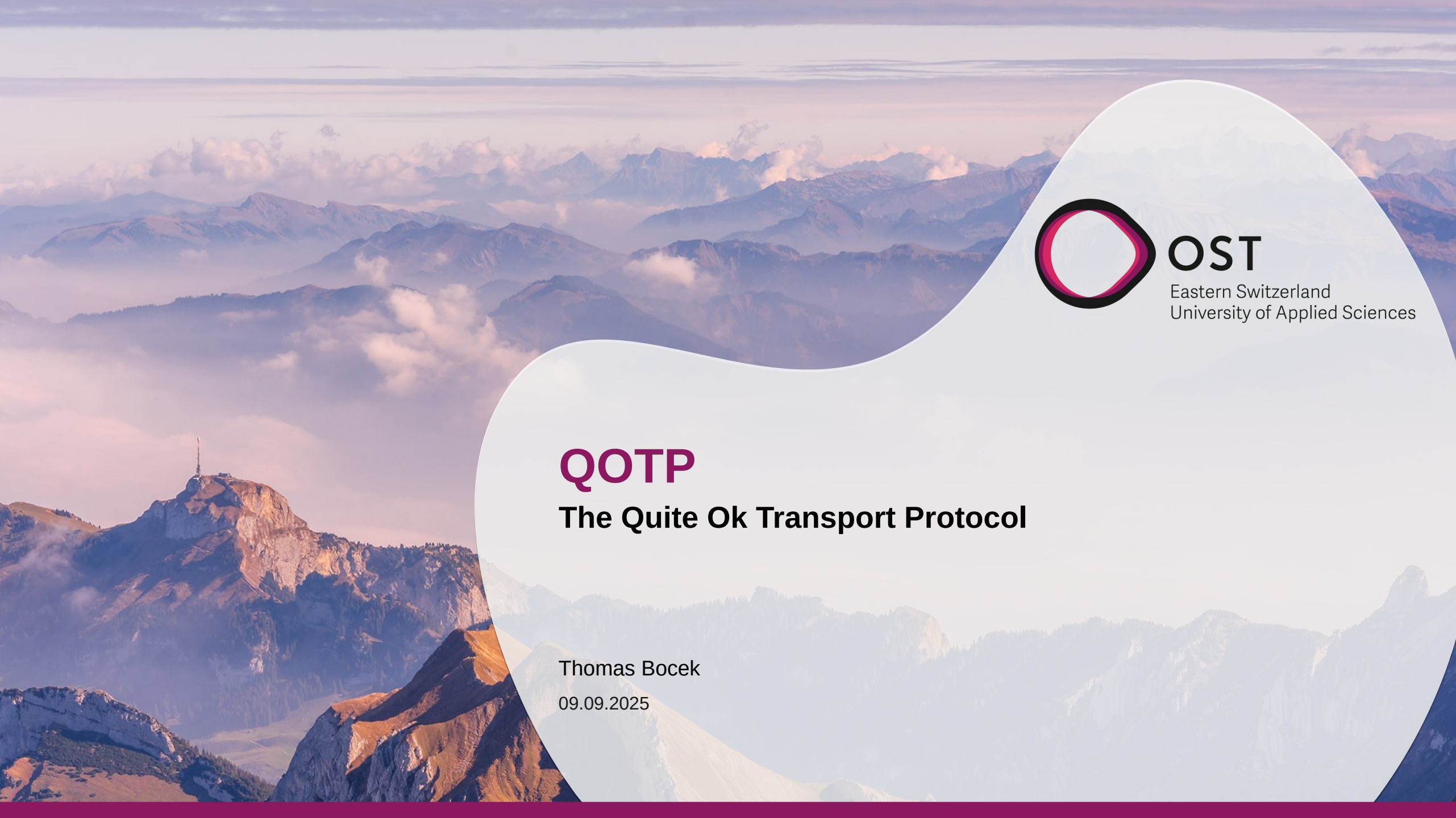


OST
Eastern Switzerland
University of Applied Sciences

# QOTP

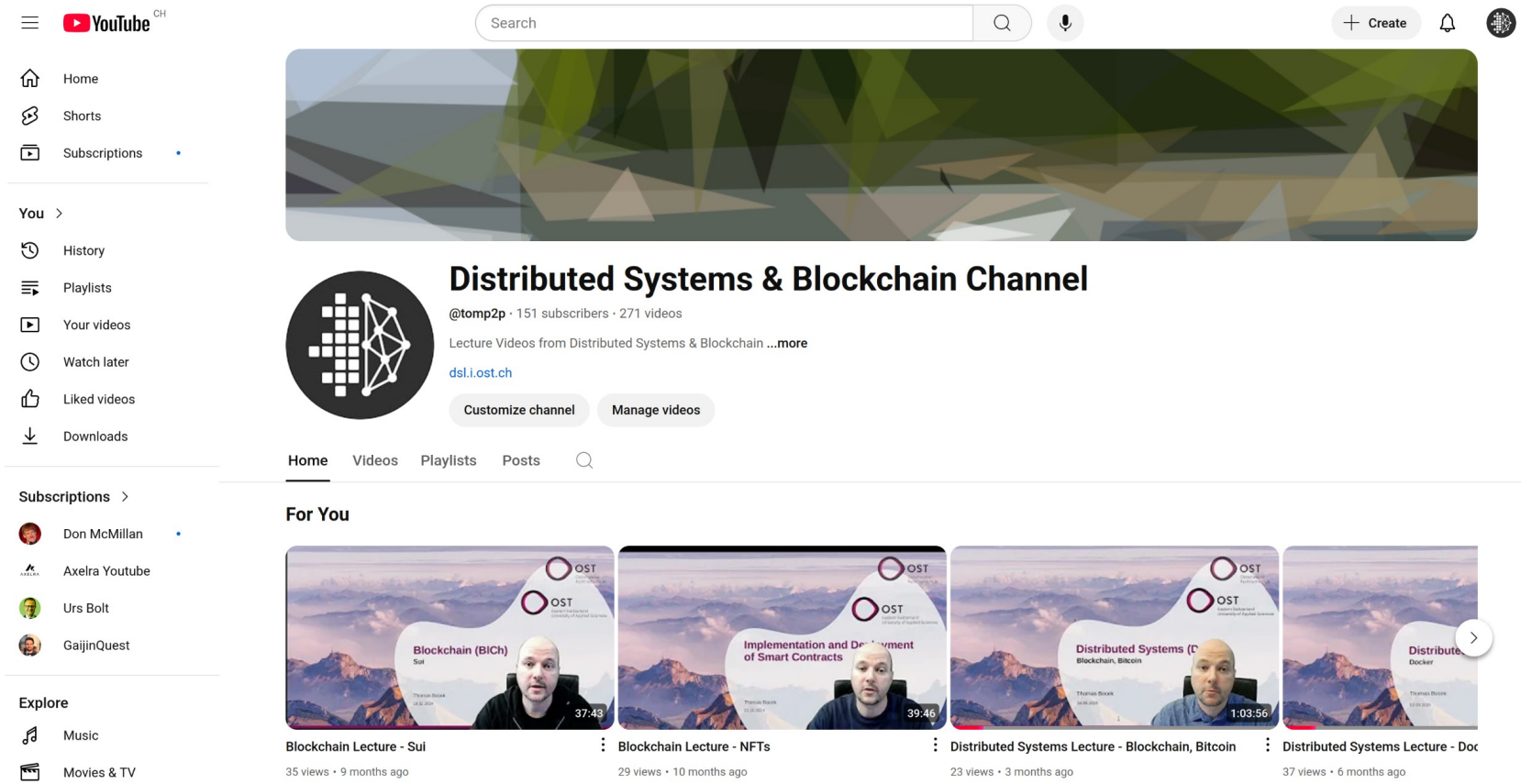
## The Quite Ok Transport Protocol

Thomas Bocek

09.09.2025

# About Me / Introduction

- Professor of Computer Science (Architecture, Distributed Systems, Blockchain) at OST
  - Architucture, Distributed systems, Blockchains
  - On YouTube, since Corona
    - Like and subscribe 😀

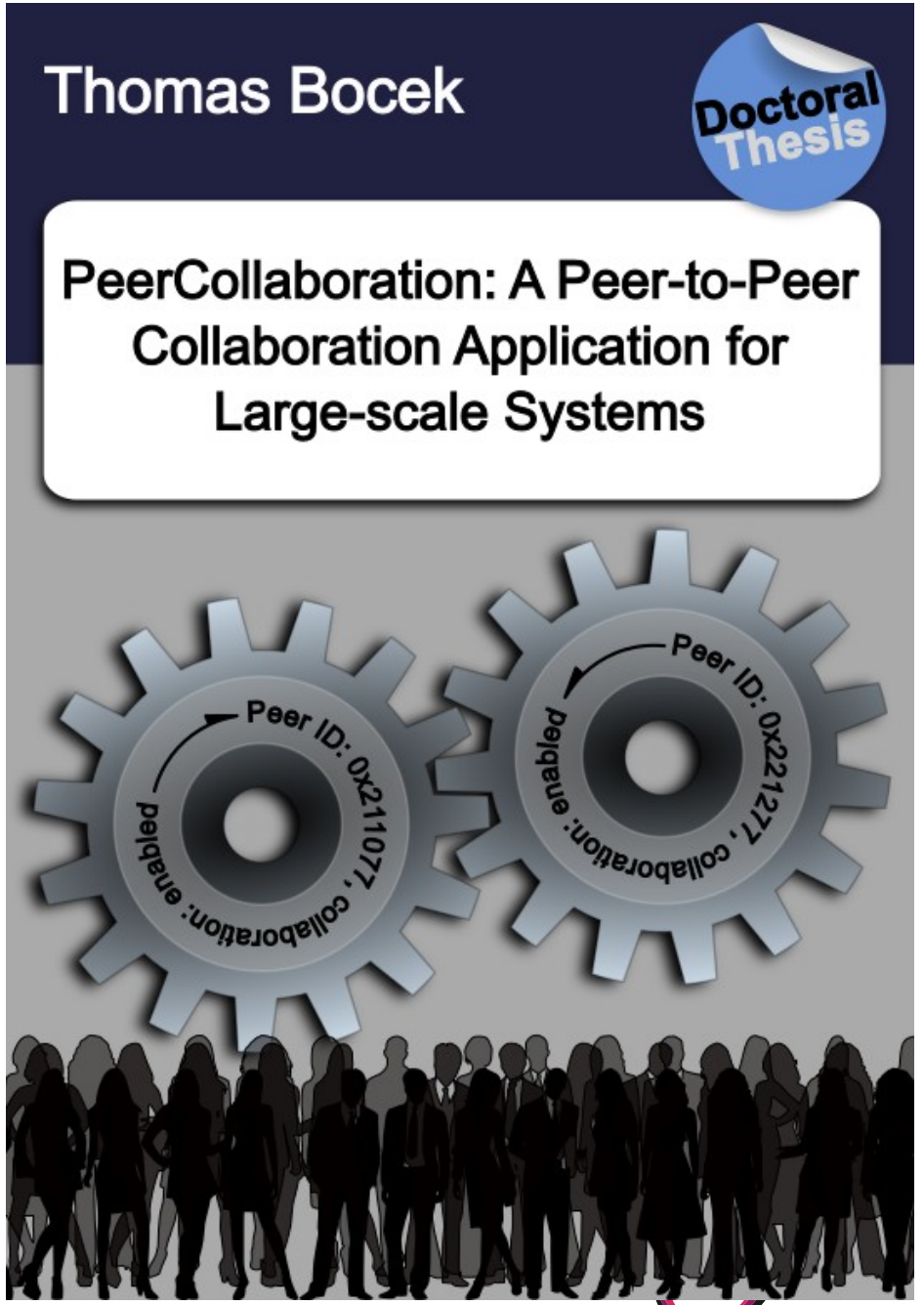


# PhD Topic

- Included TomP2P, a DHT implemented in Java (archived)
  - Used TCP, but TCP has many issues…
  - Not P2P friendly (hole-punching) – predict sequence numbers (besides predicting ports)
  - “too many open files in system”, 2 files per TCP connection

```
[root@12core qotp]# ulimit -n
1024
```

  - “time wait state” - TCP connection termination process → port exhaustion - issues with many short lived connections



# 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

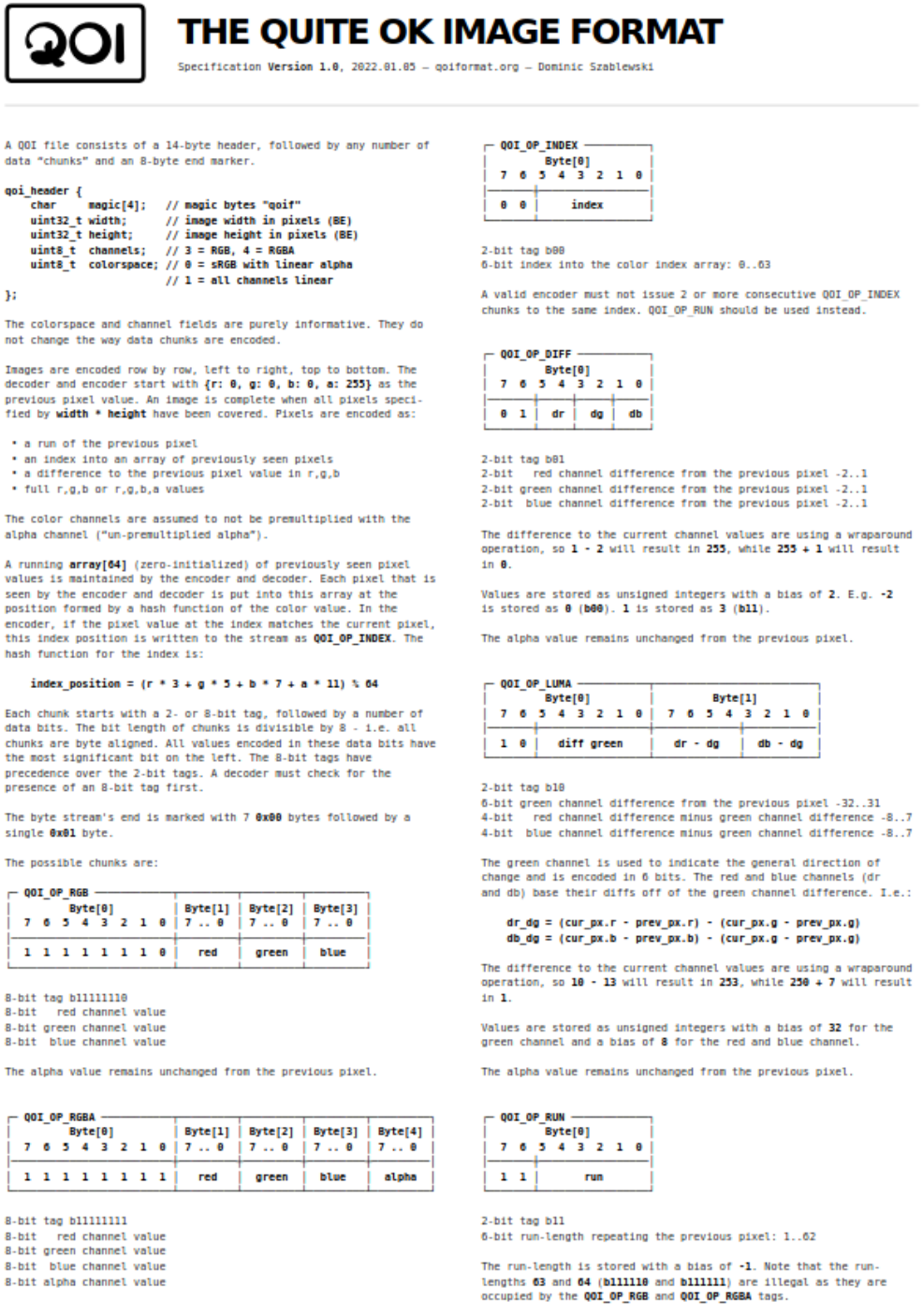




OST

# 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!

- I can implement it with AI
- 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
  - Very good at “semi-automated” tasks
    - “Write a sortedmap / linkedmap in golang” → very good
    - “For this code, write me testcases” → okish, but need to verify
  - Very bad at “thinking”
    - “Implement rcv_wnd in an efficint manner” → no chance
    - “Where in the code is the best place to add XYZ” → you will get some random location



OST

# QUIC vs. QOTP

- QUIC
  - Variable-length integer encoding:
    - First 2 bits | Total Length | Value Range

      ---|---|---

      00 | 1 byte | 0 to 63

      01 | 2 bytes | 0 to 16,383

      10 | 4 bytes | 0 to 1,073,741,823

      11 | 8 bytes | 0 to 4,611,686,018,427,387,903
  - Max. 62bit, complicated in my implementation: not knowing the size beforehand
- QOTP: 1 byte paket header defines size
  - Knowing the size beforehand
- QOTP
  - Bit 1-2 in packet header:
    - 00: No ACK/Data with 24bit → 8 bytes

      01: No ACK/Data with 48bit → 11 bytes

      10: ACK with 24bit/Data with 24bit → 17 bytes

      11: ACK with 48bit/Data with 48bit → 23 bytes
  - QUIC Flow Control
    - Connection based flow control with varint
    - Stream based flow control with varint
  - QOTP Bit 3-7 in packet header only congestion:
    - 0 | 0-511 | 0

      1 | 512-1023 | 768

      2 | 1024-2047 | 1536 (1.5KB)

      17 | 33554432-67108863 | 50331648 (48MB)

      30 | 274877906944+ | 412316860416 (384GB)
  - Necessary to know the exact receiver window?



OST

# QUIC vs. QOTP

- QUIC – Security built-in (TCP no security)
- TLS 1.3
  - Key Exchange Algorithms - secp256r1, secp384r1, secp521r1, X25519, X448
  - Symmetric Encryption + Integrity – AES_128_GCM_SHA256, AES_256_GCM_SHA384, CHACHA20_POLY1305_SHA256
  - Digital Signatures – RSA-PSS-RSAE-SHA256, RSA-PSS-RSAE-SHA384, RSA-PSS-RSAE-SHA512, ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, ed25519, ed448
  - Key Derivation – HKDF-SHA256, HKDF-SHA384
  - TLS: 9 primary RFCs and 48 extensions and informational RFCs, totalling 57 RFC [wikipedia]

- QOTP – Security built-in
  - X25519, CHACHA20_POLY1305_SHA256
    - No key renegotiation



OST

# QOTP Features

- **1 byte** crypto header → exact size known
- **1 byte** protocol header → exact size known
- First **crypto key** exchange can be **out of band** (e.g., TXT field of DNS), or **in band**
  - **0-RTT** possible (no perfect forward secrecty + first packet filled to max ~1400 bytes)
- **Always encrypted**
- **Stream** support, flow control on connection only
- Congestion control: **BBR** (Bottleneck Bandwidth and Round-trip propagation time)
- **No delayed Ack** – 1 ACK = 1 packet
  - Simpler time measurement, simpler header
- FIN/ACK teardown with timeout
  - **Not yet implemented**
- MTU detection
  - **Not yet implemented**
- Less than 3k LoC



OST

# Goals

- Make SPAs load faster
  - 1st packet: GET request
  - 1st reply: compressed HTML/Javascript in ~1.3KB with relevant Fetch requests
  - /api call with the 2nd packet
- Current approach: SPAs with backend rendering (SvelteKit, Next.js)
  - Go back to no backend rendering
- PrevelteKit
  - Server-Side Pre Rendering (SSPR) with hydration / jdom
- qh:// - current Bachelor Thesis
  - Quite Ok HTTP on top of QOTP
  - No Let's encrypt
    - Key material via DNS TXT record
  - Certificates needed?
    - Yes/no, but **never** for encryption, for signatures
      - Where to put? `Trailer: Signature, Certificate`
      - Streaming? `Transfer-Encoding: chunked`
    - No need for Oracles in the blockchain world → JSON signed by qh:// if certificate provided



OST

**Questions?**

Prof. Dr. Thomas Bocek
thomas.bocek@ost.ch