University of Glasgow

Internet Protocols Laboratory

Improving Secure Connection Establishment

Networked Systems (H)

Lecture 4

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Lecture Outline

- Limitations of TLS v1.3
  - Slow Connection Establishment
  - Metadata Leakage
  - Protocol Ossification
- QUIC Transport Protocol
  - Performance, security, and avoiding ossification
  - Unified protocol handshake
  - Reliable multi-streaming transport

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

Limitations of TLS v1.3

- TLS v1.3 is a significant improvement on prior versions: faster and more secure
- TLS v1.3 runs within a TCP connection:
  - Must wait for TCP connection establishment
  - Some metadata leakage
- Implementations of TLS are ossified and hard to extend

CC BY ND

Limitations of TLS v1.3

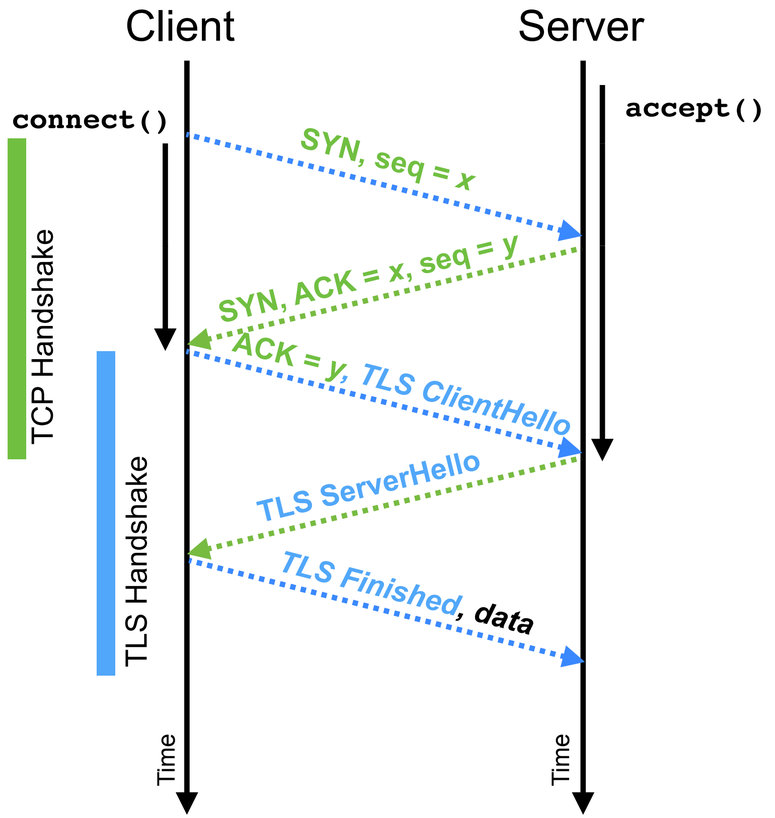


- TLS v1.3 is a tremendous success
  - **Significant security improvements** compared to TLS v1.2
    - Removed support for older and less secure encryption and key exchange algorithms
    - Removed support for secure algorithms that have proven difficult to implement correctly
  - Some **performance improvements** to the initial handshake and with 0-RTT mode
- Despite this, TLS v1.3 has some limitations that are hard to fix
  - Connection establishment is still relatively slow
  - Connection establishment leaks potentially sensitive metadata
  - The protocol is ossified due to middlebox interference

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TLS v1.3 Connection Establishment Performance (1/2)

- TCP connection established as usual:
  - **`SYN`** → **`SYN+ACK`** → **`ACK`**
- TLS handshake protocol runs inside TCP connection:
  - TLS **`ClientHello`** sent with final **`ACK`**
  - TLS **`ServerHello`** sent in response
  - TLS **`Finished`** message concludes, and carries initial secure data record
- First data sent 2x RTT after connection establishment starts
- Earliest response received 3x RTT after connection establishment starts



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TLS v1.3 Connection Establishment Performance (2/2)

- Average web page comprises 1.7 MB of data, fetched as 69 HTTP requests, using 15 TCP connections
- 83% of HTTP requests run over TLS

  https://httparchive.org/reports/page-weight

- **Enormous amount of time wasted, waiting for TCP and TLS connection establishment handshakes**

Destination	Average RTT (ms)
London	72.5
New York	153.3
Los Angeles	221.2
Sydney	381.2

RTT measurements (ping times) from residential site in Glasgow

- Can we speed up TLS connection setup?
  - 0-RTT Connection Reestablishment – speed-up connections to known servers
  - Concurrent TCP and TLS handshake – speed-up connections to all servers

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

0-RTT Connection Reestablishment (1/4)

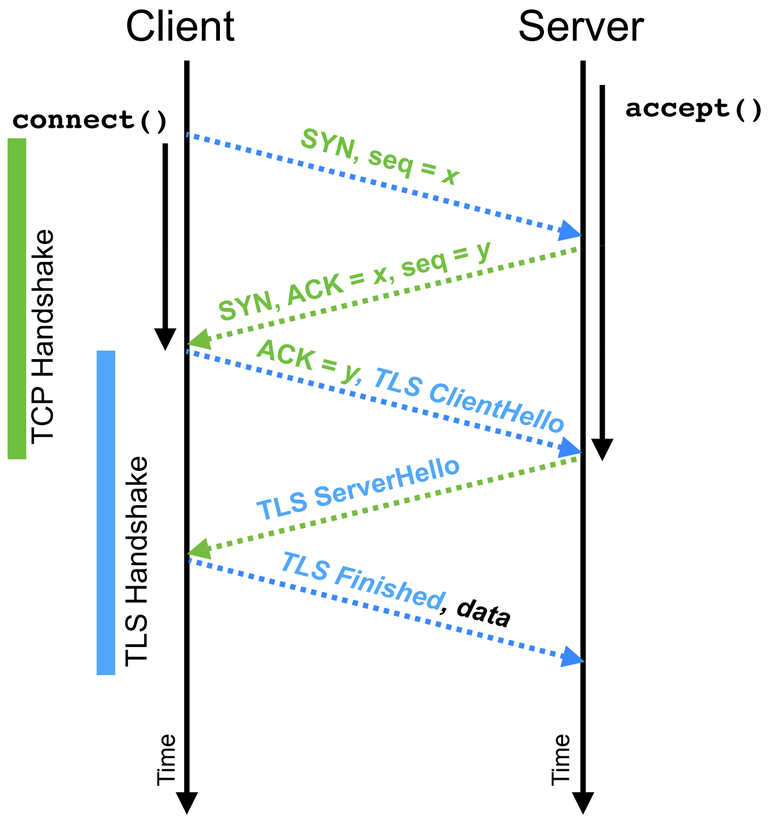
- Common to connect to a previously known TLS server – is it possible to shortcut the connection establishment in such cases?

- Need to understand:
  - What is the role of the TLS handshake?
  - How to encrypt initial data?
  - What are the potential risks?

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

0-RTT Connection Reestablishment (2/4)

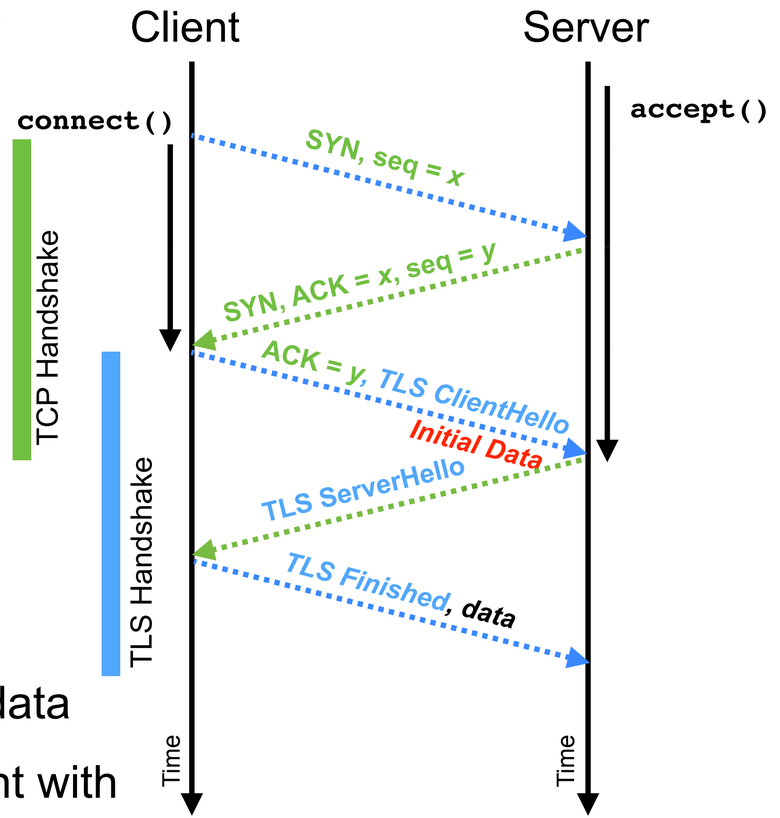
- **The TLS handshake does two things:**
  - 1 – uses public key cryptography to establish an **ephemeral session key** that is used to encrypt the data
    - The **`ClientHello`** and **`ServerHello`** are used to exchange material used to derive a session key – using ECDHE key negotiation
    - The session is ephemeral – different for each connection; derived from the public keys and a random value
    - The ephemeral session key provides **forward secrecy** – each connection has a unique key; if the encryption key for one session leaks, it doesn't help an attacker break other sessions
  - 2 – retrieves the server's certificate, that allows the client to verify the identify of the server via a PKI
    - The **`ServerHello`** contains the certificate



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

0-RTT Connection Reestablishment (3/4)

- **Can we send encrypted initial data to speed up the connection?**
  - Usual TLS encryption relies on data exchanged in the handshake, so can't be used for initial data sent before the handshake finishes
  - Can reuse a **pre-shared key** agreed in previous TLS session
- In a previous TLS connection:
  - Server sends **`PreSharedKey`** with **`SessionTicket`** to identify key
- When reestablishing a connection:
  - Client sends **`SessionTicket`**, initial data encrypted with corresponding **`PreSharedKey`**, along with **`ClientHello`**
  - Server uses **`SessionTicket`** to find saved **`PreSharedKey`**, decrypt the data
  - **`ClientHello`** and **`ServerHello`** complete usual key exchange; data sent with **`ServerHello`** and later protected using ephemeral session key



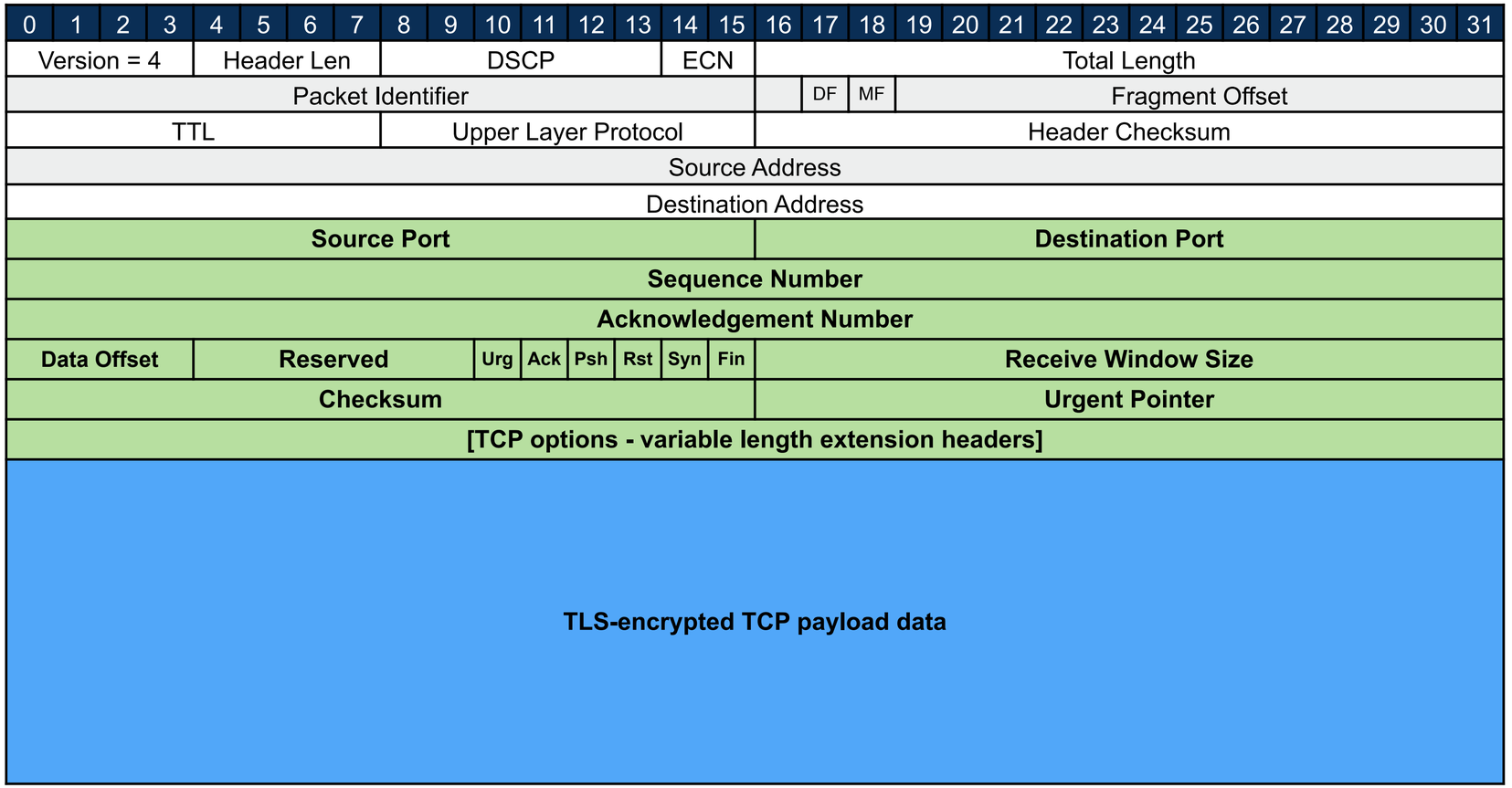
Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

0-RTT Connection Reestablishment (4/4)

- **What are the potential risks?**
  - 0-RTT data sent with **`ClientHello`** using a **`PreSharedKey`** is not forward secret
    - Use of **`PreSharedKey`** links TLS connections – if session where **`PreSharedKey`** is distributed is compromised, 0-RTT data sent using that key in future connections will also be compromised
  - 0-RTT data sent with **`ClientHello`** using a **`PreSharedKey`** is subject to replay attack
    - The 0-RTT data is accepted during TLS connection establishment
    - If on-path attacker captures and replays the TCP segment with the **`ClientHello`**, **`SessionTicket`**, and data protected with the **`PreSharedKey`**, that data will be accepted by the server again
      - The server will respond to the replay, trying to complete the handshake – this might fail
      - But – by then, the data will have been accepted
    - Ensure 0-RTT data is **idempotent** to avoid this risk
- **Be careful using 0-RTT data in TLS v1.3** – trades performance for safety

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TLS v1.3 Metadata Leakage (1/2)



IP exposes addresses

TCP exposes port numbers and connection metadata

Is there a privacy concern?

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

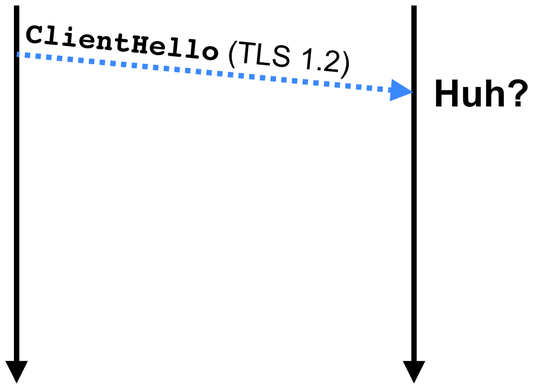
TLS v1.3 Metadata Leakage (2/2)

- When TLS is used with HTTPS, **`ClientHello`** includes the Server Name Indication (SNI) extension
  - Identifies requested site, so server knows what public key to use in **`ServerHello`**
    - Required to support shared hosting, with multiple websites on one server
    - Has to be unencrypted – sent before session keys are negotiated
    - Can't encrypt with **`PreSharedKey`**, since that's provided by server, and goal is to select the server
- **A privacy concern with TLS v1.3**
  - See https://datatracker.ietf.org/doc/draft-ietf-tls-esni/ for work-in-progress attempt to resolve

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TLS v1.3 Protocol Ossification (1/3)

- TLS is widely implemented, but many poor quality implementations:
  - Some TLS servers fail if **`ClientHello`** uses unexpected version number, rather than try to negotiate older version
  - Some firewalls block connections if **`ClientHello`** is structured differently to that used by TLS 1.2 and earlier, even if TLS 1.3 is signalled

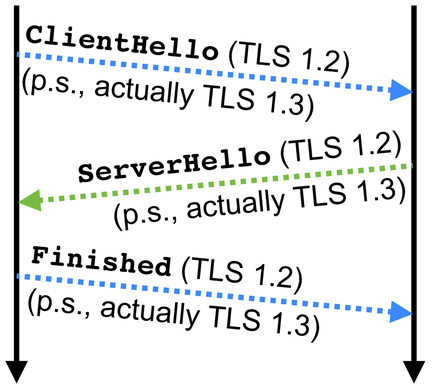


- Original design of TLS 1.3 changed **`ClientHello`**
  - Updated the version number (1.2 → 1.3)
  - Removed some now unused header fields
- Measurements showed this caused ~8% of TLS 1.3 connections to fail

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

TLS v1.3 Protocol Ossification (2/3)

- Later versions of TLS 1.3 changed the design to work around these bugs
  - Version number in **`ClientHello`** says **TLS 1.2**; unused header fields present with dummy values; extension header to **`ClientHello`** signals actual version
  - (Similar changes in **`ServerHello`**)
  - When TLS 1.3 client talks to TLS 1.3 server, version negotiated in extensions
  - When TLS 1.3 client talks to TLS 1.2 server, extension ignored and TLS 1.2 is negotiated



- **Protocol ossification is a significant concern**
  - TLS is not the only protocol to include such workarounds
  - Widely deployed faulty implementations constrain design of most protocols

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

How to Avoid Protocol Ossification?

- Ossification happens when extension mechanisms, or allowed flexibility, are not used
  - TLS 1.3 was released ten years after TLS 1.2
  - Allowed products to be built and deployed that didn't do version negotiation correctly, since no new versions to negotiate
  - Allowed products to be built that relied on the presence and order of fields in **`ClientHello`**, since all implementations included the same fields in the same order
- **Generate Random Extensions And Sustain Extensibility** (GREASE)
  - If the protocol allows extensions, send extensions
  - If the protocol allows different versions, negotiate different versions
  - **Do this even if you don't need to → "use it or lose it"**
    - Send meaningless dummy extensions that are ignored
    - Change the version number to prove you can

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

QUIC Transport Protocol

- **Development and basic features**
- Connection establishment; data transfer
- Avoiding ossification; limitations

CC BY ND

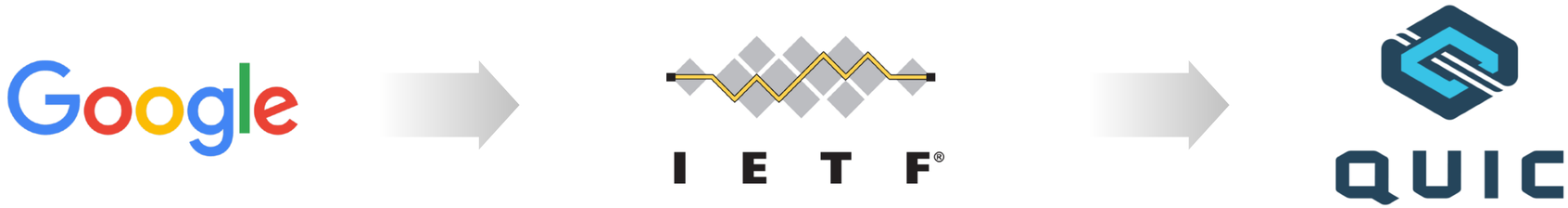
QUIC
Performance, Security, Avoiding Ossification

- What's wrong with TLS v1.3 over TCP?
  - Slow to connect – due to sequential TCP and TLS handshakes
  - Leaks some metadata
  - Ossified and hard to extend

- QUIC aims to replace TLS v1.3 and TCP with a single secure transport protocol
  - Reduce latency by overlapping TLS and transport handshake
  - Avoid metadata leakage via pervasive encryption
  - Avoid ossification via systematic application of GREASE and encryption

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

History of QUIC



- 2012-2015: Initial proposal and experimental deployment by Google
- 2016-2021: IETF QUIC working group
  - Specifications went through 34 drafts in IETF before RFCs were published – standardisation led to **significant differences and improvements compared to initial Google proposal**
    - RFC 8999 “Version-Independent Properties of QUIC” https://datatracker.ietf.org/doc/rfc8999/
    - RFC 9000 “QUIC: A UDP-Based Multiplexed and Secure Transport” https://datatracker.ietf.org/doc/rfc9000/
    - RFC 9001 “Using TLS to Secure QUIC” https://datatracker.ietf.org/doc/rfc9001/
    - RFC 9002 “QUIC Loss Detection and Congestion Control” https://datatracker.ietf.org/doc/rfc9002/
  - GitHub repo for the specification has ~2200 issues and ~2500 pull requests

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Overview (1/2)

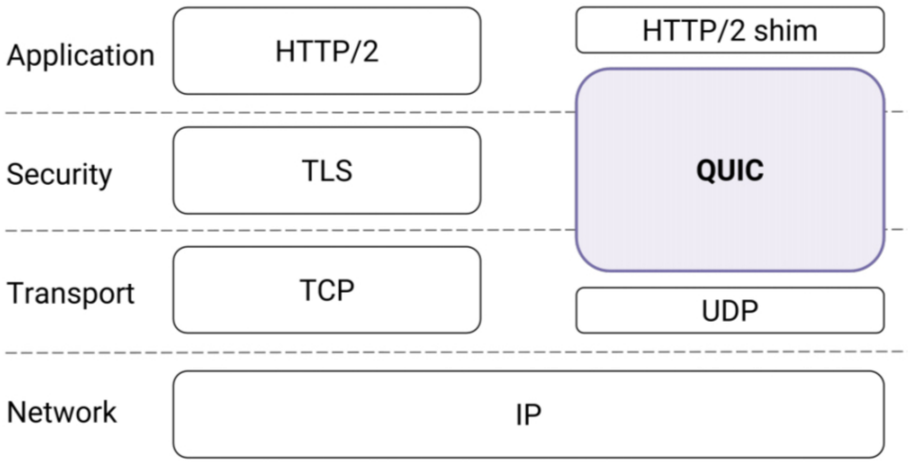


Figure 1: QUIC in the traditional HTTPS stack.

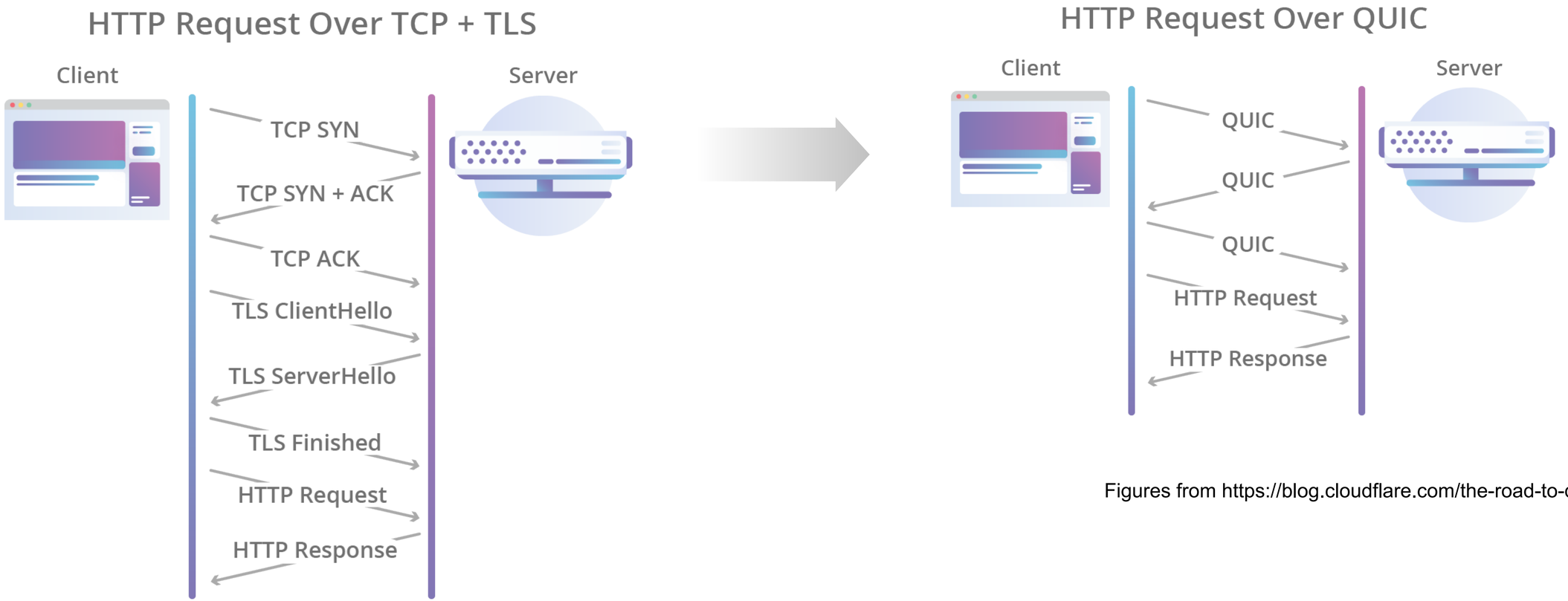
Figure from: A. Langley *et al.*, "The QUIC transport protocol: Design and Internet-scale deployment", Proceedings of the ACM SIGCOMM Conference, Los Angeles, August 2017. https://dl.acm.org/authorize.cfm?key=N33907

- QUIC replaces TCP, TLS, and parts of HTTP
  - HTTP → stream multiplexing
  - TLS → security
  - TCP → reliability, ordering, congestion control
  - Runs on UDP for ease of deployment
- QUIC is a general purpose client-server transport
  - Designed to run HTTP effectively

HTTP/3

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

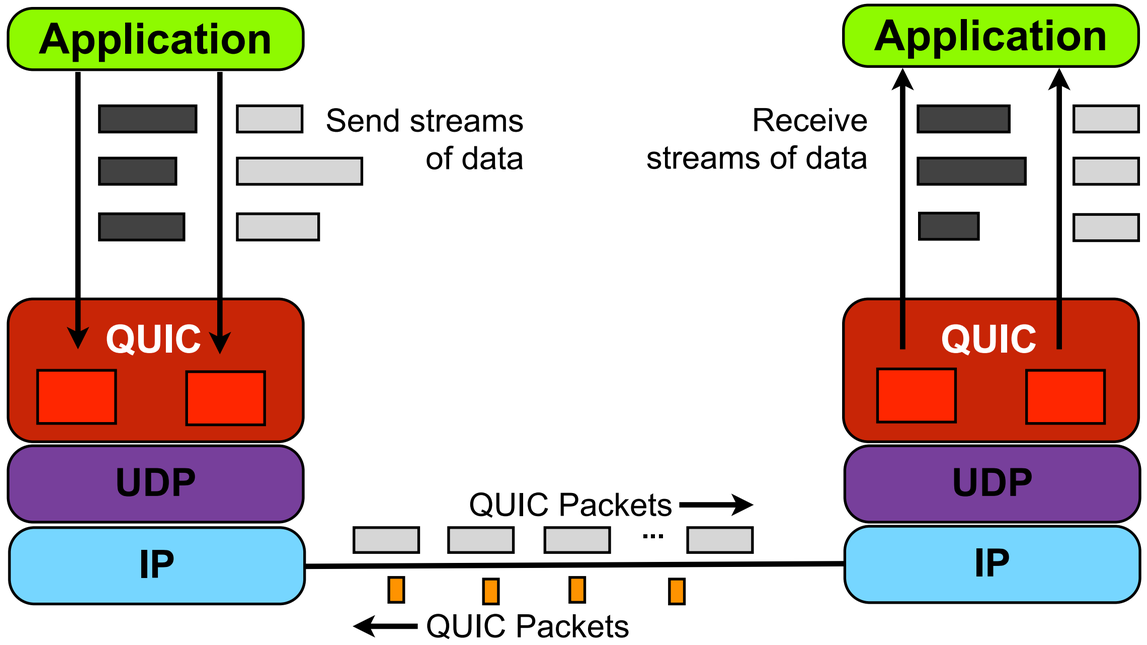
QUIC Overview (2/2)



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

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Packets, Headers, and Frames

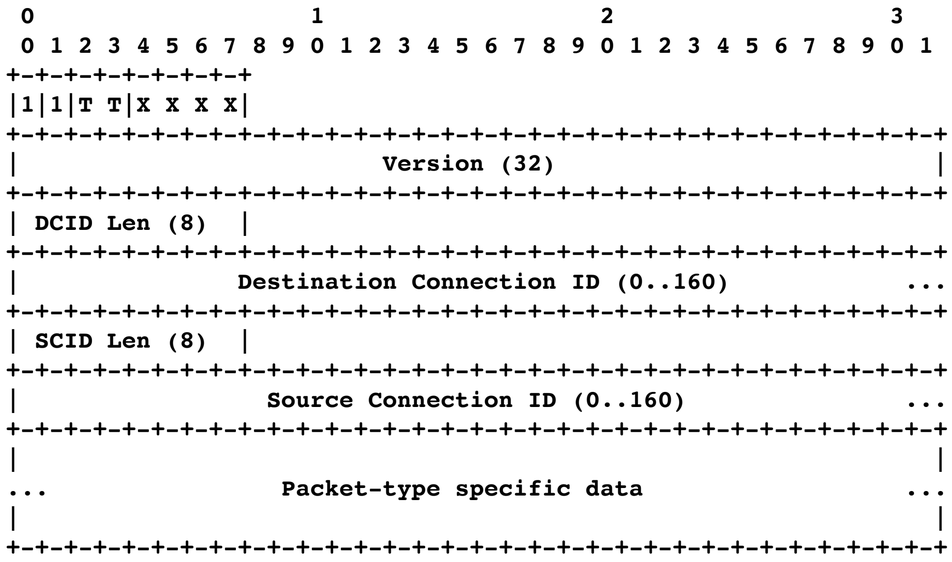


- QUIC sends and receives streams of data within a connection
  - Up to 2^{62} different streams in each direction in a single QUIC connection
- A connection comprises QUIC packets sent within UDP datagrams
- Each QUIC packet starts with a header, and contains one or more frames
  - Frames contain data, acknowledgements, or control messages

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Headers (1/2)

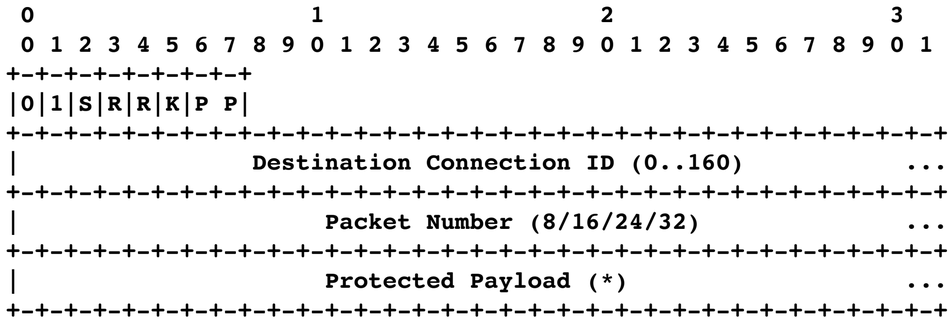
- QUIC packets can be long header packets or short header packets
- **Long header packets** are used to establish QUIC connections
  - They all start with a common header, followed by packet-type specific data
  - Four different long-header packet types, denoted by the TT field in the header:
    - **Initial** – initiates connection, starts TLS handshake
    - **0-RTT** – idempotent data sent with initial handshake, when resuming a session
    - **Handshake** – completes connection establishment
    - **Retry** – used to force address validation
- Several **Initial**, **0-RTT**, and **handshake** packets can be included in one UDP datagram, one after the other, followed by a short header packet



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Headers (2/2)

- QUIC packets can be long header packets or short header packets
- The is one **short header packet** defined in QUIC:
  - **1-RTT** – Used for all packets sent after the TLS handshake is complete
  - The short header is followed by QUIC frames in the enclosing UDP packet
  - Compared to long header, omits information that can be inferred from context



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Frames

- QUIC packet contain an encrypted sequence of **frames**
  - CRYPTO frames are used to carry TLS messages such as the **`ClientHello`**, **`ServerHello`** etc.
  - STREAM and ACK frames send data and acknowledgements
  - Migration between two network interfaces is supported by PATH_CHALLENGE and PATH_RESPONSE frames
  - Other frames control progress of a QUIC connection

- Compared to TCP, QUIC headers are limited in scope
  - Functionality provided by frames instead → more extensible
  - e.g., TCP sends sequence numbers and acknowledgements in the header; QUIC sends this information in STREAM and ACK frames

Type Value	Frame Type Name
0x00	PADDING
0x01	PING
0x02 - 0x03	ACK
0x04	RESET_STREAM
0x05	STOP_SENDING
0x06	CRYPTO
0x07	NEW_TOKEN
0x08 - 0x0f	STREAM
0x10	MAX_DATA
0x11	MAX_STREAM_DATA
0x12 - 0x13	MAX_STREAMS
0x14	DATA_BLOCKED
0x15	STREAM_DATA_BLOCKED
0x16 - 0x17	STREAMS_BLOCKED
0x18	NEW_CONNECTION_ID
0x19	RETIRE_CONNECTION_ID
0x1a	PATH_CHALLENGE
0x1b	PATH_RESPONSE
0x1c - 0x1d	CONNECTION_CLOSE
0x1e	HANDSHAKE_DONE

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

QUIC Transport Protocol

- Development and basic features
- **Connection establishment; data transfer**
- Avoiding ossification; limitations

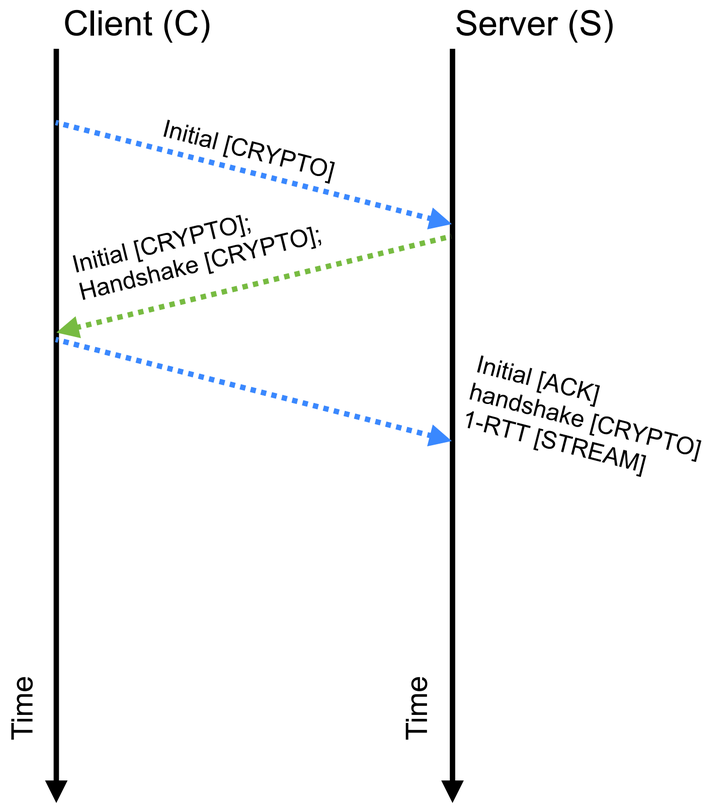
CC BY ND

QUIC Connection Establishment and Data Transfer

- A QUIC connection proceeds in two phases: handshake and data transfer
  - The **handshake** uses long header packets – establishes the connection, negotiates encryption keys, authenticates the server
  - The **data transfer** phase uses short header packets – sends data and acknowledgements after the connection is established

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

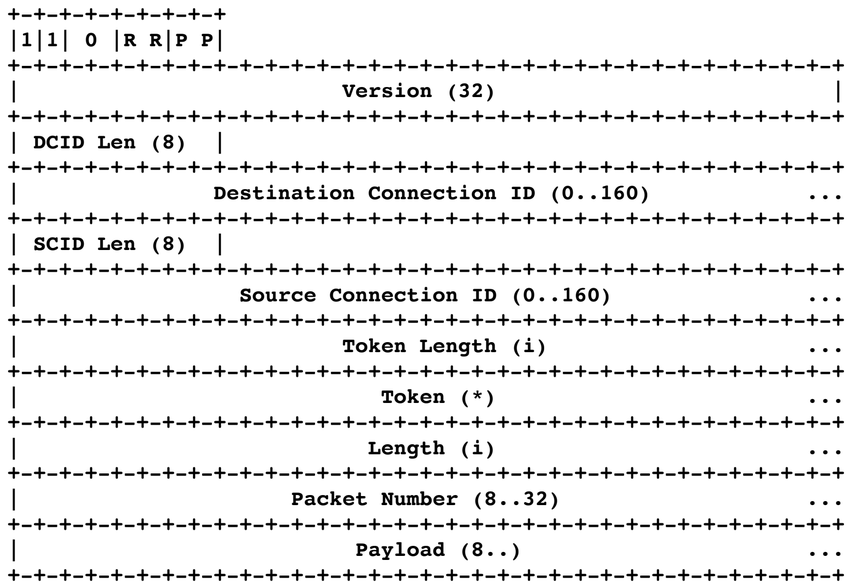
QUIC Connection Establishment (1/5)



- QUIC combines connection establishment and TLS handshake into one round-trip
  - C → S: QUIC **Initial** packet
    - **Initial** packet contains a CRYPTO frame that contains TLS **`ClientHello`**
  - S → C: QUIC **Initial** and **Handshake** packets
    - Both QUIC packets can be sent in a single UDP datagram
    - **Initial** packet contains CRYPTO frame that contains TLS **`ServerHello`**
    - **Handshake** packet contains other connection setup information
  - C → S: QUIC **Initial**, **Handshake**, and **1-RTT** packets
    - All three QUIC packets can be sent in a single UDP datagram
    - QUIC **Initial** packet contains ACK frame, acknowledging the server's Initial packet
    - QUIC **Handshake** packet contains CRYPTO frame that contains TLS **`Finished`**
    - QUIC **1-RTT** (short header) packet contains STREAM frame with initial data sent from client to server

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

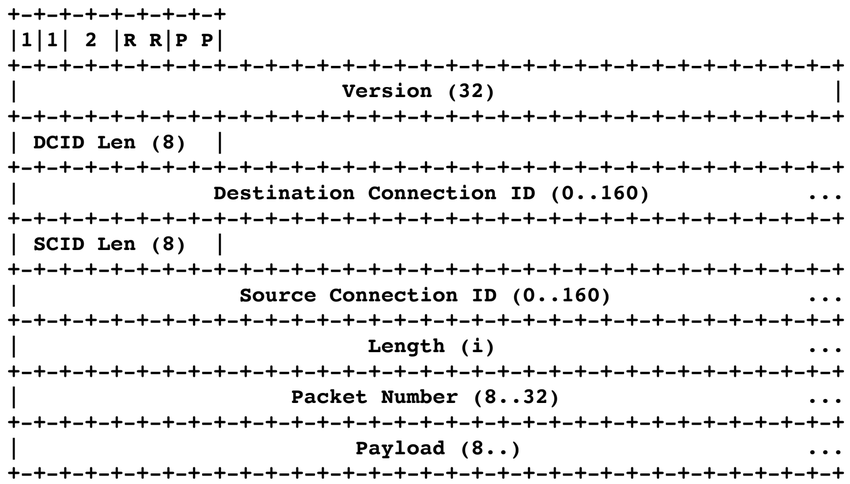
QUIC Connection Establishment (2/5)



- QUIC **Initial** packets play two main roles:
  - Synchronise client and server state – like TCP's **`SYN`** and **`SYN+ACK`** packets
  - Contains CRYPTO frame, with TLS **`ClientHello`** or **`Finished`**, and may also contain ACK frame
  - Combines connection setup and security negotiation in one packet
- QUIC **Initial** packets also carry optional **`Token`**
  - Server can refuse the initial connection attempt, and send a **Retry** packet containing a **`Token`**.
  - Client must then retry the connection, providing the **`Token`** in its **Initial** packet
  - Can be used to prevent connection spoofing

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

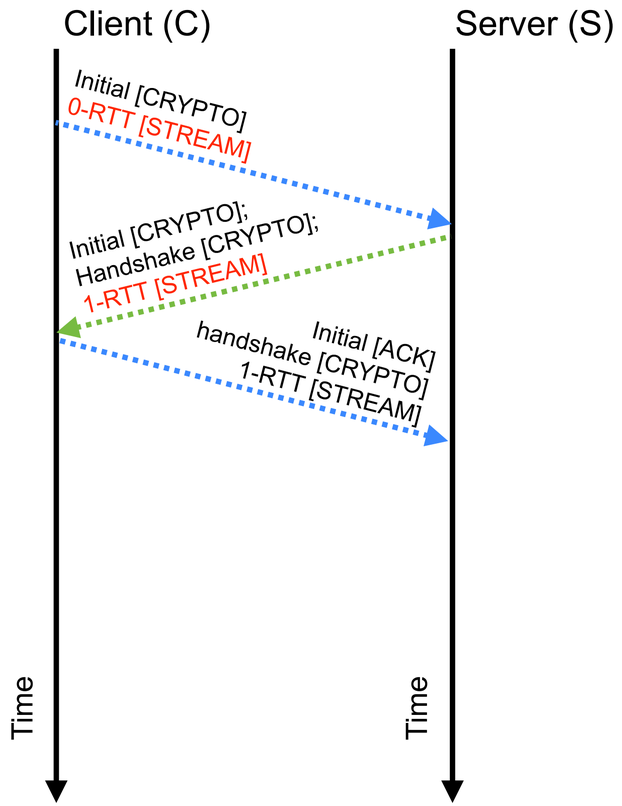
QUIC Connection Establishment (3/5)



- QUIC **Handshake** packets complete the TLS 1.3 exchange
  - The TLS **`ServerHello`** and **`Finished`** messages
  - TLS can also renegotiate new keys part-way through a connection – this is done in **`Handshake`** packets

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Connection Establishment (4/5)



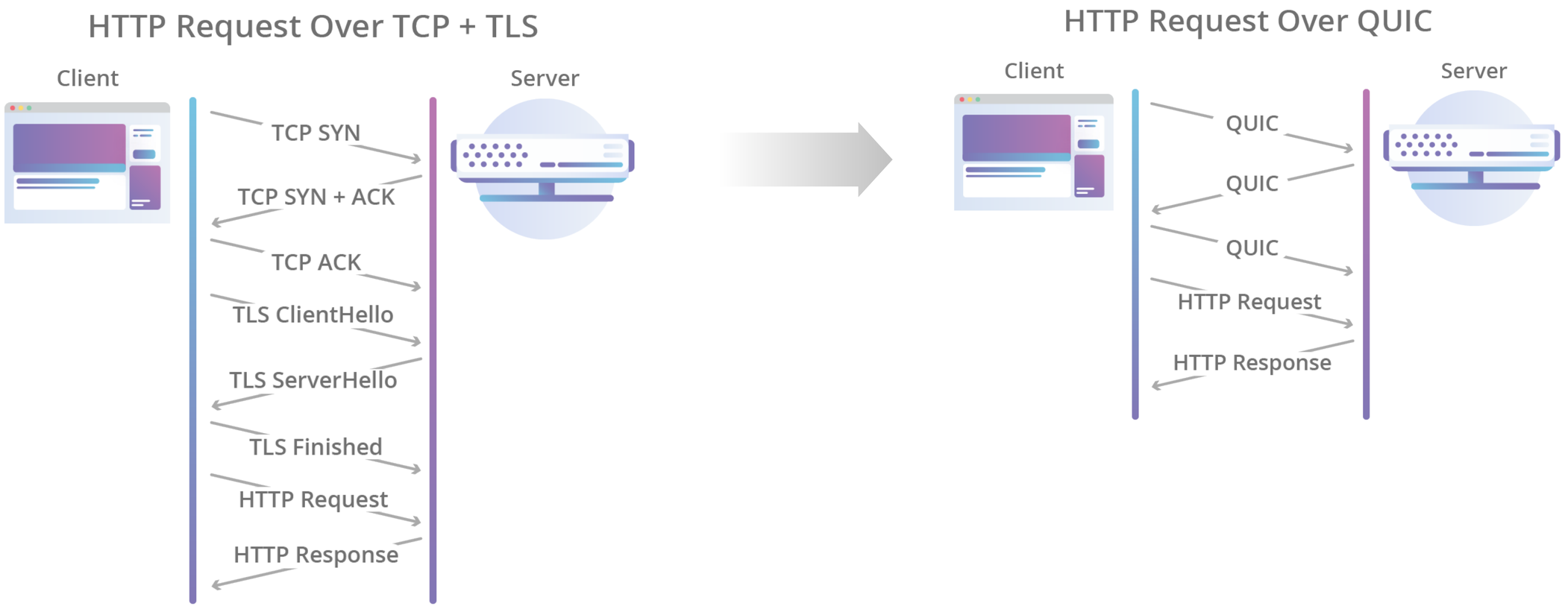
- QUIC supports TLS 0-RTT session re-establishment:
  - **Initial** packet contains CRYPTO frame with a TLS **`ClientHello`** and a **`SessionTicket`**
  - **0-RTT** packet included in the same UDP datagram contains a STREAM frame carrying idempotent 0-RTT data:

```
+-+-+-+-+-+-+-+-+
|1|1| 1 |R R|P P|
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         Version (32)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| DCID Len (8)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|               Destination Connection ID (0..160)            ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| SCID Len (8)  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                 Source Connection ID (0..160)               ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Length (i)                         ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    Packet Number (8..32)                    ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                          Payload (8..)                      ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  - Server responds with data in 1-RTT (short header) packet, along with its **Initial** and **handshake** packets

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Connection Establishment (5/5)

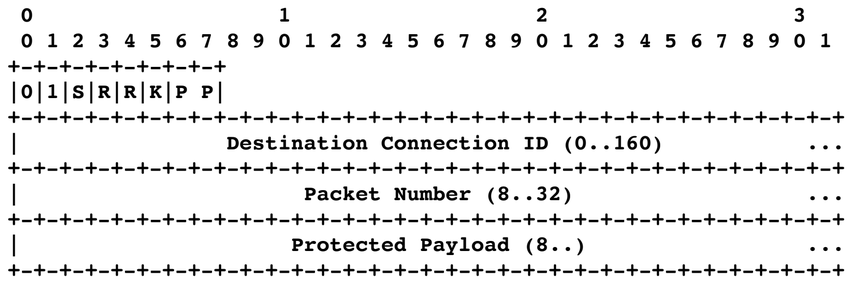


QUIC combines connection establishment and encryption key negotiation into a single handshake → TLS-over-TCP runs them sequentially

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Data Transfer, Streams, and Reliability (1/3)

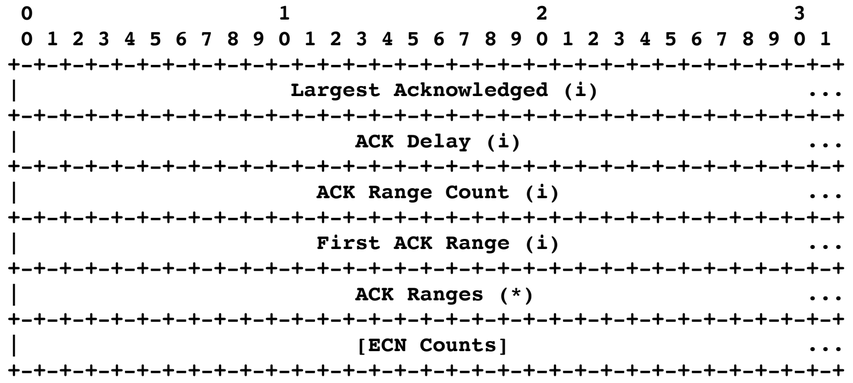
- After handshake has finished, QUIC switches to sending short header packets
  - The short header contains a **Packet Number** field
    - Packet numbers **increase by one** for each packet sent; ACK frames indicate received packet numbers
      - QUIC packet numbers count packets sent; TCP sequence numbers count bytes of data sent
    - QUIC **never** retransmits packets – retransmits frames sent in lost packets in new packets, with new packet numbers
      - TCP retransmits lost packet with original sequence number
  - Protected payload section of short header packets contains encrypted QUIC frames
    - STREAM frames contain data
    - ACK frames contain acknowledgements
  - Performs congestion control as in TCP → Lecture 6



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Data Transfer, Streams, and Reliability (2/3)

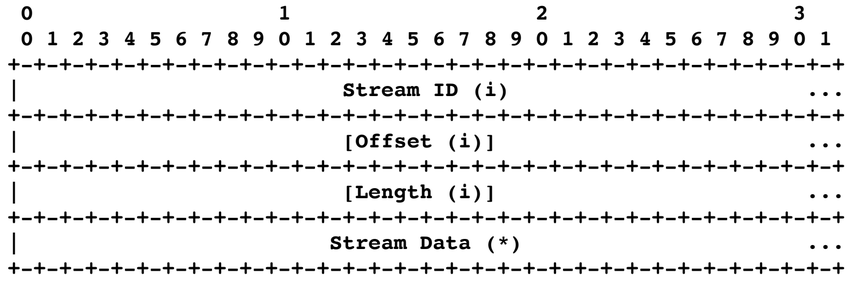
- QUIC sends acknowledgements of received packets in ACK frames
  - Sent **inside** a long- or short-header packets; unlike TCP, not part of headers
  - Indicate sequence numbers of QUIC **packets** that were received, not frames



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Data Transfer, Streams, and Reliability (3/3)

- Data is sent within STREAM frames, sent within QUIC packets
  - Contain a stream identifier, offset of the data within the stream, data length, and data
  - QUIC provides **multiple** reliable byte streams within a single connection
    - Data for each stream is delivered reliably and in-order
    - Order is not preserved between streams
    - Avoids head-of-line blocking between streams → Lecture 5
- Can view QUIC streams as multiple unframed byte streams sent within a single connection; alternatively can view each stream as framing a message, and the connection as a series of messages



Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

QUIC Transport Protocol

- Development and basic features
- Connection establishment; data transfer
- **Avoiding ossification; limitations**

CC BY ND

QUIC over UDP

- **Why run QUIC over UDP rather than over IP?**
- To ease end-system deployment in user-space applications
  - User-space applications, running over UDP, are easy to build
    - BSD sockets; portable → same API works everywhere
    - Widely understood programming model
    - No need for privileged access
  - No portable, unprivileged, interface to build applications that run directly over IP
    - Implementations have to run within the operating system kernel
    - Deploying kernel updates is difficult
    - Deploying application updates is straightforward
- To work around protocol ossification due to middleboxes
  - Firewalls block anything other than TCP and UDP



This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/COMST.2016.2626780, IEEE Communications Surveys & Tutorials

1

De-ossifying the Internet Transport Layer: A Survey and Future Perspectives

Giorgos Papastergiou, Gorry Fairhurst, David Ros, Anna Brunstrom, Karl-Johan Grinnemo, Per Hurtig, Naeem Khademi, Michael Tüxen, Michael Welzl, Dragana Damjanovic and Simone Mangiante

Abstract—**It is widely recognized that the Internet transport layer has become ossified, where further evolution has become hard or even impossible. This is a direct consequence of the ubiquitous deployment of middleboxes that hamper the deployment of new transports, aggravated further by the limited flexibility of the Application Programming Interface (API) typically presented to applications. To tackle this problem, a wide range of solutions have been proposed in the literature, each aiming to address a particular aspect. Yet, no single proposal has emerged that is able to enable evolution of the transport layer. In this work, after an overview of the main issues and reasons for transport-layer ossification, we survey proposed solutions and discuss their potential and limitations. The survey is divided into five parts, each covering a set of point solutions for a different facet of the problem space: 1) designing middlebox-proof transports, 2) signaling for facilitating middlebox traversal, 3) enhancing the API between the applications and the transport layer, 4) discovering and exploiting end-to-end capabilities, and 5) enabling user-space protocol stacks. Based on this analysis, we then identify further development needs towards an overall solution. We argue that the development of a comprehensive transport layer framework, able to facilitate the integration and cooperation of specialized solutions in an application-independent and flexible way, is a necessary step toward making the Internet transport architecture truly evolvable. To this end, we identify the requirements for such a framework and provide insights for its development.**

Index Terms—**Transport protocols, protocol-stack ossification, API, middleboxes, user-space networking stacks.**

I. Introduction and Background

Networks can and do vary significantly in the set of functions they offer and their ability to move data between endpoints. The transport layer operates across the network and is responsible for efficient and robust end-to-end communication between network endpoints. The term end-to-end is often associated with a principle, called the end-to-end argument [1]. This suggests that "functions placed at low levels of a system may be redundant, or of little value, when compared with the cost of providing them at that low level." This argument followed Schroeder et al.'s earlier work on system design and security [2], and is now generally considered as a simple guide for which services should be realised at the transport layer.

The transport layer was designed to hide the details and variability of the network service from the applications that need to use it. The Internet's transport layer also contains other functions that are difficult or impossible to provide within a network, such as reliability, verification of delivery, flow control to prevent the application from overwhelming the remote endpoint, congestion control to prevent the application from overwhelming the network, etc. People using the Internet mostly run applications that are based on the Transmission Control Protocol, TCP [3], which provides these transport functions.

Some applications need a different set of services to those offered by TCP. For example, a web client may wish to be able to prioritize sub-flows carrying specific objects, a multimedia flow may prefer timeliness to reliable delivery, and IP telephony can be tolerant to packet loss or in some cases to bit errors. There are many cases where TCP simply does not meet the need of applications—yet it ends up being used because it "just works," but not necessarily very well [4]. Applications that do not want the transport semantics of TCP typically just use the User Datagram Protocol, UDP [5]. While UDP provides flexibility that allows any set of services to be defined, every function needed has to be implemented at the application layer.

Some initiatives have developed alternate protocols to TCP, suited for other application types, for instance: the Datagram Congestion Control Protocol (DCCP) [6] was proposed to support streaming multimedia; the Stream Control Transport Protocol (SCTP) [7], [8] originally targeted telephony signaling; UDP-Lite [9] supports error-tolerant audio and video services over wireless links. However, despite being standardized, with available implementations for common platforms, these transports are seldom seen in the general Internet, and TCP and UDP remain the only widely used transports.

A. Transport-layer ossification: overview of issues

Why do developers and users not adopt more modern protocols? It is not because new transports do not meet a real need. The following paragraphs examine the main reasons for this *ossification* of the transport layer.

G. Papastergiou and D. Ros are with Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway. E-mail: {gpapastr,dros}@simula.no.
G. Fairhurst is with University of Aberdeen, AB24 3UE Aberdeen, United Kingdom. E-mail: gorry@erg.abdn.ac.uk.
A. Brunstrom, K.-J. Grinnemo and P. Hurtig are with Karlstad University, Universitetsgatan 2, 651 88 Karlstad, Sweden. E-mail: {anna.brunstrom,karl-johan.grinnemo,per.hurtig}@kau.se.
N. Khademi and M. Welzl are with University of Oslo, P.O. Box 1072 Blindern, 0316 Oslo, Norway. E-mail: {naeemk,michawe}@ifi.uio.no.
M. Tüxen is with Münster University of Applied Sciences, Binnenkreuzstraße 11, 48565 Steinfurt, Germany. E-mail: tuexen@fh-muenster.de.
D. Damjanovic is with Mozilla Corporation. E-mail: ddamjanovic@mozilla.com.
S. Mangiante is with EMC Corporation, Ovens, Co. Cork, Ireland. E-mail: simone.mangiante@emc.com.

1553-877X (c) 2016 IEEE. Translations and content mining are permitted for academic research only. Personal use is also permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

G. Papastergiou *et al*, "De-ossifying the Internet transport layer: A survey and future perspectives", IEEE Communications Surveys and Tutorials (Nov. 2016). https://dx.doi.org/10.1109/COMST.2016.2626780/

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Ossification (1/2)

- Deployment experience → if a field in a protocol is visible to the network, someone will implement a middlebox that relies on its presence
- Once a protocol has been widely deployed, very hard to change



Is it Still Possible to Extend TCP?

Michio Honda[+], Yoshifumi Nishida[+], Costin Raiciu[†], Adam Greenhalgh[‡], Mark Handley[‡], Hideyuki Tokuda[+]
Keio University[+], Universitatea Politehnica Bucuresti[†], University College London[‡]
{micchie,nishida}@sfc.wide.ad.jp, costin.raiciu@cs.pub.ro
{a.greenhalgh,m.handley}@cs.ucl.ac.uk, hxt@ht.sfc.keio.ac.jp

ABSTRACT

We've known for a while that the Internet has ossified as a result of the race to optimize existing applications or enhance security. NATs, performance-enhancing-proxies, firewalls and traffic normalizers are only a few of the middleboxes that are deployed in the network and look beyond the IP header to do their job. IP itself can't be extended because "IP options are not an option" [10]. Is the same true for TCP?

In this paper we develop a measurement methodology for evaluating middlebox behavior relating to TCP extensions and present the results of measurements conducted from multiple vantage points. The short answer is that we can still extend TCP, but extensions' design is very constrained as it needs to take into account prevalent middlebox behaviors. For instance, absolute sequence numbers cannot be embedded in options, as middleboxes can rewrite ISN and preserve undefined options. Sequence numbering also must be consistent for a TCP connection, because many middleboxes only allow through contiguous flows.

We used these findings to analyze three proposed extensions to TCP. We find that MPTCP is likely to work correctly in the Internet or fallback to regular TCP. TcpCrypt seems ready to be deployed, however it is fragile if resegmentation does happen—for instance with hardware offload. Finally, TCP extended options in its current form is not safe to deploy.

Categories and Subject Descriptors

C.2.2 [**Computer-communication Networks**]: Network Protocols; C.2.6 [**Computer-communication Networks**]: Internetworking

General Terms

Measurement, Design, Experimentation, Standardization

Keywords

Middleboxes, Measurements, TCP, Protocol design

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IMC'11, November 2–4, 2011, Berlin, Germany.
Copyright 2011 ACM 978-1-4503-1013-0/11/11 ...$10.00.

M. Honda *et al.,* "Is it still possible to extend TCP?", In Proc. ACM Internet Measurement Conference, Berlin, Nov. 2011. https://dx.doi.org/10.1145/2068816.2068834

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Ossification (2/2)

- Protocol ossification affected the design of:
  - TLS 1.3
  - Multipath TCP
  - TCP Fast Open
  - TCP Selective Acknowledgements
  - …
- Increasingly viewed as a problem in the standards community
  - Difficult to evolve network protocols to address new requirements
  - A system that can no longer evolve and change will die

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Avoiding Ossification in QUIC

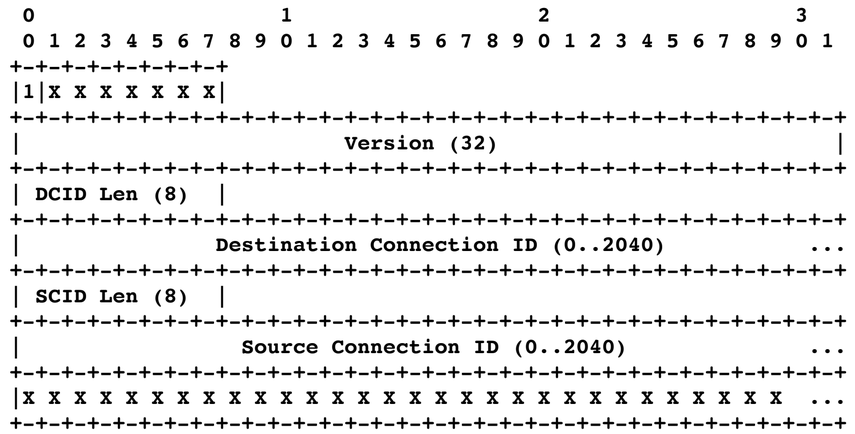
- Three tools to prevent ossification of QUIC:
  - Published protocol invariants
  - Pervasive encryption of transport headers
  - GREASE

- QUIC is a new design – want to **avoid ossification**, starting with initial deployments
- Design the protocol to make it difficult, ideally impossible, for middleboxes to interfere with QUIC connections

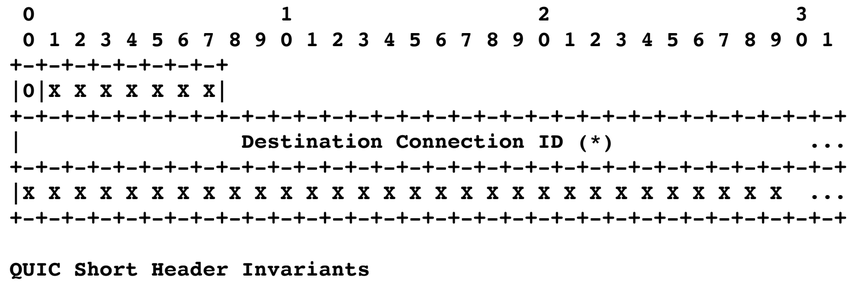
Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Invariants

- Properties of QUIC that will remain unchanged as new versions of the protocol of developed
  - Explicit guidance to middlebox designers: can assume these properties of QUIC will not change
  - The IETF **will** change other fields or properties between QUIC versions
- What invariants are guaranteed?
  - Packets start with a long header or a short header
    - The first bit of long header packets is always 1; they include version number, destination- and source-connection identifiers
    - The first bit of short header packets is always 0; they include a destination connection identifier
  - Connections can arbitrarily switch between long header and short header packets



QUIC Long Header Invariants



QUIC Short Header Invariants

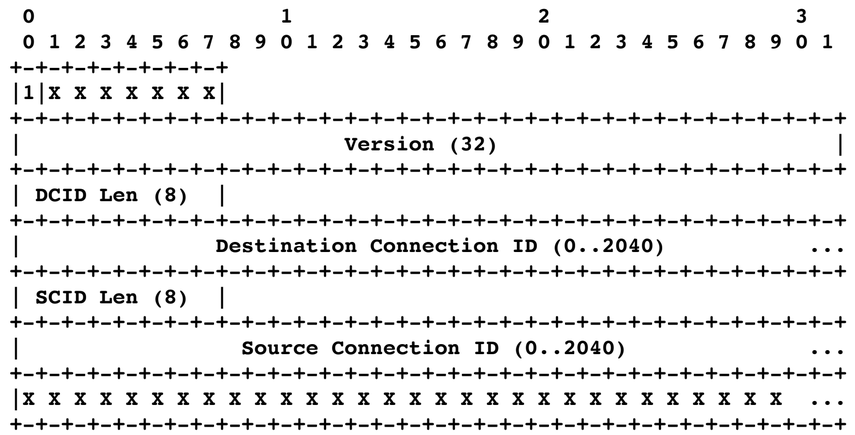
X = bit may take arbitrary value

https://datatracker.ietf.org/doc/draft-ietf-quic-invariants/

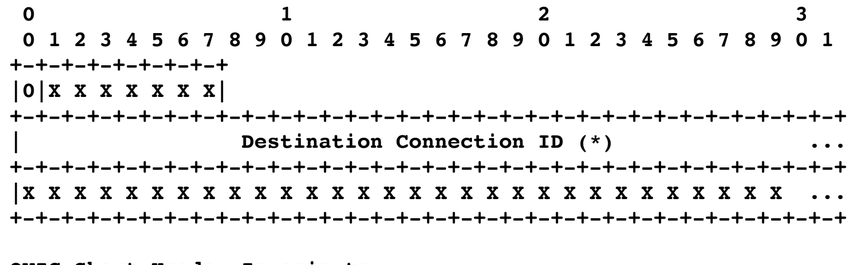
Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Avoiding Ossification via Pervasive Encryption

- QUIC encrypts as much data as possible
  - Entire packet **except** invariant fields and the last 7-bits of the first byte is encrypted; entire packet is authenticated
  - Encryption keys for initial handshake packets are derived from connection identifiers
    - Contain **`ClientHello`** and **`ServerHello`**, which are unencrypted when using TLS over TCP, and **don't need to be encrypted**
    - QUIC provides no more security than TLS over TCP, but makes it **expensive** for middleboxes to read handshake
  - Rest of QUIC connection protected by TLS 1.3 as normal
- QUIC authenticates **all** data
  - If a middlebox changes the headers, the change can be detected



QUIC Long Header Invariants



QUIC Short Header Invariants

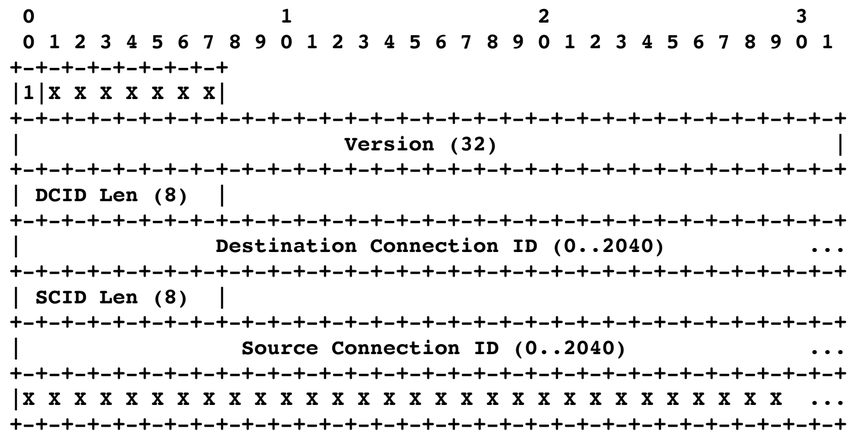
X = bit may take arbitrary value

https://datatracker.ietf.org/doc/draft-ietf-quic-invariants/

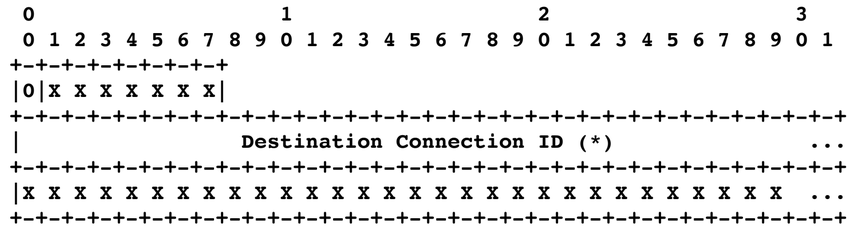
Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

Avoiding Ossification via GREASE

- QUIC makes extensive use of GREASE
  - Every field encrypted or has a value that is unpredictable
    - New connections use randomly chosen connection identifies
    - Clients randomly try to negotiate new version numbers
      - Version numbers matching **`0x?a?a?a?a`** for testing version negotiation – will be rejected by servers
    - Unused header fields given randomly chosen values
  - Goal is that middleboxes can't make any assumptions about QUIC header values → nothing in the header is predictable
  - Hopefully avoids ossification → nothing to ossify around



QUIC Long Header Invariants



QUIC Short Header Invariants

X = bit may take arbitrary value

https://datatracker.ietf.org/doc/draft-ietf-quic-invariants/

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

QUIC Benefits and Costs

- **Why is QUIC desirable?**
  - Reduces secure connection establishment latency
  - Reduces risk of ossification; easy to deploy
  - Supports multiple streams within a single connection
- **Why is QUIC problematic?**
  - Libraries and support new, poorly documented, and frequently buggy
  - CPU usage is high compared to TLS-over-TCP
    - TCP stack currently much better optimised
    - TCP and TLS often has hardware offload; QUIC doesn't yet
  - These issues will be resolved – but it will take some years before QUIC is as stable and performant as TLS-over-TCP
- **TCP lasted 40 years – QUIC is a similarly long-term project, that's only just reached version 1.0**

Colin Perkins | https://csperkins.org/ | Copyright © 2026 University of Glasgow

University of Glasgow

Improving Secure Connection Establishment

- Limitations of TLS 1.3
- QUIC transport protocol
- TCP has dominated for 40 years – is QUIC the future?

CC BY ND