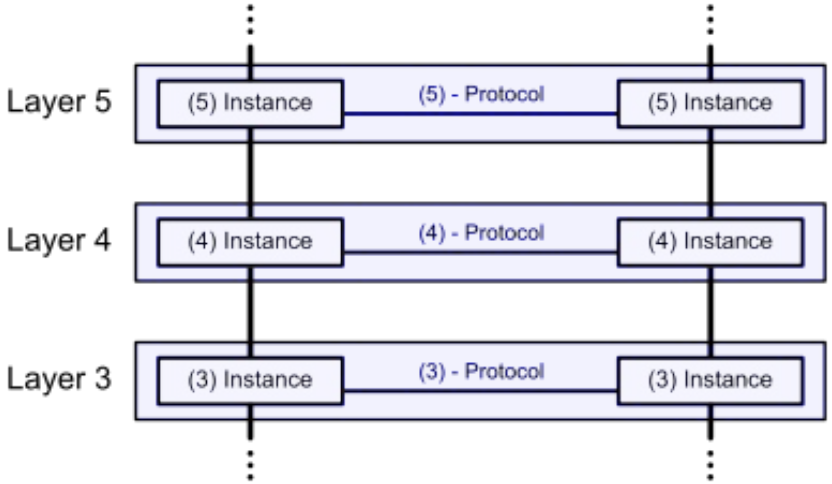
RFC 1122, Internet STD 3 (1989)	OSI model
Four layers	Seven layers
"Internet model"	OSI model
Application	Application
	Presentation
	Session
Transport	Transport
Internet	Network
Link	Data link
	Physical

source: https://en.wikipedia.org/wiki/Internet_protocol_suite

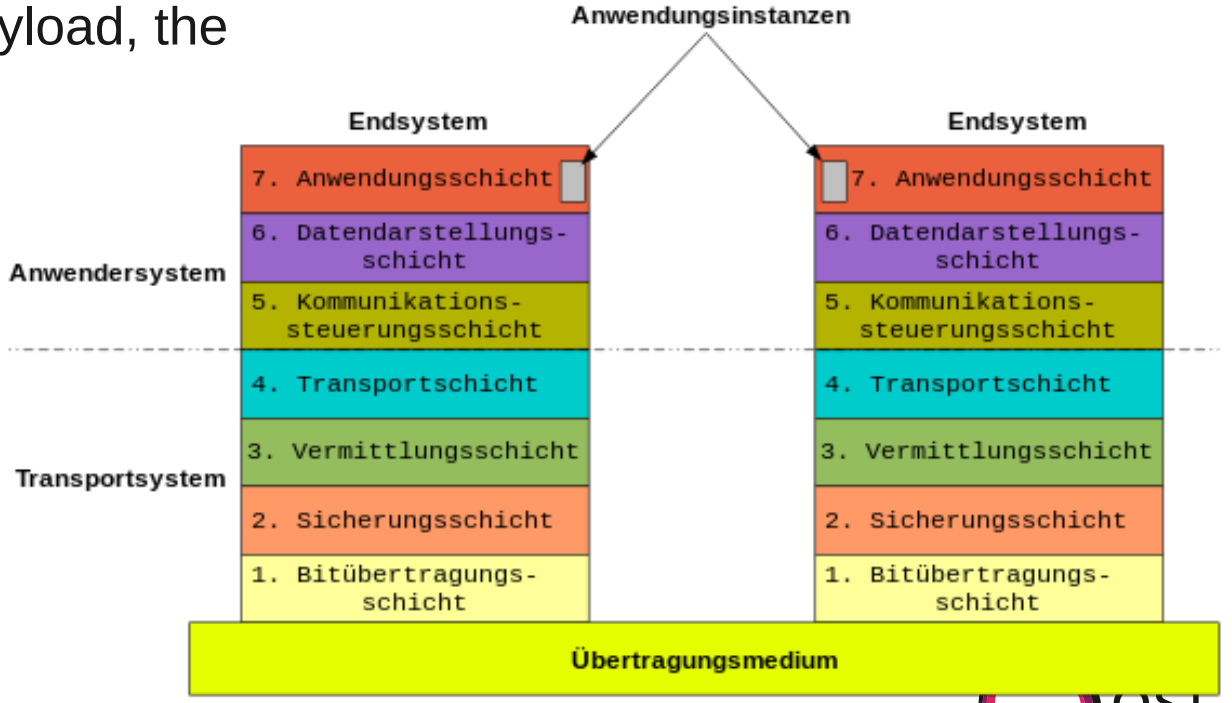
Layer Abstraction

- Protocols enable an entity/instance to interact with an entity/instance at the same layer in another host
- Service definitions: provide functionality to an (N)-layer by an (N-1) layer
- Each PDU contains a protocol header and payload, the service data unit (SDU). E.g. PDU of L3:





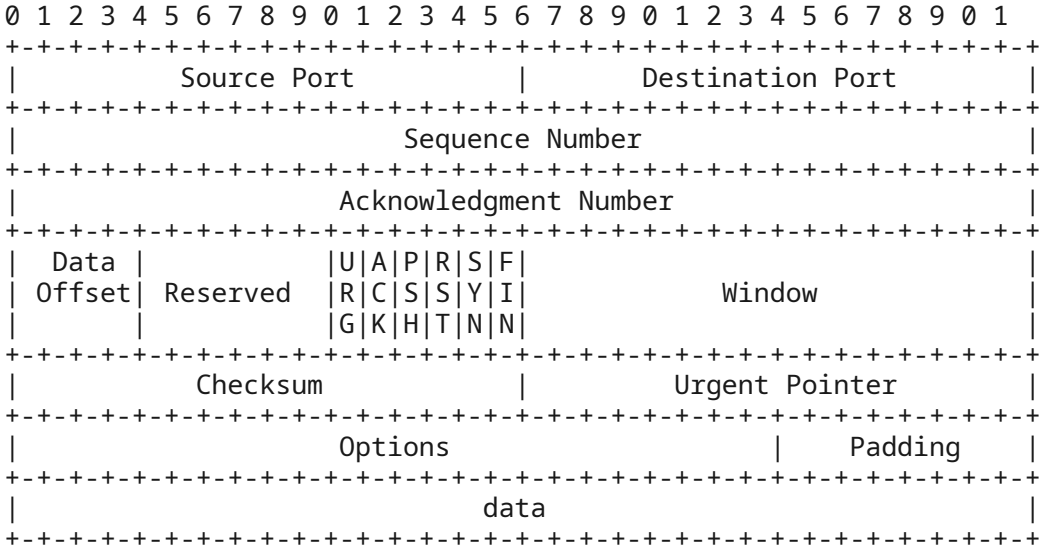
source: https://en.wikipedia.org/wiki/OSI_model



source: https://de.wikipedia.org/wiki/OSI-Modell

Layer 4 - Transport

- TCP (Transmission Control Protocol)
  - Reliable (retransmission)
  - Ordered
  - Window – capacity of receiver
  - Checksum – 16bit (crc16)
  - TCP overhead: 20bytes (MTU 1500 bytes)
    - IP overhead: 20bytes
    - Ethernet frame: 18bytes (crc32)
- TCP tries to correct errors; you don’t need to worry…
  - Sometimes, you need to worry…

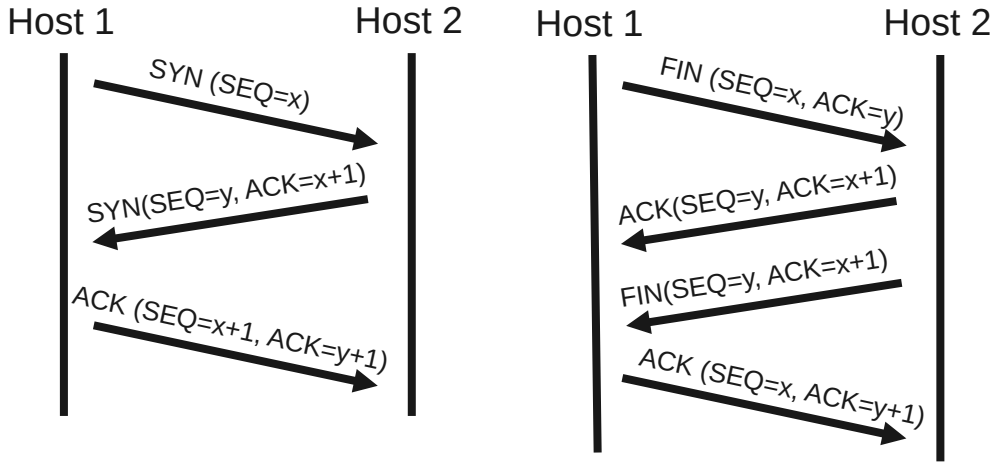


source: https://datatracker.ietf.org/doc/html/rfc9293

OST

Layer 4 - TCP

- Connection establishment
  - SYN, SYN-ACK, ACK (three way)
  - Initiates TCP session: initial sequence number is ~random
- Connection termination
  - FIN, ACK + FIN, ACK (three/four way)
  - 3-way handshake, when host 1 sends a FIN and host 2 replies with a FIN & ACK
- Sequences and ACKs
  - Identification each byte of data
  - Order of the bytes → reconstruction
  - Detecting lost data: RTO, DupACK:

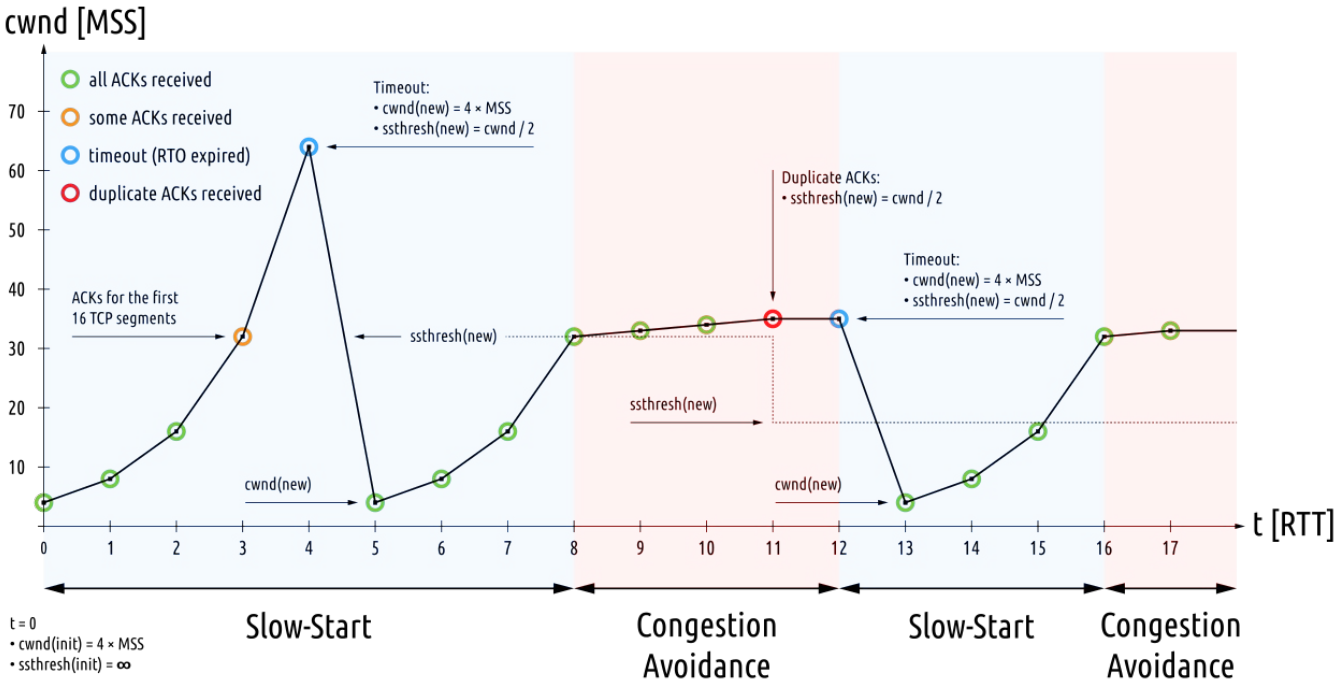


- Retransmission timeout
  - If no ACK is received aftert timout (e.g. 2xRTT), resend.
- Duplicate cumulative acknowledgements, selective ACK [link]
  - ACKs for last consecutive packets
  - 3 times same ACK → retransmit missing packets (fast retransmit)

OST

Layer 4 - TCP

- Flow control
  - Sender is not overwhelming a receiver
  - Back pressure
  - Sliding window:
    - Receiver specifies the amount of additionally received data in bytes that can be buffered
    - Sender up to that amount of data before ACK
- Congestion control
  - slow-start
  - congestion avoidance
- Difference flow/congestion control



source: https://upload.wikimedia.org/wikipedia/commons/thumb/2/24/TCP_Slow-Start_and_Congestion_Avoidance.svg/1280px-TCP_Slow-Start_and_Congestion_Avoidance.svg.png

OST

TCP/IP from an Application Developer View

- Server in golang (repo)
  - git clone https://github.com/tbocek/DSy [link]
  - go run server.go → server
- Listening on TCP port 8081
  - Return string in uppercase
- Node.js version
- Client: nc localhost 8081

```
const net = require('net');
const server = new net.Server();
server.listen(8081, function() {
   console.log('Launching server...');
});

server.on('connection', function(socket) {
   socket.on('data', function(chunk) {
      console.log(`Data received from client: $
{chunk.toString()}`);

socket.write(chunk.toString().toUpperCase() +
"\n");
   });
});
```

```
package main
import ("bufio"
    "fmt"
    "net"
    "strings")
func main() {
    fmt.Println("Launching server...")
    ln, _ := net.Listen("tcp", ":8081") // listen on all
interfaces
    for {
        conn, _ := ln.Accept() // accept connection on
port
        message, _ :=
bufio.NewReader(conn).ReadString('\n') //read line
        fmt.Print("Message Received:", string(message))
        newMessage := strings.ToUpper(message) //change
to upper
        conn.Write([]byte(newMessage + "\n")) //send
upper string back
    }
}
```


OST

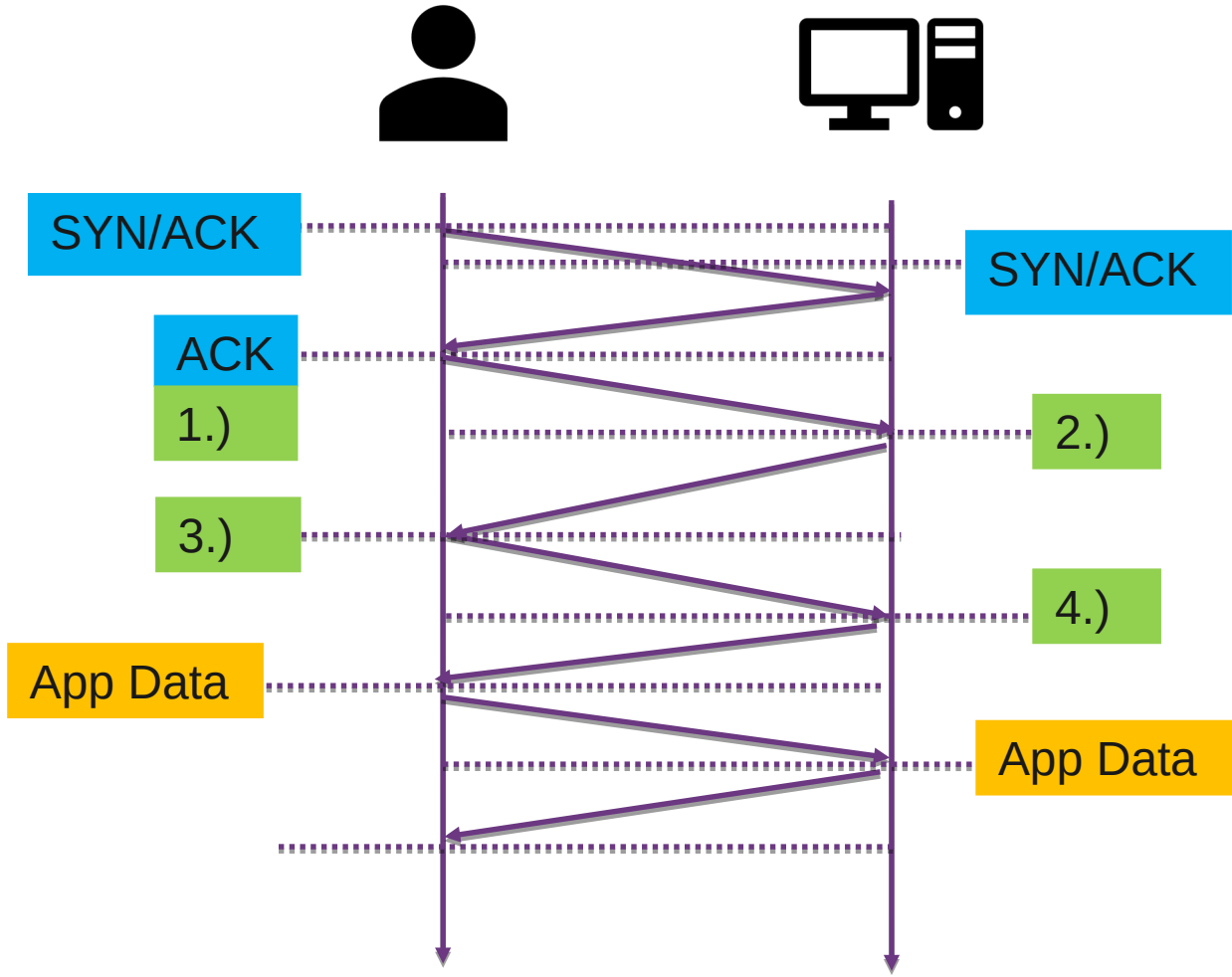
TCP Considerations

- Fallacy 2: Latency is zero
  - Nürnberg data center: 12ms, Australia: 300ms
  - Ping ftp.au.debian.org , Starklink + ~30ms
- Problem: TCP handshake is not flexible
  - You need a handshake (1RT)
    1) If you want to make sure the other side accepts packets (and not drop it) - ensure both sides are ready to transmit and receive data
    2) If you want to exchange public / private keys
  - TCP supports 1) but not 2)
    - Use another security layer for 2), but a security layer needs at least 1 RT
    - TCP + Security = at least 2 RT
      - Nürnberg + Starlink:2 x (12 + 30ms) = 84ms
      - Australian: (2 x 300ms) = 600ms
- TCP + Security at least 2 RT
  - DNS query may be required too: 3 RT
  - Old security protocols add RT: 4RT
- Worst case: Starlink/Australia/DNS/TCP/old sec: 1.4s before data can be sent → new protocols on the way (HTTP/3)

OST

Layer 4 – TCP + TLS

- Security: Transport Layer Security (TLS)
  1. "client hello" lists cryptographic information, TLS version, ciphers/keys
  2. "server hello" chosen cipher, the session ID, random bytes, digital certificate (checked by client), optional: "client certificate request"
  3. Key exchange using random bytes, now server and client can calc secret key
  4. "finished" message, encrypted with the secret key
  - 3 RTT to send first byte, 4RTT to receive first byte



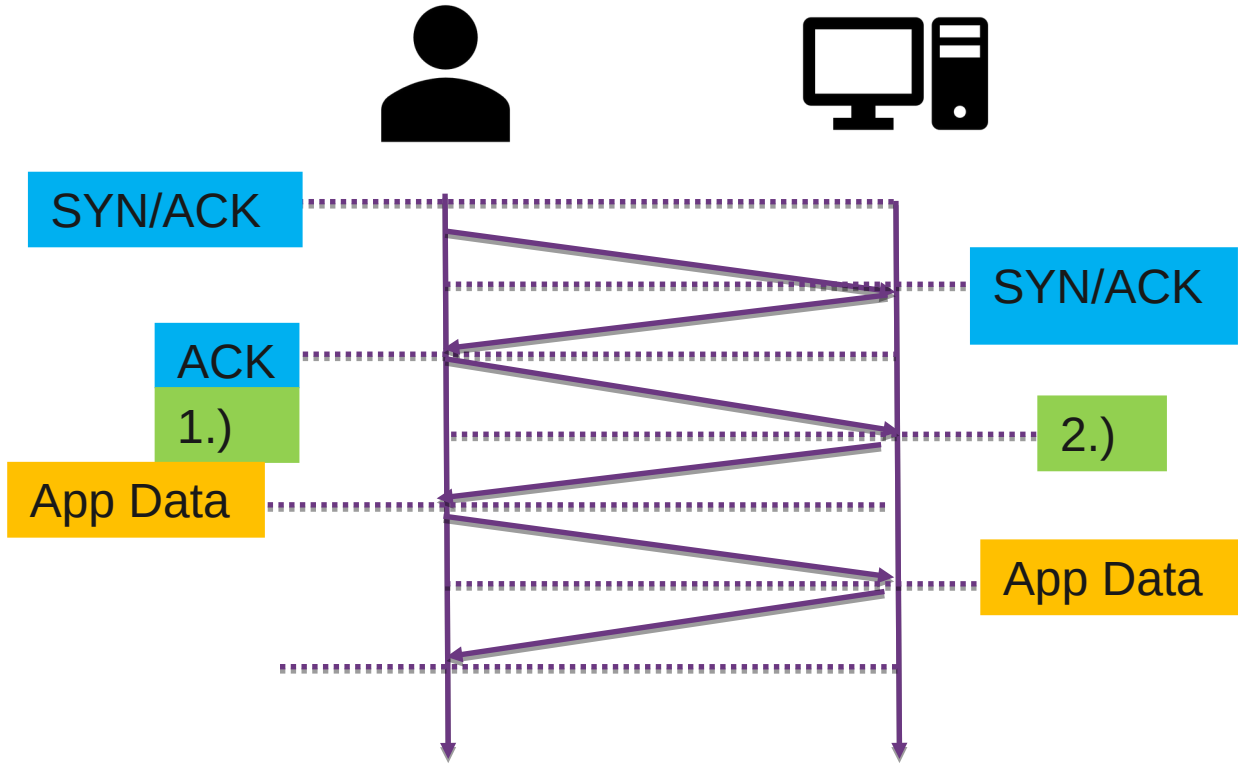
https://www.ibm.com/support/knowledgecenter/en/SSFKSJ_8.0.0/com.ibm.mq.sec.doc/q009930_.htm

OST

```
PING sydney.edu.au (129.78.5.8) 56(84) bytes of data.
64 bytes from scilearn.sydney.edu.au (129.78.5.8): icmp_seq=1 ttl=233 time=307 ms
64 bytes from scilearn.sydney.edu.au (129.78.5.8): icmp_seq=2 ttl=233 time=305 ms
64 bytes from scilearn.sydney.edu.au (129.78.5.8): icmp_seq=3 ttl=233 time=305 ms
```

Layer 4 – TCP + TLS

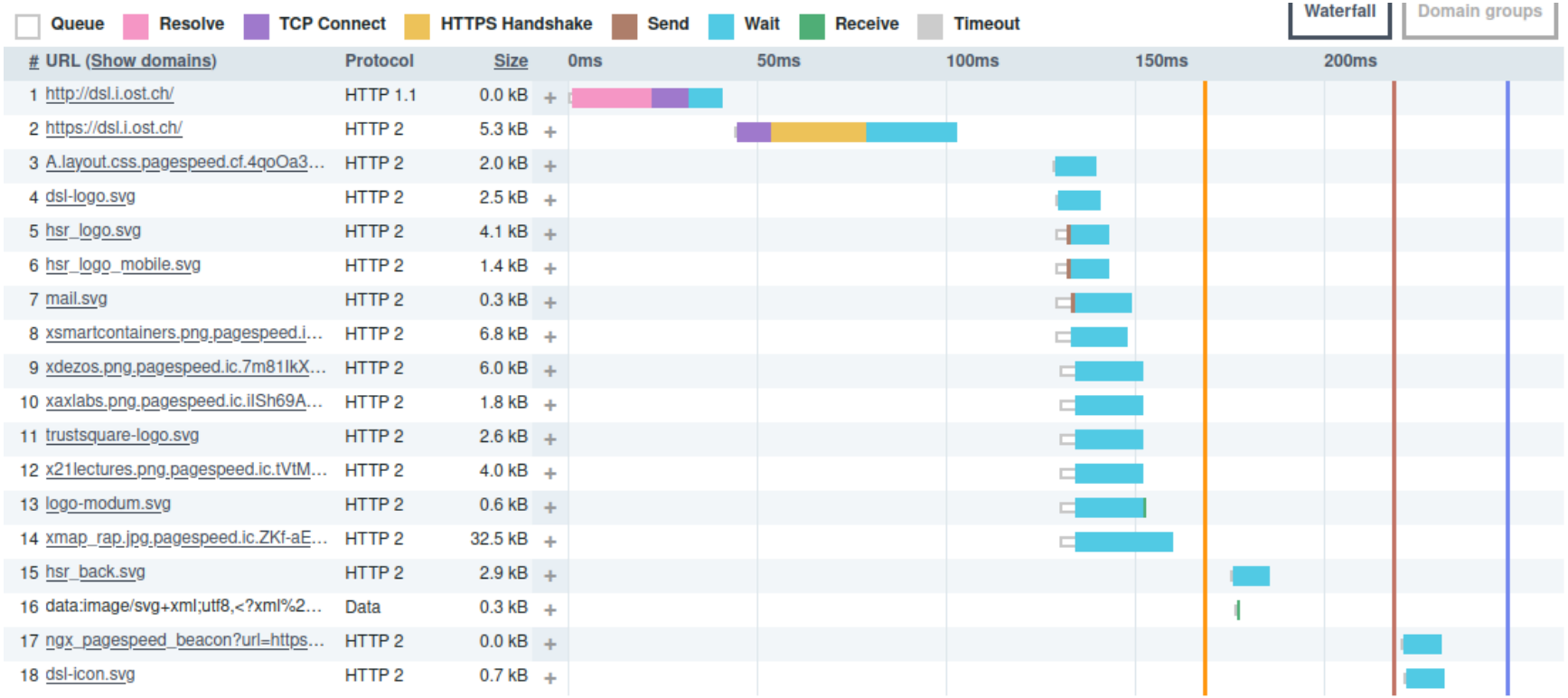
- TCP + TLS handshake:
  - 1RTT - Ping to Australia: 329ms
  - 3RTT = 987ms! No data sent yet
- TLS 1.3, finished Aug 2018
  - 1 RTT instead of 2
    - 1.) Client Hello, Key Share
    - 2.) Server Hello, key Share, Verify Certificate, Finished
  - 95% of browsers used already support it



OST

Layer 4 – TCP + TLS

- Website Speed Test [link]
  - Resolve → DNS, TCP Connect → TCP Handshake, HTTPS Handshake → TLS/SSL

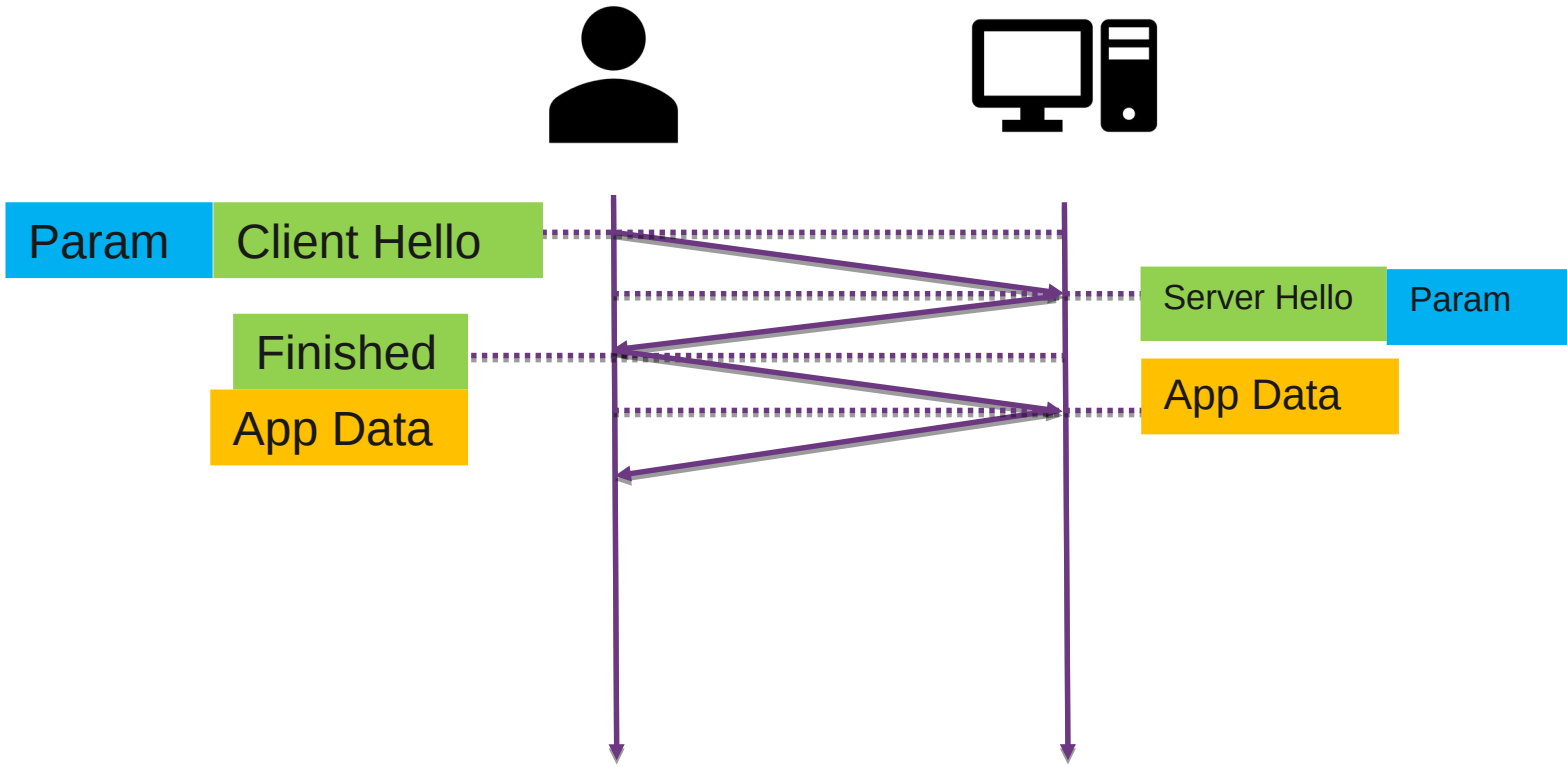


#	URL (Show domains)	Protocol	Size
1	http://dsl.i.ost.ch/	HTTP 1.1	0.0 kB
2	https://dsl.i.ost.ch/	HTTP 2	5.3 kB
3	A.layout.css.pagespeed.cf.4qoOa3...	HTTP 2	2.0 kB
4	dsl-logo.svg	HTTP 2	2.5 kB
5	hsr_logo.svg	HTTP 2	4.1 kB
6	hsr_logo_mobile.svg	HTTP 2	1.4 kB
7	mail.svg	HTTP 2	0.3 kB
8	xsmartcontainers.png.pagespeed.i...	HTTP 2	6.8 kB
9	xdezos.png.pagespeed.ic.7m81IkX...	HTTP 2	6.0 kB
10	xaxlabs.png.pagespeed.ic.ilSh69A...	HTTP 2	1.8 kB
11	trustsquare-logo.svg	HTTP 2	2.6 kB
12	x21lectures.png.pagespeed.ic.tVtM...	HTTP 2	4.0 kB
13	logo-modum.svg	HTTP 2	0.6 kB
14	xmap_rap.jpg.pagespeed.ic.ZKf-aE...	HTTP 2	32.5 kB
15	hsr_back.svg	HTTP 2	2.9 kB
16	data:image/svg+xml;utf8,<?xml%2...	Data	0.3 kB
17	ngx_pagespeed_beacon?url=https...	HTTP 2	0.0 kB
18	dsl-icon.svg	HTTP 2	0.7 kB

OST

QUIC / HTTP3

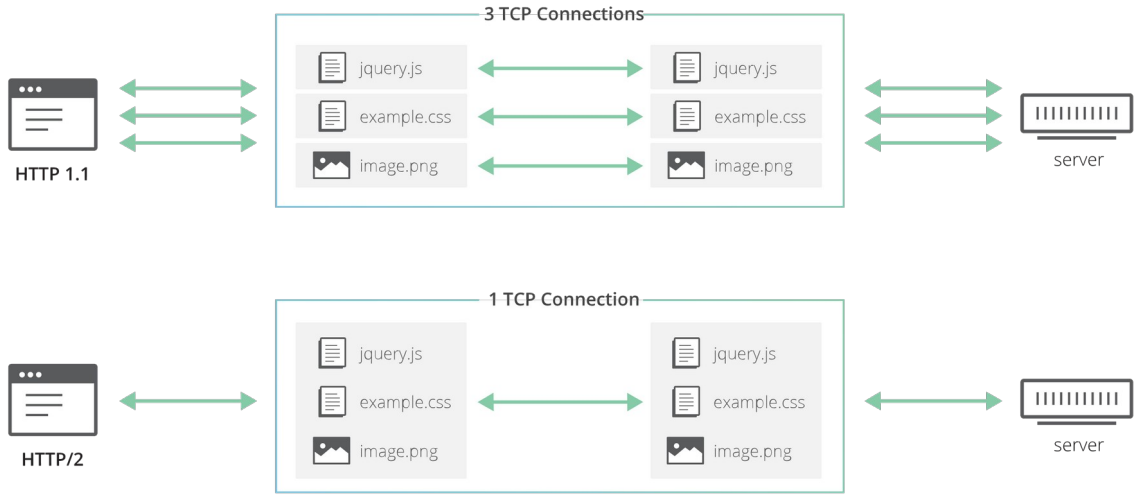
- QUIC: 1RTT connection + security handshake
  - For known connections: 0RTT
  - Built in security
  - Reports: state of HTTP
- Example Australia: from 987ms to 329ms



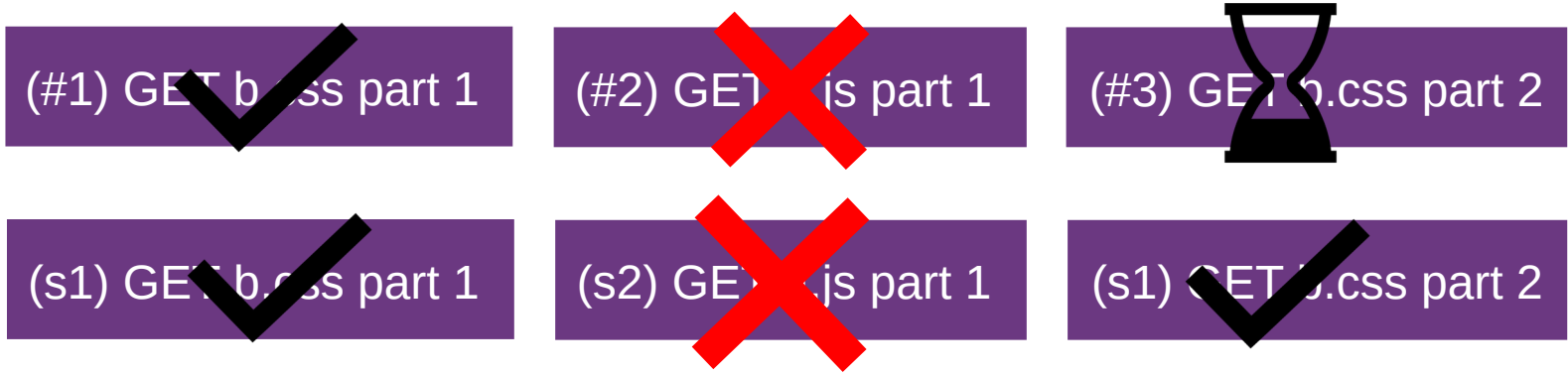
OST

QUIC / HTTP3

- Multiplexing in HTTP/2
  - HTTP/1 → HTTP/2
- HTTP/2: Head-of-line blocking
  - One packet loss, TCP needs to be ordered
  - QUIC can multiplex requests: one stream does not affect others
- HTTP/3 is great, but…
  - NAT → SYN, ACK, FIN, conntrack knows when connection ends, not with QUIC, timeouts, new entries, many entries
  - HTTP header compression, referencing previous headers
  - Many TCP optimizations



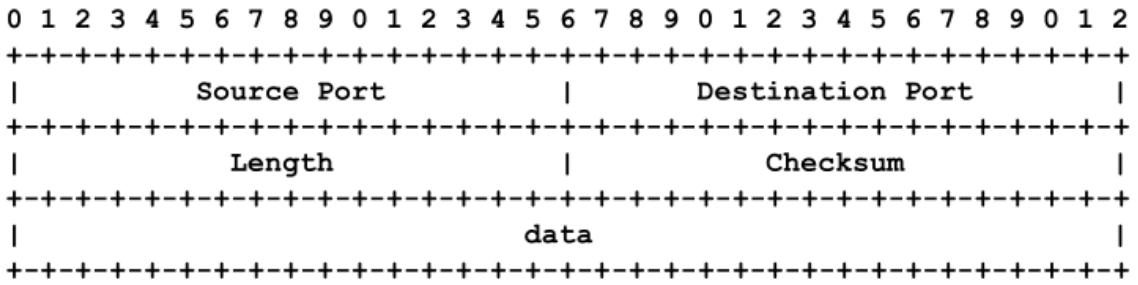
source: https://blog.cloudflare.com/the-road-to-quic/



OST

Layer 4 - Transport

- User Datagram Protocol (UDP)
  - UDP is used for DNS, streaming audio and video
  - Simple connectionless communication model
  - No guarantee
    - Delivery
    - Ordering
    - Duplicate protection

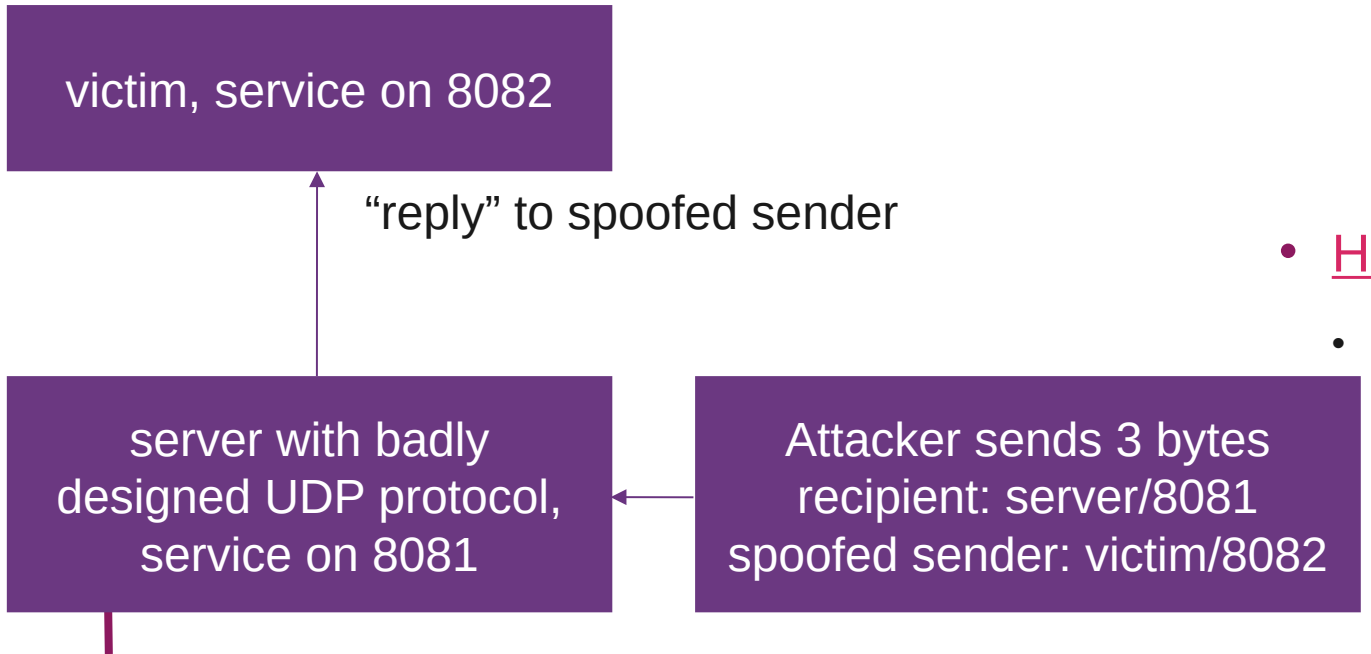


- SCTP (Stream Control Transmission Protocol)
  - Message-based
  - Allows data to be divided into multiple streams
  - Syn cookies - SCTP uses a four-way handshake with a signed cookie.
  - Multi-homing multiple IP addresses of endpoints
  - Not widely used: “We have been deploying SCTP in several applications now, and encountered significant problem with SCTP support in various home routers.” [link]
    - E.g., OpenWRT – not enabled by default
    - E.g., UFW - Uncomplicated Firewall – not supported
  - SCTP used by WebRTC, but tunneled over UDP

OST

Layer 4 - Transport

- DDoS Amplification Attacks
  - Request 10 bytes, reply 100 bytes → factor 10
- Local demo with server-ra/victim, and hping3
  - hping3 --udp IP -p 8081 -E test.tmp -d 6 -s 8082 -c 1



- Attacker in go/Java/node/c#
  - You need to spoof UDP packets, typically not supported in those languages
  - Go: `func DialUDP(network string, laddr, raddr *UDPAddr) (*UDPConn, error)`
    - laddr: we need to set here the victims IP/port
    - But go tries to bind to that
    - Not yours: “bind: cannot assign requested address”
- Hping3: Pen test tool
  - hping3 is a command-line oriented IP, TCP, UDP, ICMP and RAW-IP packet assembler

OST

Comparison – Transport Layer

TCP *	UDP *	SCTP *	QUIC *
Transport layer	Transport layer	Transport layer	Transport layer*
Connection oriented	Connection less	Connection oriented	Connection oriented
Reliable transfer	Unreliable transfer	Reliable transfer	Reliable transfer
Streams	Messages	Messages	Multistream
Guaranteed order	Unordered	User can choose	Guaranteed order
Widely used – HTTP/1, HTTP/2	Widely used – DNS, HTTP/3	WebRTC	HTTP/3
Flow and congestion control	No flow, congestion	Flow and congestion control	Flow and congestion co
Heavyweight	Lightweight	Heavyweight	Heavyweight
Error checking and recovery	Error checking, no recovery	Error checking and recovery	Integrity check

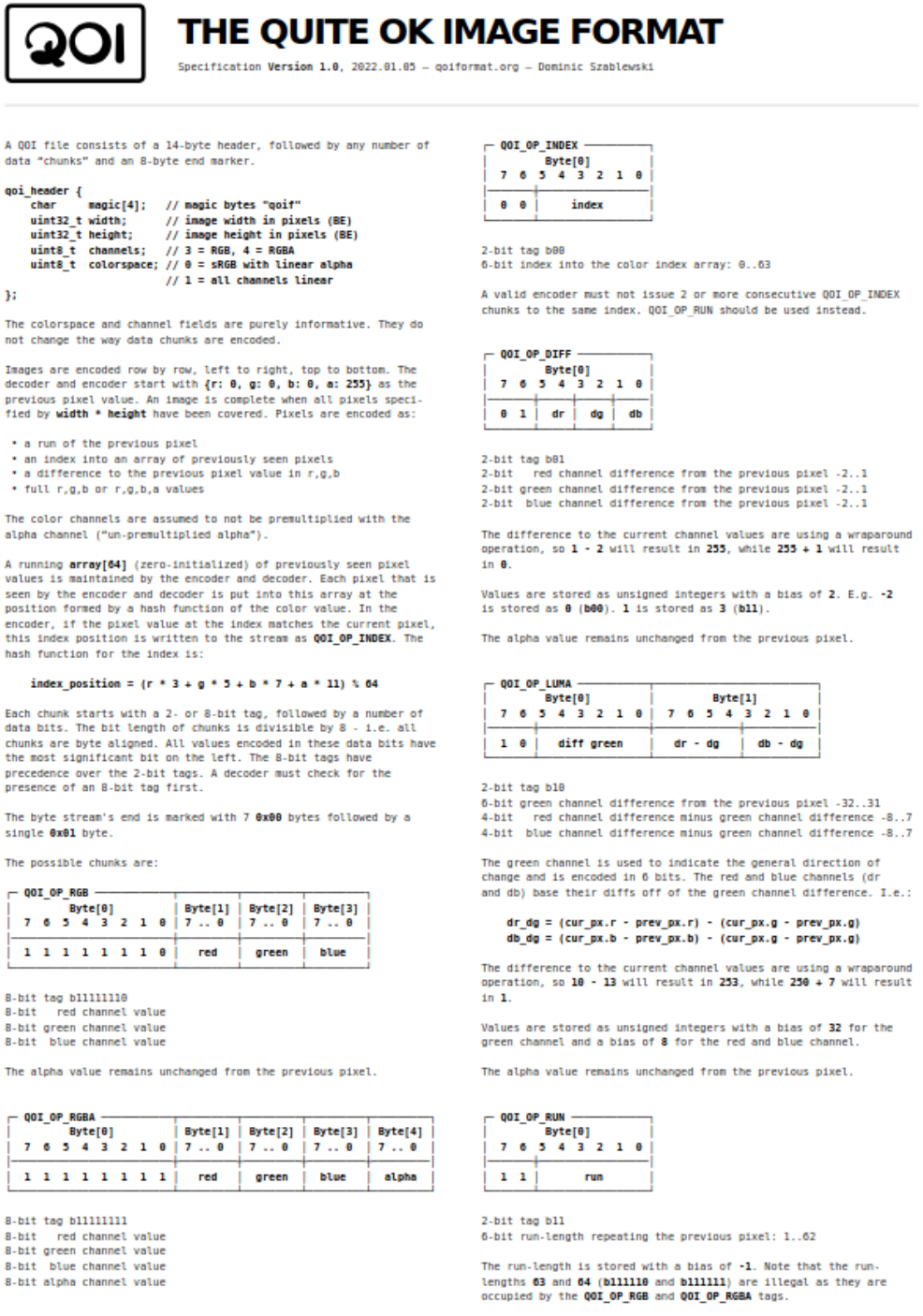
Alternatives?

- KCP - no encryption, GFCP - a KCP variant, UDT - unmaintained
- utp4j - Micro Transport Protocol for Java – handed over to tribler (Delft University of Technology)
  - 0-RTT Protocol in Golang [link]
  - ATP – A P2P Protocol [link]
  - P2P Library in Golang (BA) [link] [link]
- QUIC – clever ways to save bytes, but comes with complexity

OST

Complexity

- I like simple solutions, *e.g.*,
- QOI – simpler version of PNG
  - Encoder / decoder in 300 loc
  - PNG generates smaller images [link], QOI is much faster
  - Specification: 1 page
    - PNG – dictionary based compression with Huffman coding, 92 pages
  - Compression not much worse, but a lot simpler



QOI

THE QUITE OK IMAGE FORMAT

Specification **Version 1.0**, 2022.01.05 – qoiformat.org – Dominic Szablewski

A QOI file consists of a 14-byte header, followed by any number of data "chunks" and an 8-byte end marker.

```
qoi_header {
    char     magic[4];   // magic bytes "qoif"
    uint32_t width;      // image width in pixels (BE)
    uint32_t height;     // image height in pixels (BE)
    uint8_t  channels;   // 3 = RGB, 4 = RGBA
    uint8_t  colorspace; // 0 = sRGB with linear alpha
                         // 1 = all channels linear
};
```

The colorspace and channel fields are purely informative. They do not change the way data chunks are encoded.

Images are encoded row by row, left to right, top to bottom. The decoder and encoder start with **{r: 0, g: 0, b: 0, a: 255}** as the previous pixel value. An image is complete when all pixels specified by **width * height** have been covered. Pixels are encoded as:

- a run of the previous pixel
- an index into an array of previously seen pixels
- a difference to the previous pixel value in r,g,b
- full r,g,b or r,g,b,a values

The color channels are assumed to not be premultiplied with the alpha channel ("un-premultiplied alpha").

A running **array[64]** (zero-initialized) of previously seen pixel values is maintained by the encoder and decoder. Each pixel that is seen by the encoder and decoder is put into this array at the position formed by a hash function of the color value. In the encoder, if the pixel value at the index matches the current pixel, this index position is written to the stream as **QOI_OP_INDEX**. The hash function for the index is:

```
index_position = (r * 3 + g * 5 + b * 7 + a * 11) % 64
```

Each chunk starts with a 2- or 8-bit tag, followed by a number of data bits. The bit length of chunks is divisible by 8 - i.e. all chunks are byte aligned. All values encoded in these data bits have the most significant bit on the left. The 8-bit tags have precedence over the 2-bit tags. A decoder must check for the presence of an 8-bit tag first.

The byte stream's end is marked with 7 **0x00** bytes followed by a single **0x01** byte.

The possible chunks are:

QOI_OP_RGB

Byte[0]	Byte[1]	Byte[2]	Byte[3]
7 6 5 4 3 2 1 0	7 .. 0	7 .. 0	7 .. 0
1 1 1 1 1 1 1 0	red	green	blue

8-bit tag b11111110
8-bit red channel value
8-bit green channel value
8-bit blue channel value

The alpha value remains unchanged from the previous pixel.

QOI_OP_RGBA

Byte[0]	Byte[1]	Byte[2]	Byte[3]	Byte[4]
7 6 5 4 3 2 1 0	7 .. 0	7 .. 0	7 .. 0	7 .. 0
1 1 1 1 1 1 1 1	red	green	blue	alpha

8-bit tag b11111111
8-bit red channel value
8-bit green channel value
8-bit blue channel value
8-bit alpha channel value

QOI_OP_INDEX

Byte[0]
7 6 5 4 3 2 1 0
0 0 | index

2-bit tag b00
6-bit index into the color index array: 0..63

A valid encoder must not issue 2 or more consecutive QOI_OP_INDEX chunks to the same index. QOI_OP_RUN should be used instead.

QOI_OP_DIFF

Byte[0]
7 6 5 4 3 2 1 0
0 1 | dr | dg | db

2-bit tag b01
2-bit red channel difference from the previous pixel -2..1
2-bit green channel difference from the previous pixel -2..1
2-bit blue channel difference from the previous pixel -2..1

The difference to the current channel values are using a wraparound operation, so **1 - 2** will result in **255**, while **255 + 1** will result in **0**.

Values are stored as unsigned integers with a bias of **2**. E.g. **-2** is stored as **0** (**b00**). **1** is stored as **3** (**b11**).

The alpha value remains unchanged from the previous pixel.

QOI_OP_LUMA

Byte[0]	Byte[1]
7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
1 0 | diff green	dr - dg | db - dg

2-bit tag b10
6-bit green channel difference from the previous pixel -32..31
4-bit red channel difference minus green channel difference -8..7
4-bit blue channel difference minus green channel difference -8..7

The green channel is used to indicate the general direction of change and is encoded in 6 bits. The red and blue channels (dr and db) base their diffs off of the green channel difference. I.e.:

```
dr_dg = (cur_px.r - prev_px.r) - (cur_px.g - prev_px.g)
db_dg = (cur_px.b - prev_px.b) - (cur_px.g - prev_px.g)
```

The difference to the current channel values are using a wraparound operation, so **10 - 13** will result in **253**, while **250 + 7** will result in **1**.

Values are stored as unsigned integers with a bias of **32** for the green channel and a bias of **8** for the red and blue channel.

The alpha value remains unchanged from the previous pixel.

QOI_OP_RUN

Byte[0]
7 6 5 4 3 2 1 0
1 1 | run

2-bit tag b11
6-bit run-length repeating the previous pixel: 1..62

The run-length is stored with a bias of **-1**. Note that the run-lengths **63** and **64** (**b111110** and **b111111**) are illegal as they are occupied by the **QOI_OP_RGB** and **QOI_OP_RGBA** tags.

Lets Implement a P2P Friently Transport Protocol!

- Learn working with LLMs
  - Testing local and remote LLMs
- So, I vibecoded it
  - “Hey ChatGPT, I need a protocol implementation in golang that is simpler than QUIC”
  - ... it did not work
  - But at least, ChatGPT told me its a great idea!
- Working with LLMs
  - Iterate, iterate, iterate
  - Understand what the LLM generated and improve or discard it
- QOTP, making the protocol easier (but less condense)
  - Variable-length integer encoding (QUIC) vs fixed-length encoding (QOTP)
  - Multi-ack (QUIC) vs one packet, one ack (QOTP)
  - TLS support (QUIC) vs 1 supported crypto algorithm (QOTP)
- Why → best way to learn about protocols

OST