OST
Eastern Switzerland
University of Applied Sciences

qh:// - The Quite Ok HTTP Protocol

SEMESTER THESIS

Ivan Knoefler

Gian Hunold

Fall Semester 2025

Department of Computer Science

OST – University of Applied Sciences

Campus Rapperswil-Jona

Supervising examiner: Prof. Dr. Thomas Bocek

Submitted: 18. December 2025

Abstract

Modern web protocols like HTTP/2 and HTTP/3 reduce bandwidth overhead through dynamic header compression (HPACK, QPACK), which requires complex state synchronization while increasing CPU overhead and implementation difficulty. This raises the question whether protocol complexity is necessary or if a simpler approach can achieve performance within 10-15% for typical web workloads.

This thesis presents QH (Quite OK HTTP), a deliberately simplified web protocol that replaces dynamic compression with a static header table based on empirical analysis of real-world header frequencies. QH uses binary message encoding with varint length prefixes and runs on QOTP, a UDP-based transport providing 0-RTT connections and mandatory encryption. The protocol omits dynamic compression, server push, and other complex HTTP/2 and HTTP/3 features to prioritize implementation simplicity.

We implemented QH in Go with complete client-server support and evaluated wire format efficiency against HTTP/1.1, HTTP/2, and HTTP/3 using 110 test cases covering edge cases and real production traffic. Results show QH achieves 38.8% smaller wire format than HTTP/1.1 while remaining within 7-11% of HTTP/2 and HTTP/3 despite lacking dynamic compression, with optimal performance on request headers and small API payloads.

The evaluation quantifies the cost of simplicity: a 7-11% wire format overhead compared to dynamic compression in exchange for eliminating state synchronization, enabling stateless debugging, and reducing implementation complexity. This demonstrates that protocol simplicity and performance are not mutually exclusive for common web applications, providing guidance for future protocol design decisions.

Management Summary

Initial Situation

Modern web protocols have evolved toward increasing complexity. HTTP/2 and HTTP/3 employ sophisticated dynamic header compression algorithms (HPACK, QPACK) that require production implementations spanning tens of thousands of lines of code. While achieving high data efficiency, this complexity introduces development and maintenance costs:

- Synchronization of dynamic compression state between client and server
- Extensive logging infrastructure required for debugging
- High entry barriers for new implementations

Central Question: Can a deliberately simplified protocol achieve comparable performance with acceptable overhead (10-15%) while reducing implementation complexity?

Approach and Technologies

We developed QH (Quite Ok HTTP), a minimalist web protocol designed to quantify the cost-benefit trade-off between simplicity and efficiency.

Key Design Decisions:

- **Static header table:** Fixed compression table based on real traffic analysis instead of dynamic compression
- **Binary format:** Varint encoding for length prefixes
- **Simplified transport:** QOTP layer providing 0-RTT connections, integrated encryption, and DNS-based key distribution

Methodology:

- Browser extension (Chrome/Firefox) to collect header frequency data from real web traffic
- Complete Go implementation with client, server, and Wireshark dissector
- Evaluation using 110 test cases (10 edge cases, 100 real HTTP requests) comparing wire format sizes against HTTP/1.1, HTTP/2, and HTTP/3

Results

The evaluation demonstrates that QH achieves its design goals:

Efficiency Results:

- 38.8% smaller than HTTP/1.1
- 7-11% larger than HTTP/2 and HTTP/3

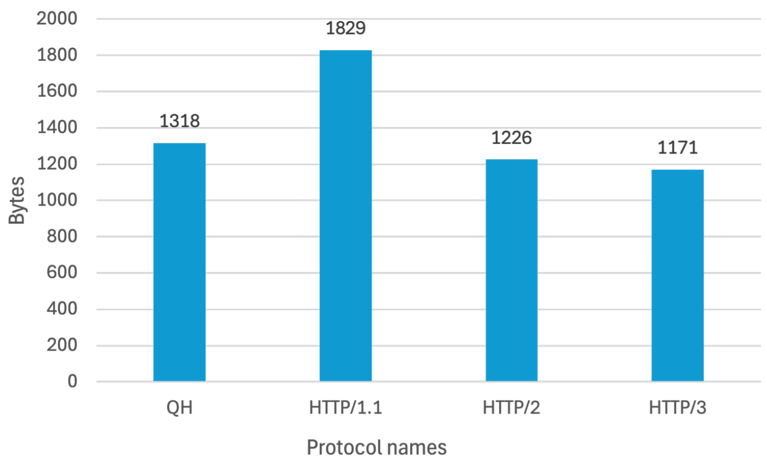


Figure 1: Average wire format bytes across protocols

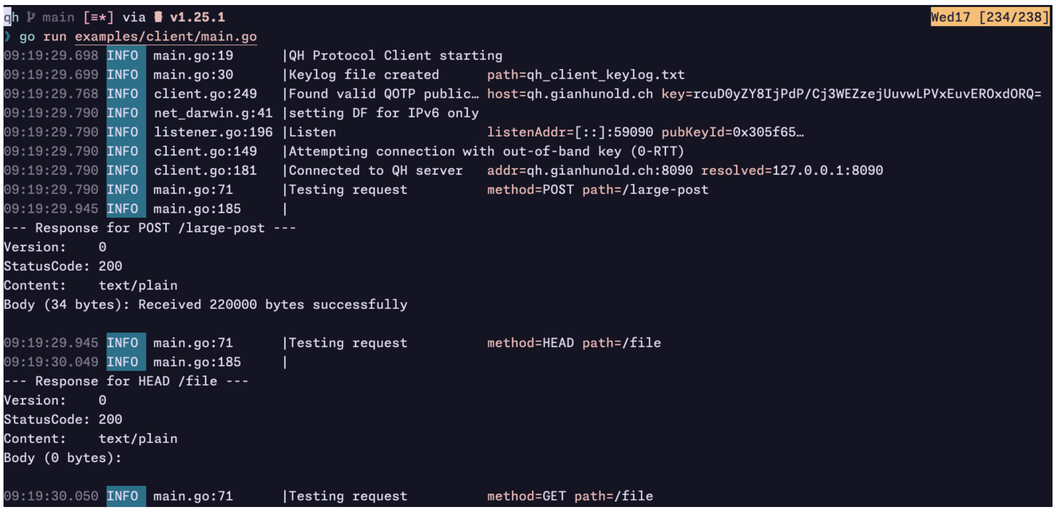
Business Value:

This work quantifies for the first time the cost of protocol simplicity: 7-11% wire format overhead in exchange for:

- **Simpler implementation:** No dynamic compression state or synchronization required
- **Lower maintenance complexity:** Deterministic encoding eliminates state-related bugs
- **Simplified debugging:** Protocol analysis via Wireshark without extensive logging infrastructure

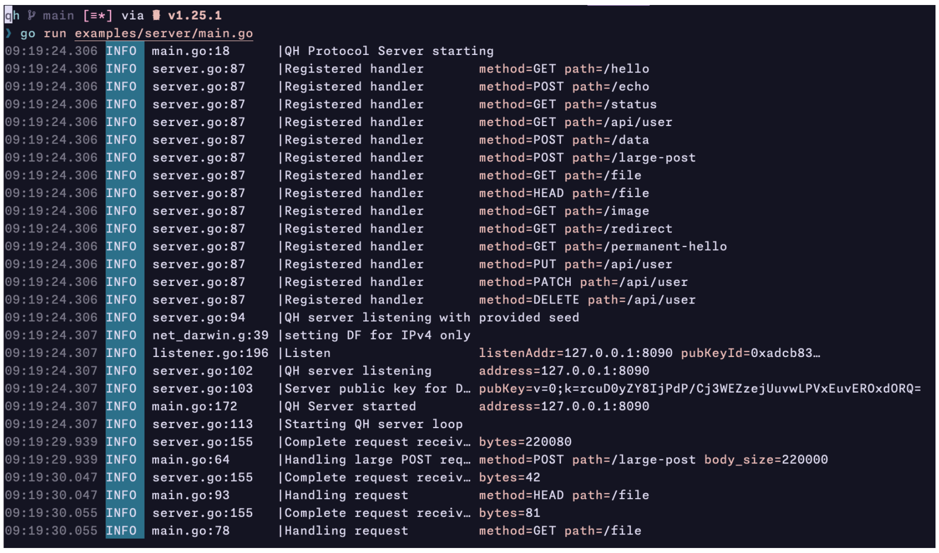
Outlook:

QH demonstrates that protocol simplicity and performance are not mutually exclusive for typical web applications. The 7-11% overhead represents an acceptable trade-off for scenarios where implementation simplicity is valued over maximum compression, such as microservices, IoT systems, and internal APIs. Future work could explore client library development, browser integration, and real-world deployment studies to validate this approach in production environments.



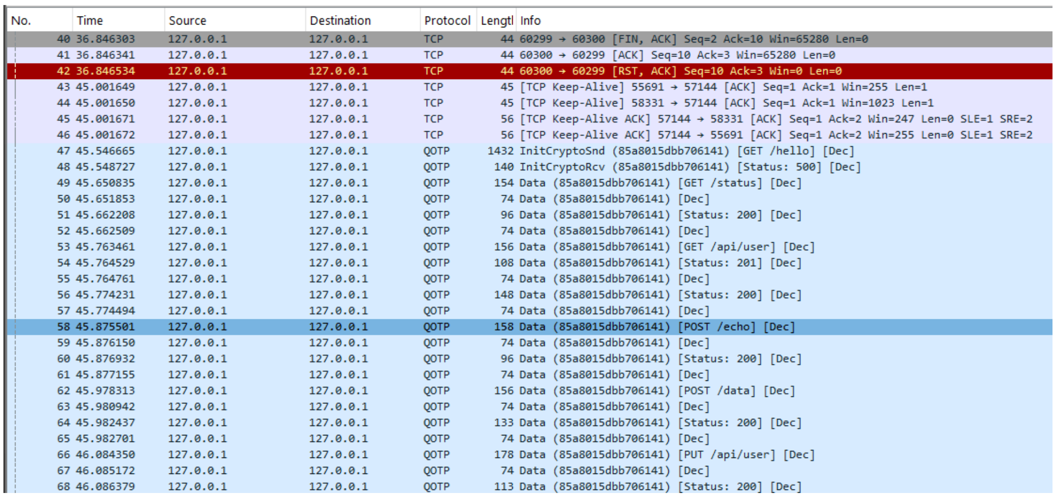
```
qh  main [≡*] via  v1.25.1                                                                                       Wed17 [234/238]
❯ go run examples/client/main.go
09:19:29.698 INFO  main.go:19       |QH Protocol Client starting
09:19:29.699 INFO  main.go:30       |Keylog file created      path=qh_client_keylog.txt
09:19:29.768 INFO  client.go:249    |Found valid QOTP public… host=qh.gianhunold.ch key=rcuD0yZY8IjPdP/Cj3WEZzejUuvwLPVxEuvEROxdORQ=
09:19:29.790 INFO  net_darwin.g:41  |setting DF for IPv6 only
09:19:29.790 INFO  listener.go:196  |Listen                   listenAddr=[::]:59090 pubKeyId=0x305f65…
09:19:29.790 INFO  client.go:149    |Attempting connection with out-of-band key (0-RTT)
09:19:29.790 INFO  client.go:181    |Connected to QH server   addr=qh.gianhunold.ch:8090 resolved=127.0.0.1:8090
09:19:29.790 INFO  main.go:71       |Testing request          method=POST path=/large-post
09:19:29.945 INFO  main.go:185      |
--- Response for POST /large-post ---
Version:    0
StatusCode: 200
Content:    text/plain
Body (34 bytes): Received 220000 bytes successfully

09:19:29.945 INFO  main.go:71       |Testing request          method=HEAD path=/file
09:19:30.049 INFO  main.go:185      |
--- Response for HEAD /file ---
Version:    0
StatusCode: 200
Content:    text/plain
Body (0 bytes):

09:19:30.050 INFO  main.go:71       |Testing request          method=GET path=/file
```

Figure 2: QH example client CLI



```
qh  main [≡*] via  v1.25.1
❯ go run examples/server/main.go
09:19:24.306 INFO  main.go:18       |QH Protocol Server starting
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/hello
09:19:24.306 INFO  server.go:87     |Registered handler       method=POST path=/echo
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/status
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/api/user
09:19:24.306 INFO  server.go:87     |Registered handler       method=POST path=/data
09:19:24.306 INFO  server.go:87     |Registered handler       method=POST path=/large-post
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/file
09:19:24.306 INFO  server.go:87     |Registered handler       method=HEAD path=/file
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/image
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/redirect
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/permanent-hello
09:19:24.306 INFO  server.go:87     |Registered handler       method=PUT path=/api/user
09:19:24.306 INFO  server.go:87     |Registered handler       method=PATCH path=/api/user
09:19:24.306 INFO  server.go:87     |Registered handler       method=DELETE path=/api/user
09:19:24.306 INFO  server.go:94     |QH server listening with provided seed
09:19:24.307 INFO  net_darwin.g:39  |setting DF for IPv4 only
09:19:24.307 INFO  listener.go:196  |Listen                   listenAddr=127.0.0.1:8090 pubKeyId=0xadcb83…
09:19:24.307 INFO  server.go:102    |QH server listening      address=127.0.0.1:8090
09:19:24.307 INFO  server.go:103    |Server public key for D… pubKey=v=0;k=rcuD0yZY8IjPdP/Cj3WEZzejUuvwLPVxEuvEROxdORQ=
09:19:24.307 INFO  main.go:172      |QH Server started        address=127.0.0.1:8090
09:19:24.307 INFO  server.go:113    |Starting QH server loop
09:19:29.939 INFO  server.go:155    |Complete request receiv… bytes=220080
09:19:29.939 INFO  main.go:64       |Handling large POST req… method=POST path=/large-post body_size=220000
09:19:30.047 INFO  server.go:155    |Complete request receiv… bytes=42
09:19:30.047 INFO  main.go:93       |Handling request         method=HEAD path=/file
09:19:30.055 INFO  server.go:155    |Complete request receiv… bytes=81
09:19:30.055 INFO  main.go:78       |Handling request         method=GET path=/file
```

Figure 3: QH example server CLI



No.	Time	Source	Destination	Protocol	Lengtl	Info
40	36.846303	127.0.0.1	127.0.0.1	TCP	44	60299 → 60300 [FIN, ACK] Seq=2 Ack=10 Win=65280 Len=0
41	36.846341	127.0.0.1	127.0.0.1	TCP	44	60300 → 60299 [ACK] Seq=10 Ack=3 Win=65280 Len=0
42	36.846534	127.0.0.1	127.0.0.1	TCP	44	60300 → 60299 [RST, ACK] Seq=10 Ack=3 Win=0 Len=0
43	45.001649	127.0.0.1	127.0.0.1	TCP	45	[TCP Keep-Alive] 55691 → 57144 [ACK] Seq=1 Ack=1 Win=255 Len=1
44	45.001650	127.0.0.1	127.0.0.1	TCP	45	[TCP Keep-Alive] 58331 → 57144 [ACK] Seq=1 Ack=1 Win=1023 Len=1
45	45.001671	127.0.0.1	127.0.0.1	TCP	56	[TCP Keep-Alive ACK] 57144 → 58331 [ACK] Seq=1 Ack=2 Win=247 Len=0 SLE=1 SRE=2
46	45.001672	127.0.0.1	127.0.0.1	TCP	56	[TCP Keep-Alive ACK] 57144 → 55691 [ACK] Seq=1 Ack=2 Win=255 Len=0 SLE=1 SRE=2
47	45.546665	127.0.0.1	127.0.0.1	QOTP	1432	InitCryptoSnd (85a8015dbb706141) [GET /hello] [Dec]
48	45.548727	127.0.0.1	127.0.0.1	QOTP	140	InitCryptoRcv (85a8015dbb706141) [Status: 500] [Dec]
49	45.650835	127.0.0.1	127.0.0.1	QOTP	154	Data (85a8015dbb706141) [GET /status] [Dec]
50	45.651853	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
51	45.662208	127.0.0.1	127.0.0.1	QOTP	96	Data (85a8015dbb706141) [Status: 200] [Dec]
52	45.662509	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
53	45.763461	127.0.0.1	127.0.0.1	QOTP	156	Data (85a8015dbb706141) [GET /api/user] [Dec]
54	45.764529	127.0.0.1	127.0.0.1	QOTP	108	Data (85a8015dbb706141) [Status: 201] [Dec]
55	45.764761	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
56	45.774231	127.0.0.1	127.0.0.1	QOTP	148	Data (85a8015dbb706141) [Status: 200] [Dec]
57	45.774494	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
58	45.875501	127.0.0.1	127.0.0.1	QOTP	158	Data (85a8015dbb706141) [POST /echo] [Dec]
59	45.876150	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
60	45.876932	127.0.0.1	127.0.0.1	QOTP	96	Data (85a8015dbb706141) [Status: 200] [Dec]
61	45.877155	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
62	45.978313	127.0.0.1	127.0.0.1	QOTP	156	Data (85a8015dbb706141) [POST /data] [Dec]
63	45.980942	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
64	45.982437	127.0.0.1	127.0.0.1	QOTP	133	Data (85a8015dbb706141) [Status: 200] [Dec]
65	45.982701	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
66	46.084350	127.0.0.1	127.0.0.1	QOTP	178	Data (85a8015dbb706141) [PUT /api/user] [Dec]
67	46.085172	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
68	46.086379	127.0.0.1	127.0.0.1	QOTP	113	Data (85a8015dbb706141) [Status: 200] [Dec]

Figure 4: Wireshark dissector showing decrypted QOTP traffic for protocol analysis

Contents

List of Figures

List of Tables

1 Introduction

1.1 Context and Motivation

The World Wide Web relies on Hypertext Transfer Protocol (HTTP) as its foundational application protocol. First introduced in 1991, HTTP has undergone significant evolution to meet the demands of modern web applications. HTTP/1.1 [1] introduced persistent connections and chunked transfer encoding. HTTP/2 [2] added binary framing, header compression through HPACK [3], and stream multiplexing over a single TCP (Transmission Control Protocol) connection. HTTP/3 [4] replaced TCP with QUIC [5] and introduced QPACK [6] header compression.

However, this evolution has come at a cost: increasing protocol complexity. HTTP/2′s HPACK compression requires maintaining dynamic compression tables synchronized between client and server, with state that persists across the connection lifetime. HTTP/3′s QPACK further complicates this by introducing dedicated encoder and decoder streams to manage compression state independently from data streams. The QUIC transport layer underlying HTTP/3 implements sophisticated congestion control, loss recovery, and connection migration mechanisms. This complexity manifests in multiple dimensions: production implementations require substantial codebases, extensive dependency chains (TLS (Transport Layer Security) libraries, compression frameworks, certificate validation via Certificate Authorities, OCSP (Online Certificate Status Protocol) responders, Let's Encrypt integration), complex state machines (dynamic compression tables, flow control, congestion control), and challenging debugging experiences where stateful compression and encryption prevent standalone message inspection.

This thesis draws inspiration from the "Quite Ok" design philosophy exemplified by [7], which achieves PNG-comparable compression with just 300 lines of code. While protocols like QUIC and TLS 1.3 span dozens of RFCs and require thousands of lines of implementation code to achieve maximum optimization, the "Quite Ok" approach prioritizes understandability and implementability. The goal is not the most efficient system, but a simpler one with comparable practical performance.

This complexity raises a fundamental question: Is this sophistication necessary to achieve good web protocol performance, or can a deliberately simplified design deliver results within 10-15% of HTTP/2 and HTTP/3 for typical workloads? This thesis explores this question through the design, implementation, and evaluation of Quite Ok HTTP Protocol (QH), a web protocol that prioritizes simplicity while targeting this performance goal.

1.2 Research Questions

This thesis addresses two research questions:

RQ1: Protocol Simplicity and Performance

Can a web protocol designed for simplicity achieve performance within 10-15% of HTTP/2 and HTTP/3 for typical web workloads?

RQ2: Static Header Compression

Can static header compression match the wire efficiency of dynamic compression schemes (HPACK/QPACK)? What is the trade-off between compression ratio and implementation simplicity?

RQ3: Debugging and Analysis

Can standard network analysis tools like Wireshark provide sufficient protocol visibility, reducing reliance on protocol implementation logging for debugging?

1.3 Contributions

This thesis makes the following contributions:

1. **QH Protocol Design:** A simplified HTTP-like application protocol built on QOTP (Quite Ok Transport Protocol), built for reduced complexity.
2. **Static Header Compression Scheme:** A static header table design based on empirical header frequency analysis.
3. **Complete Implementation and Development Tooling:** A Go implementation of QH including client, server, and a Wireshark dissector for protocol analysis.
4. **Performance Evaluation:** Benchmark suite comparing QH against HTTP/1.1, HTTP/2, and HTTP/3 across multiple workload scenarios, demonstrating wire format efficiency.

2 Background and Related Work

This chapter provides the necessary background for understanding the QH protocol design. We look at HTTP, modern transport protocols, header compression techniques, and the QOTP transport layer that QH builds upon. This helps understand the design decisions made in QH and establishes how it differs from existing approaches.

2.1 HTTP Protocol Evolution

HTTP/1.1 [1] established persistent connections but suffered from fundamental limitations: text-based verbosity, no header compression, and head-of-line blocking where a single delayed request blocks all subsequent requests. Browsers work around parallelism constraints by opening multiple connections per domain, increasing resource consumption.

HTTP/2 [2] introduced binary framing and stream multiplexing to address application layer head-of-line blocking. To reduce header overhead, HTTP/2 employs HTTP/2 Header Compression (HPACK), a stateful compression scheme that maintains synchronized dynamic compression tables between client and server. While HPACK achieves header size reduction, this comes at the cost of implementation complexity and security vulnerabilities.

HTTP/3 [4] moves to UDP-based QUIC transport [5], enabling Zero Round-Trip Time (0-RTT) connection resumption and eliminating transport-layer head-of-line blocking. QUIC Header Compression (QPACK) [6] adapts HPACK for out-of-order stream delivery but becomes even more complex, requiring separate encoder and decoder streams for dynamic table management. QUIC implements custom congestion control, packet loss detection, and recovery mechanisms, increasing implementation complexity compared to relying on kernel TCP implementations.

This progression reveals a consistent pattern: each HTTP version achieves improved performance through increased complexity. HTTP/3 with QUIC represents maximum optimization but increased implementation burden. This raises a design question: can good performance be achieved through simplicity rather than sophisticated compression and state management?

2.2 Modern Transport Protocols

2.2.1 TCP Limitations

Transmission Control Protocol (TCP), the foundation of HTTP/1.1 and HTTP/2, has limitations that impact web performance. A single lost packet blocks delivery of all subsequent data due to head-of-line blocking, affecting even independent streams in HTTP/2. Initial connection establishment requires at least two round trips (Round Trip Time (RTT)): one for the TCP three-way handshake (SYN, SYN-ACK, ACK) and one for the Transport Layer Security (TLS) 1.3 handshake. While TLS 1.3 supports 0-RTT resumption for subsequent connections, this still requires the TCP handshake to complete before application data can be sent.

2.2.2 QUIC Transport

QUIC, originally developed by Google and standardized as RFC 9000 [5], addresses these TCP limitations. The protocol enables 0-RTT resumption, allowing clients to send data in the first packet when resuming connections. Stream multiplexing provides multiple independent byte streams over a single connection, each with individual flow control. Built-in encryption integrates TLS 1.3 into the transport layer, encrypting nearly all protocol metadata.

QUIC's comprehensive feature set comes at the cost of implementation complexity. The protocol specification spans multiple RFCs totaling hundreds of pages, and production implementations require substantial engineering effort. This complexity represents a trade-off between maximum performance and ease of implementation.

2.2.3 UDP-based Custom Transports

Building custom transports on UDP offers flexibility to optimize for specific use cases. Encryption can be integrated at the transport layer, and simpler state machines can be designed for specialized applications. However, these advantages come with challenges. Implementations must provide reliability, ordering, and flow control from scratch. Custom congestion control requires careful design to avoid harming network performance, as it lacks TCP's mature, well-tested algorithms.

Quite Ok Transport Protocol (QOTP) takes this approach, providing a lightweight UDP-based transport while keeping the implementation simple.

2.3 Header Compression Techniques

HTTP headers contribute overhead, particularly for small resources. HTTP/2 and HTTP/3 use complex compression schemes to reduce this overhead.

2.3.1 HPACK (HTTP/2)

HPACK [3] compresses HTTP/2 headers using a combination of techniques. A static table defines common header name-value pairs (for example, `:method: GET` is represented by index 2). A dynamic table maintains recently seen headers, shared between client and server and updated during communication. Additionally, Huffman encoding provides variable-length encoding of header strings, while integer encoding uses variable-length representation for numeric values.

HPACK effectively compresses headers. However, achieving this compression requires state synchronization between peers and dynamic table management, adding implementation complexity. The use of compression with potentially user-controlled data creates vulnerability to compression oracle attacks such as CRIME. Debugging is difficult due to the stateful nature of compression, as understanding compressed output requires knowledge of dynamic table state.

2.3.2 QPACK (HTTP/3)

QPACK [6] adapts HPACK for HTTP/3′s QUIC transport, which delivers streams out-of-order. To handle this, QPACK introduces dedicated encoder and decoder streams for managing dynamic table state separately from data streams. Encoders can avoid dynamic table references entirely to prevent head-of-line blocking, trading compression efficiency for non-blocking delivery.

These modifications make QPACK more complex than HPACK, requiring additional control streams and real-time encoder decisions between compression ratio and blocking risk. This complexity motivated QH's static-only header table approach.

2.3.3 Static Header Tables

QH uses only a static header table, eliminating dynamic compression entirely. This removes state synchronization complexity and enables stateless debugging. Each packet can be decoded independently without connection history. The static table was populated based on real-world header frequency analysis, ensuring compression for common headers while maintaining implementation simplicity.

2.4 QOTP Transport Layer

QOTP (Quite OK Transport Protocol) is a lightweight UDP-based transport protocol designed to provide essential features with minimal complexity. Unlike QUIC's feature set spanning multiple RFCs, QOTP uses opinionated defaults: a single cryptographic suite (Curve25519/ChaCha20-Poly1305), mandatory encryption, and simplified state management.

2.4.1 Core Features

Connection Establishment: QOTP supports two connection establishment modes. With out-of-band key exchange, clients can establish 0-RTT connections by obtaining the server's public key beforehand (e.g., through DNS TXT records at `_qotp.example.com`). When the server's key is known, encrypted data can be transmitted in the first packet without handshake delays. For connections without pre-shared keys, QOTP performs a standard key exchange handshake using ephemeral X25519 keys.

Encryption and Authentication: All QOTP packets are encrypted using ChaCha20-Poly1305, which provides both confidentiality and authentication. Each connection derives a shared secret through X25519 key exchange, which is then used to encrypt all subsequent traffic. Connection IDs are derived from ephemeral public keys, providing 64-bit connection identifiers that enable connection tracking without additional overhead.

Reliability and Flow Control: QOTP implements selective acknowledgment (SACK) for lost packet recovery, retransmitting only missing data rather than entire windows. Flow control operates at both the connection and stream level to prevent sender from overwhelming receiver buffers. The protocol uses a simplified congestion control based on packet loss detection and exponential backoff, prioritizing implementation simplicity over maximum throughput optimization.

Simplified Design Philosophy: Where QUIC offers broad configurability (multiple cipher suites, pluggable congestion control, connection migration), QOTP makes fixed choices. This eliminates negotiation complexity and reduces the specification to a single implementation path.

2.4.2 Relevance to QH

QH and QOTP share the same design philosophy: prioritizing simplicity and reasonable defaults over configurability. Both protocols follow the "Quite Ok" approach, providing essential functionality with minimal complexity rather than comprehensive feature sets. Where HTTP/3 builds on QUIC's multi-RFC specification, QH deliberately builds on QOTP's opinionated simplicity.

This architecture separates concerns cleanly: QOTP handles transport (connections, encryption, reliability), allowing QH to focus entirely on application-layer semantics (binary message formats, header encoding, and request-response patterns). QH does not implement its own security layer or manage streams; it encodes requests and responses as bytes over QOTP streams.

2.5 Alternative Web Protocols

Several projects have explored alternatives to HTTP, each revealing different trade-offs. gRPC [8] uses Protocol Buffers over HTTP/2, achieving efficient binary serialization but requiring full HTTP/2 infrastructure: it extends HTTP rather than replacing it. CoAP [9] provides a UDP-based alternative optimized for resource-constrained IoT devices, trading general-purpose features for minimal overhead.

QH differs by targeting general-purpose web use as a direct HTTP replacement. Unlike gRPC, it replaces the protocol stack rather than building atop it. Unlike CoAP, it targets standard web workloads rather than constrained devices, while maintaining HTTP semantics (methods, status codes, headers) for familiar integration.

2.6 Comparison of HTTP/1.1, HTTP/2, HTTP/3, and QH

The following table summarizes the key differences between HTTP versions and QH:

Feature	**HTTP/1.1**	**HTTP/2**	**HTTP/3**	**QH**
Transport Protocol	TCP	TCP	QUIC (UDP)	QOTP (UDP)
Multiplexing	No (6-8 connections)	Yes (binary frames)	Yes (QUIC streams)	Yes (QOTP streams)
Connection Establishment	TCP: 1 RTT TLS: +1-2 RTT Total: 2-3 RTT	TCP: 1 RTT TLS: +1-2 RTT Total: 2-3 RTT	1-RTT initial (0-RTT on resumption)	0-RTT with DNS key (1-RTT fallback)
Encryption	Optional TLS	TLS 1.2+ (required by browsers)	Mandatory TLS 1.3 (integrated)	Mandatory (ChaCha20-Poly1305)
Certificate Management	X.509 + CA	X.509 + CA	X.509 + CA	DNS TXT records
Header Compression	None	HPACK (dynamic table, Huffman)	QPACK (dynamic + blocking control)	Static table only
Wire Format	Text (ASCII)	Binary frames	Binary frames	Binary (varint-prefixed)
Implementation Complexity	Low	Medium (HPACK, framing)	High (QPACK, QUIC transport)	Low (static headers, simple framing)

2.7 Summary

This chapter established the context for QH's design. HTTP has grown increasingly complex with each version, particularly in header compression where HPACK and QPACK require sophisticated state management. Transport layer evolution from TCP to UDP-based protocols like QUIC and QOTP enables 0-RTT connections. Dynamic header compression achieves high compression ratios but requires complex implementation and state management, while static tables provide simpler implementations with deterministic behavior. Traditional PKI requires certificate provisioning, renewal, and CA trust management, whereas DNS-based key distribution eliminates this overhead. Together, these observations motivate QH's design: achieving practical web performance through simplicity rather than sophisticated optimization.

The next chapter presents QH's protocol design, showing how these observations inform specific design decisions.

3 Protocol Design

This chapter presents the design rationale and technical specification of the QH protocol.

3.1 Design Rationale

3.1.1 Design Goals

The QH protocol design was guided by three primary goals:

Simplicity: The protocol prioritizes implementation simplicity and maintainability over feature completeness. A competent programmer should be able to implement a basic client or server in a few hundred lines of code without extensive libraries or complex state machines. Wire formats should be deterministic and debuggable with standard tools. This goal explicitly rejects features that increase complexity (e.g., dynamic header compression, server push, complex content negotiation).

Performance: Within the simplicity constraint, the protocol must remain within 10-15% wire format size of HTTP/2 and HTTP/3 for real-world web workloads. This means efficient binary wire formats, minimal per-request overhead, and leveraging modern transport features (multiplexing, 0-RTT) through the underlying QOTP layer rather than reimplementing them at the application layer.

Security: Security must be mandatory and built-in rather than optional. The protocol leverages QOTP's transport-layer encryption (Curve25519/ChaCha20-Poly1305) and implements DNS-based key distribution to avoid the complexity of traditional Public Key Infrastructure (PKI) and certificate validation while ensuring all communication is encrypted.

These goals create inherent tensions. Performance optimization often increases complexity (e.g., dynamic compression vs static tables). Security features add implementation overhead (though QH delegates this to QOTP). The subsequent sections explain how specific design decisions navigate these trade-offs.

3.1.2 Simplification Strategy

QH achieves simplicity through deliberate feature omission rather than sophisticated optimization. This philosophy differs fundamentally from HTTP/2 and HTTP/3, which layer increasingly complex mechanisms to maximize performance.

The core simplification strategy involves three principles:

Static over Dynamic: Where HTTP/2 and HTTP/3 use dynamic compression tables that change during connection lifetime, QH uses a static header table fixed at protocol design time. This eliminates state synchronization, reduces implementation complexity, and enables stateless debugging.

Binary-First Design: QH was designed as binary from the start, without HTTP compatibility constraints. While HTTP/2 also uses binary encoding, it maintains complex HTTP/1.1 semantics including large method sets, detailed status codes, and backwards-compatible header names. QH simplifies these by design rather than inheriting them.

Opinionated Defaults: Instead of making everything configurable, QH makes reasonable default choices. The protocol specifies a single cryptographic suite, a fixed header table, and a simple encoding scheme without negotiation overhead.

Several HTTP features were deliberately excluded from QH Version 0:

Server Push: HTTP/2′s server push mechanism allows servers to send resources proactively. QH omits this feature due to implementation complexity and limited practical adoption.

Transfer-Encoding: QH uses explicit length prefixes for all fields, eliminating chunked transfer encoding and associated parsing complexity.

Content Negotiation Extensions: While QH supports basic Accept-Encoding for compression, it omits quality values, wildcards, and complex negotiation features. Clients list acceptable encodings in preference order.

Each excluded feature makes QH simpler to implement but less capable than HTTP. This is acceptable for most web applications.

3.1.3 Static Header Table Design

Header compression presents a fundamental design choice: static tables, dynamic tables, or both. HTTP/2′s HPACK and HTTP/3′s QPACK use both, achieving high compression ratios at the cost of state management complexity. QH uses only static tables, prioritizing simplicity over maximum compression.

3.1.3.1 Rationale for Static vs Dynamic Compression

Dynamic compression maintains a table of recently seen headers shared between client and server. As communication proceeds, frequently used headers enter the dynamic table and can be referenced by index. This adapts to each connection's specific traffic patterns.

However, dynamic compression introduces several complexities:

State Synchronization: Both endpoints must maintain identical dynamic table state. Table updates must be processed in order, creating dependencies between streams. HTTP/3′s QPACK requires dedicated encoder and decoder streams solely for managing this state.

Memory Management: Dynamic tables consume memory proportional to connection lifetime and traffic patterns. Implementations must handle table size limits, policies, and memory pressure.

Debugging Complexity: Compressed messages cannot be decoded without reconstructing dynamic table state from connection history. Network traces become difficult to analyze.

Static tables (Static header table) eliminate these issues entirely. The header table is fixed at protocol design time and distributed with implementations. Every endpoint uses identical mappings. Messages can be decoded independently without connection state.

The trade-off is encoding efficiency. Dynamic compression adapts to each application's specific headers, while static tables must anticipate common usage patterns. QH accepts this trade-off.

3.1.3.2 Header Frequency Analysis Methodology

To design an effective static header table, empirical data about real-world header usage was required. We developed a custom Chrome browser extension to collect header frequency statistics from actual web browsing. The extension is publicly available in both the Chrome Web Store and Firefox Add-ons.

The Chrome extension (`http-header-tracker`) records all HTTP request and response headers from the browser's network activity. For each header, it tracks complete key-value pairs (e.g., `content-type: application/json`) with occurrence counts. The extension automatically anonymizes sensitive headers (authorization, cookies, API keys) and detects potential secrets through pattern matching (JWT tokens, UUIDs, Bearer tokens). It supports both local-only mode (data stored in browser) and server mode (automatic upload to a remote server for aggregation).

The extension operates transparently during normal browsing, capturing headers from all resource types: HTML documents, stylesheets, scripts, images, API requests, and third-party resources.

Collection Details:

- Collection period: 27.10.2025 to 05.11.2025
- Request headers: 1,010,192 occurrences (732,705 with complete key-value pairs)
- Response headers: 1,800,806 occurrences (1,276,592 with complete key-value pairs)
- Independent data sources: We tracked headers separately (one using Chrome, one using Firefox), with data merged to ensure diverse coverage across browsers

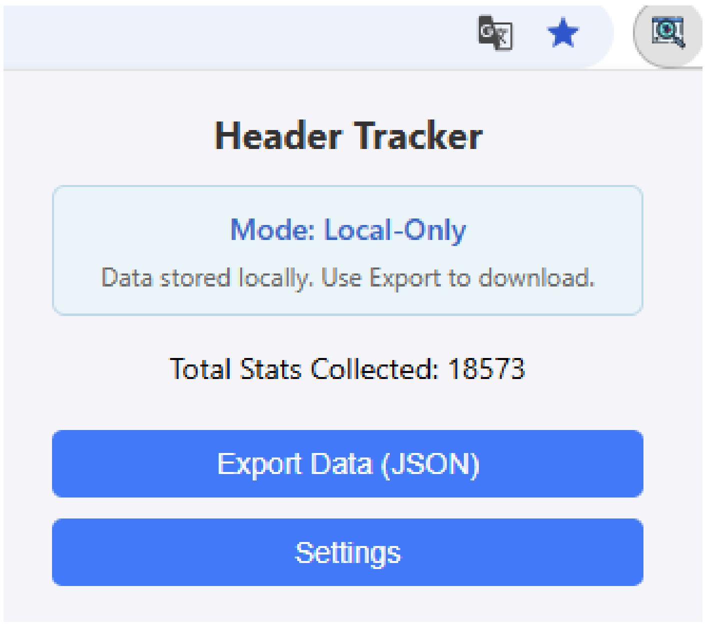
Data Processing:

After collection, we processed the header statistics using Python scripts:

1. `merge_headers.py` - Merges multiple JSON statistics files from independent collection sources, aggregating header counts and generating an Excel file with four sheets: Request Complete Pairs, Request Name Only, Response Complete Pairs, and Response Name Only
2. Manual review - We manually reviewed the Excel output to select headers for inclusion in the static table, prioritizing high-frequency headers while excluding platform-specific

headers (e.g., `X-GitHub-*`, `X-Amz-*`) to maintain generality. During this process, we also added missing entries that were not captured in our browsing data to ensure comprehensive coverage of common web headers, referencing the QPACK static table [10] for additional common headers

3. `generate_outputs.py` - Generates the static header table in multiple formats (Go source code, Markdown documentation, JSON specification) from the curated Excel data

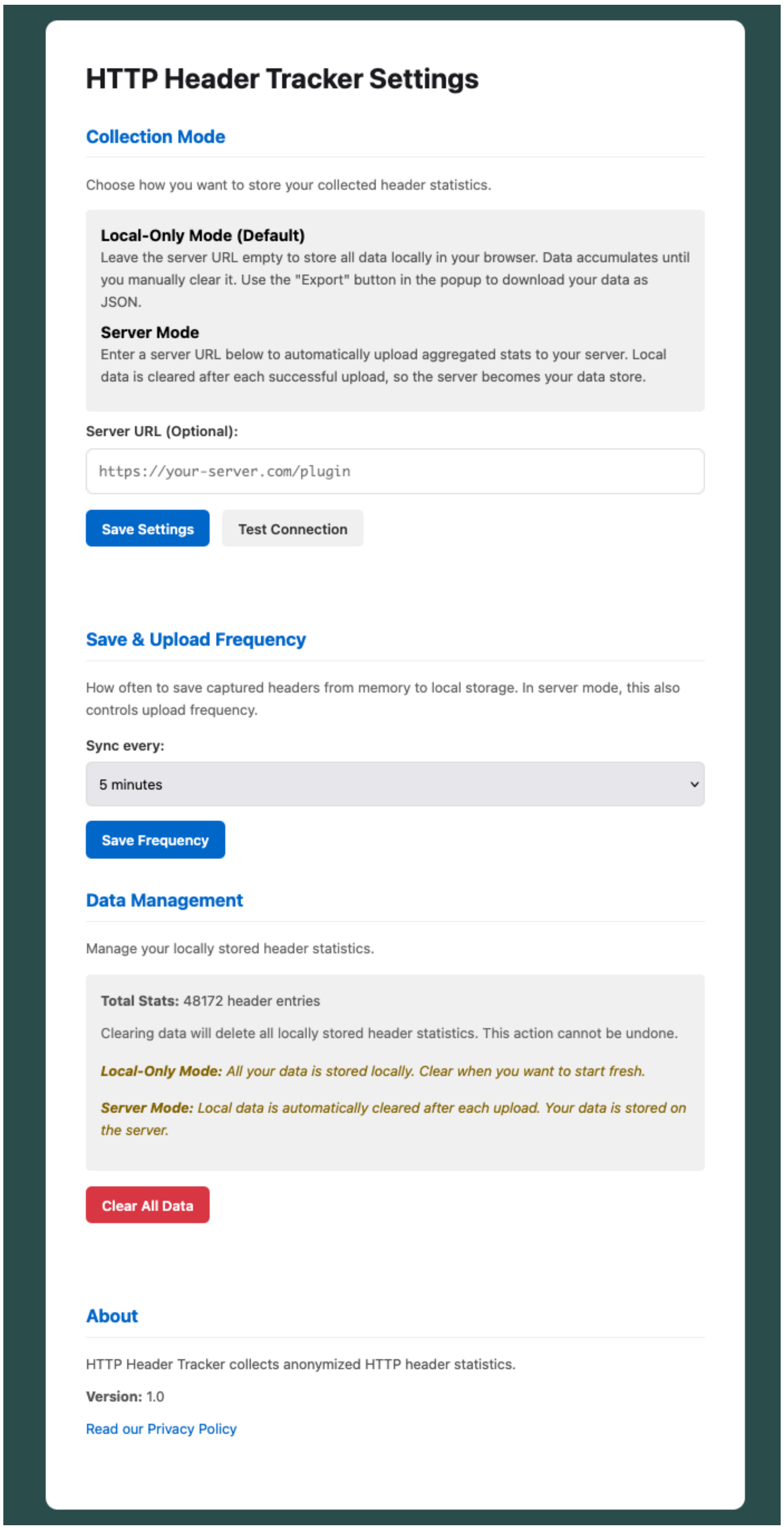

Header Tracker

Mode: Local-Only

Data stored locally. Use Export to download.

Total Stats Collected: 18573

Figure 5: Screenshot of the Chrome Header Tracking Extension Popup



HTTP Header Tracker Settings

Collection Mode

Choose how you want to store your collected header statistics.

Local-Only Mode (Default)
Leave the server URL empty to store all data locally in your browser. Data accumulates until you manually clear it. Use the "Export" button in the popup to download your data as JSON.

Server Mode
Enter a server URL below to automatically upload aggregated stats to your server. Local data is cleared after each successful upload, so the server becomes your data store.

Server URL (Optional): https://your-server.com/plugin

Save & Upload Frequency

How often to save captured headers from memory to local storage. In server mode, this also controls upload frequency.

Sync every: 5 minutes

Data Management

Manage your locally stored header statistics.

Total Stats: 48172 header entries

Clearing data will delete all locally stored header statistics. This action cannot be undone.

Local-Only Mode: *All your data is stored locally. Clear when you want to start fresh.*

Server Mode: *Local data is automatically cleared after each upload. Your data is stored on the server.*

About

HTTP Header Tracker collects anonymized HTTP header statistics.

Version: 1.0

Read our Privacy Policy

Figure 6: Extension Settings Page Showing Collection Modes and Data Management

3.1.3.3 Analysis Results

The frequency analysis revealed clear patterns in header usage. A small number of header combinations account for the vast majority of traffic.

For request headers, browser-generated headers dominate:

- `Sec-Ch-Ua-Mobile: ?0` appears in nearly every request from desktop browsers
- `Accept: */*` appears in most resource requests
- `Sec-Fetch-*` headers appear consistently due to browser security features
- User-Agent headers always vary by browser version, making them poor candidates for complete key-value storage

For response headers, server configuration patterns emerge:

- Security headers (`X-Content-Type-Options: nosniff`, `Strict-Transport-Security`) use consistent values
- Cache-Control directives cluster around common patterns (`public, max-age=31536000`)
- Content-Type headers show clear favorites (`application/json`, `text/html`)

The analysis quantified the compression potential of static tables. For request headers, the top 64 complete key-value pairs covered approximately 45% of all headers, while the top 36 header names covered an additional 40%, leaving only 15% requiring custom encoding. Response headers showed similar distributions.

3.1.3.4 Selected Headers and Justification

The final static header table (`headers.go`) contains 100 request header entries and 190 response header entries. Headers were manually curated from the frequency analysis, prioritizing broadly applicable patterns over vendor-specific headers. Platform-specific headers (e.g., `X-GitHub-*`, `X-Amz-*`, `CF-*`, `X-Google-*`) were excluded to maintain generality across diverse web applications.

Request header table: complete pairs 0x01-0x40 (64 entries), name-only 0x41-0x64 (36 entries). Response header table: complete pairs 0x01-0x8D (141 entries), name-only 0x8E-0xBE (49 entries).

Headers with high frequency but diverse values use name-only encoding: `User-Agent` varies by browser version, `Date` contains timestamps, `Cookie` and `Authorization` carry session-specific data, and `Content-Length` varies per response.

3.1.4 Binary Format Design

QH uses binary encoding for all protocol metadata: methods, status codes, header names, and length fields.

Binary encoding enables compact representation and deterministic parsing. Unlike HTTP/1.1, which requires buffering until delimiter sequences (`\r\n\r\n`), QH messages are self-describing through first-byte encoding, varint length prefixes, and fixed field ordering.

This eliminates tokenization, delimiter scanning, and parsing ambiguities around whitespace or character encoding.

The trade-off is reduced human readability. Protocol tooling, including the Wireshark dissector, compensate for this limitation.

3.1.5 Varint Encoding Selection

QH uses unsigned Little Endian Base 128 (LEB128) Variable-length Integer (Varint) encoding for all length fields. Variable-length encoding represents smaller values with fewer bytes, providing space savings since most message fields are short (hosts, paths, headers).

LEB128 was chosen because it is an industry standard (Protocol Buffers, Apache Avro), has built-in Go library support (`encoding/binary`), and efficiently handles common web message sizes. Fixed-width length fields would waste space for the small field sizes typical in web requests.

3.1.6 Method and Status Code Encoding

QH encodes HTTP methods and status codes as compact binary values rather than text strings.

3.1.6.1 Request Method Encoding

Request methods are encoded in the first byte using a 3-bit field (bits 3-5), allowing up to 8 methods. The first byte layout is:

```
[VV][MMM][RRR]
Bits 7-6: Version (2 bits)
Bits 5-3: Method (3 bits)
Bits 2-0: Reserved (3 bits)
```

QH Version 0 defines seven methods with the following encoding:

```
GET     = 000 = 0x00
POST    = 001 = 0x08
PUT     = 010 = 0x10
PATCH   = 011 = 0x18
DELETE  = 100 = 0x20
HEAD    = 101 = 0x28
OPTIONS = 110 = 0x30
```

This encoding provides constant size (1 byte), fast decoding via bitwise operations, and type safety (invalid values immediately detected). HTTP/1.1 methods require 3-7 ASCII characters (“GET”, “DELETE”), while QH uses a single byte.

3.1.6.2 Response Status Code Encoding

Response status codes are packed into the first byte using a 6-bit compact code field:

```
[VV][CCCCCC]
Bits 7-6: Version (2 bits)
Bits 5-0: Compact status code (6 bits)
```

The 6-bit field supports 64 status codes (0-63). QH maps HTTP status codes to compact codes based on frequency. Common mappings include:

```
200 OK                    -> compact 20
404 Not Found             -> compact 44
500 Internal Server Error -> compact 50
302 Found                 -> compact 32
...
```

For status codes not explicitly mapped, the encoder defaults to compact code 50 (500 Internal Server Error).

The trade-off is limited extensibility. QH accepts this limitation, betting that standard HTTP methods and common status codes suffice for typical applications.

3.1.7 Transport Layer Selection

QH is built on top of QOTP (Quite Ok Transport Protocol), an existing UDP-based transport layer. QOTP provides the necessary features for QH: 0-RTT connection establishment, mandatory encryption (Curve25519/ChaCha20-Poly1305) and stream multiplexing.

The clean layer separation means QH focuses entirely on application layer concerns (request and response semantics, message encoding, header compression) while QOTP handles all transport responsibilities.

3.2 Protocol Specification

This section provides the complete technical specification of the QH protocol wire format, message structure, and encoding rules.

3.2.1 Protocol Stack Overview

QH implements an application-layer protocol running on top of the QOTP transport. The protocol stack layers are:

```
┌──────────────────────────────────────────────────────────────┐
│           Application Layer: QH Protocol                     │
│  • HTTP-inspired request/response semantics                  │
│  • Compact binary encoding with static header table          │
│  • DNS TXT record key distribution                           │
├──────────────────────────────────────────────────────────────┤
│           Transport Layer: QOTP                              │
│  • 0-RTT connection establishment                            │
│  • Built-in encryption (Curve25519/ChaCha20-Poly1305)        │
│  • UDP-based communication                                   │
│  • Stream multiplexing                                       │
├──────────────────────────────────────────────────────────────┤
│              Network Layer: UDP/IP                           │
│  • Standard network layer                                    │
└──────────────────────────────────────────────────────────────┘
```

Data flows through the layers sequentially. An outgoing request follows this path:

1. Application constructs a `Request` object
2. QH encodes the request into binary format
3. QH writes the encoded bytes to a QOTP stream
4. QOTP encrypts and packetizes the stream data
5. QOTP transmits UDP packets to the server
6. Server's QOTP receives and decrypts packets
7. Server's QH decodes the binary format
8. Server's application processes the request

The response follows the reverse path, with optional compression applied by the server based on client capabilities.

3.2.1.1 Wire Format Encoding

QH wire formats use two fundamental encoding mechanisms:

- **Varint Encoding (LEB128):** All variable-length integers use unsigned LEB128 encoding, which represents integers using 7 bits per byte for data and 1 bit (MSB) as a continuation flag.
- **Length-Prefixed Format:** All variable-length fields follow the pattern:

```
<varint:length><data>
```

For example, encoding the string "example.com" (11 bytes):

```
0x0B example.com
```

Where `0x0B` is the varint encoding of 11. This format is binary-safe: the data portion can contain any byte values including nulls, newlines, or control characters. The length prefix specifies exactly where the field ends.

3.2.2 Request Format

QH requests encode the HTTP method, target host and path, headers, and optional body.

3.2.2.1 Structure

The complete request format is:

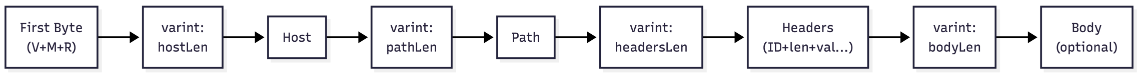


Figure 7: QH Request Structure Overview

3.2.2.2 Field Breakdown

First Byte: Encodes protocol version and method code (see Section 3.1.6).

Host: Varint length followed by hostname as UTF-8 string. Required, maximum 253 bytes.

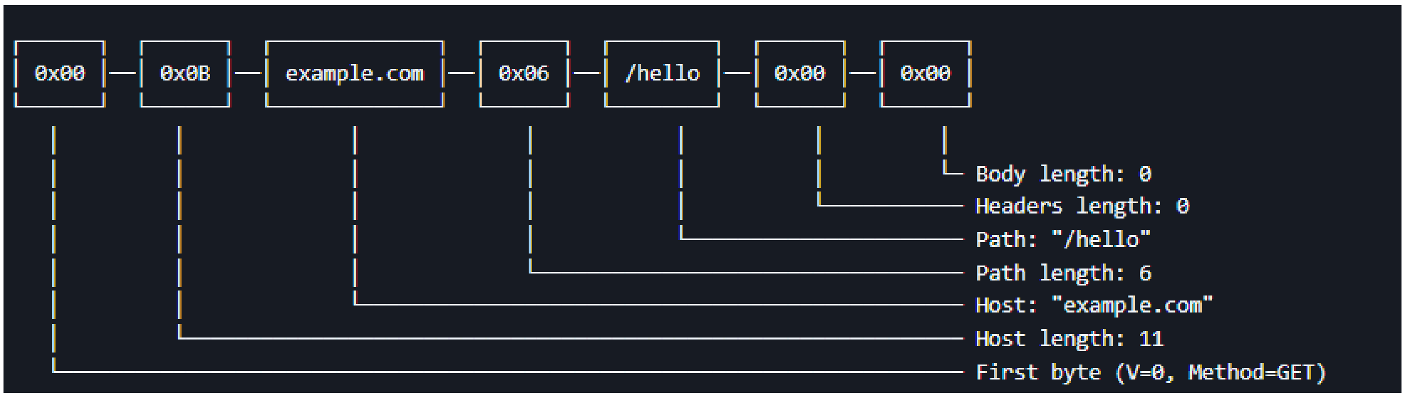
Path: Varint length followed by path as UTF-8 string. Defaults to "/" if empty.

Headers: Varint total byte count followed by encoded header entries (see Section 3.2.4).

Body: Varint length followed by optional request body content.

3.2.2.3 Example

A GET request to `/hello` with no headers or body:



```
┌──────┐  ┌──────┐  ┌─────────────┐  ┌──────┐  ┌────────┐  ┌──────┐  ┌──────┐
│ 0x00 │──│ 0x0B │──│ example.com │──│ 0x06 │──│ /hello │──│ 0x00 │──│ 0x00 │
└──────┘  └──────┘  └─────────────┘  └──────┘  └────────┘  └──────┘  └──────┘
   │         │             │             │          │          │         │
   │         │             │             │          │          │         └─ Body length: 0
   │         │             │             │          │          └─────────── Headers length: 0
   │         │             │             │          └────────────────────── Path: "/hello"
   │         │             │             └───────────────────────────────── Path length: 6
   │         │             └─────────────────────────────────────────────── Host: "example.com"
   │         └───────────────────────────────────────────────────────────── Host length: 11
   └─────────────────────────────────────────────────────────────────────── First byte (V=0, Method=GET)
```

Figure 8: QH Request Format (from the GitHub protocol-definitions.md file)

3.2.3 Response Format

QH responses encode the HTTP status code, headers, and optional body.

3.2.3.1 Structure

The complete response format is:

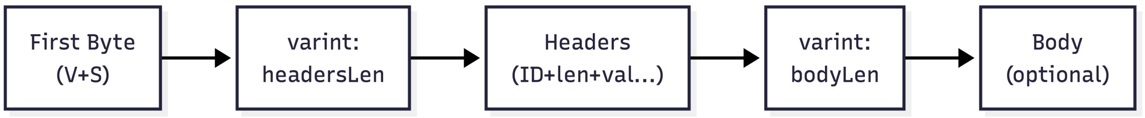


Figure 9: QH Response Structure Overview

3.2.3.2 Field Breakdown

First Byte: Encodes protocol version and compact status code (see Section 3.1.6).

Headers: Varint total byte count followed by encoded header entries using the response header static table.

Body: Varint length followed by optional response body content. May be compressed if Content-Encoding header is present.

3.2.3.3 Example

A 200 OK response with JSON body:

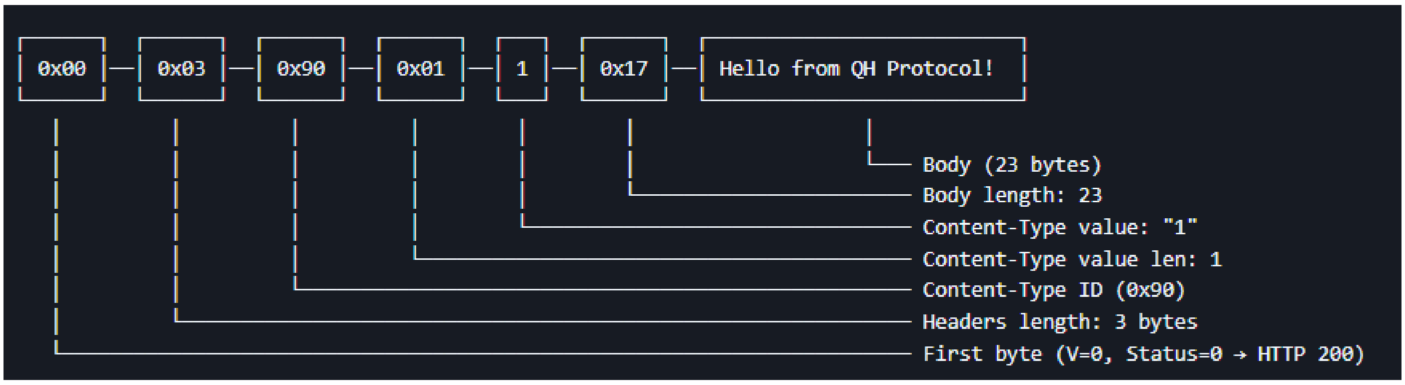


Figure 10: QH Response Format (from the GitHub protocol-definitions.md file)

3.2.4 Header Encoding

QH headers use a static header table with three encoding formats.

3.2.4.1 Three Encoding Formats

Format 1: Complete Key-Value Pair

For headers in the static table with fixed values:

```
<1-byte:headerID>
```

Example: `Content-Type: application/json` is request header ID 0x0E, encoded as `0x0E` (1 byte vs 32 bytes in HTTP/1.1 including CRLF).

Format 2: Header Name with Custom Value

For common header names with variable values:

```
<1-byte:headerID><varint:valueLen><value>
```

Example: `User-Agent: Mozilla/5.0 (custom browser)` where User-Agent is header ID 0x41 in request table. Encoded as:

```
0x41 0x1E Mozilla/5.0 (custom browser)
```

Where `0x1E` is varint encoding of 30 (the value length).

Format 3: Custom Header

For headers not in the static table:

```
<1-byte:0x00><varint:keyLen><key><varint:valueLen><value>
```

Example: `X-Request-ID: abc123` encodes as:

```
0x00 0x0C X-Request-ID 0x06 abc123
```

Where:
- `0x00` indicates custom header
- `0x0C` is varint encoding of 12 (length of "X-Request-ID")
- `0x06` is varint encoding of 6 (length of "abc123")

3.2.5 Content Encoding and Compression

QH supports optional response body compression using standard algorithms. Request body compression is not supported, as it is uncommon in typical web traffic.

3.2.5.1 Compression Strategy

QH integrates compression into the message encoding process. Compression is applied when:

- Client sends Accept-Encoding header with supported algorithms
- Response body size exceeds 1 KB
- Content-type suggests compressible data
- Compressed size is actually smaller than uncompressed size

3.2.5.2 Supported Algorithms

QH supports three compression algorithms:
- `gzip` - GNU zip compression
- `br` - Brotli compression
- `zstd` - Zstandard compression

3.2.5.3 Negotiation

Clients indicate supported compression algorithms via the Accept-Encoding request header:

```
Accept-Encoding: zstd, br, gzip
```

Multiple encodings are comma-separated in preference order. QH interprets this as a preference-ordered list: the first algorithm is most preferred.

The server selects the first algorithm from the client's list that the server also supports. For example:

```
Client sends:       Accept-Encoding: zstd, br, gzip
Server supports:    br, gzip
Server selects:     br (first match)
```

If no common algorithm exists, or if Accept-Encoding is missing/empty, the response is sent uncompressed.

Unlike HTTP/1.1, QH does not support:
- Quality values (`gzip;q=0.8`)
- Wildcard encodings (`*`)

- Identity encoding (use empty Accept-Encoding)

3.2.6 Protocol Extensions and Future Compatibility

The 2-bit version field in the first byte supports four protocol versions (0-3). Servers receiving unsupported version numbers should respond with status 505 (Version Not Supported).

The 3 reserved bits in the request first byte provide space for future extensions without changing the wire format. Current implementations must set reserved bits to 0.

Header IDs above the currently allocated ranges (0x64 for requests, 0xBE for responses) are reserved for future allocation. The 0x00 custom header format allows any header to be transmitted, ensuring forward compatibility even when headers are not in the static table.

4 Implementation

4.1 Project Structure

The implementation is structured as follows:

```
qh/
├── *.go                      # Core protocol implementation
├── benchmark/
│   ├── cmd/qhbench/         # Benchmark runner
│   ├── testdata/            # Test cases
│   └── encoder_*.go         # Protocol encoders (HTTP/1, HTTP/2, HTTP/3, QH)
├── examples/                # Example applications
│   ├── client/
│   ├── server/
│   └── *-concurrent/        # Concurrent request handling examples
├── docs/
│   ├── protocol-definition.md # Protocol specification
│   ├── benchmarks/README.md # Benchmark methodology
│   └── benchmarks/report.md # Benchmark results
└── data/                     # Static header table (JSON)
```

The core protocol logic resides in the root-level Go files, following the flat structure convention used by Go's standard library `net/http` package and similar networking libraries. This approach keeps related protocol components together while separating benchmarking and example code into dedicated modules. All public APIs are documented using Go doc comments, making the implementation accessible through standard Go tooling (`go doc`, `pkg.go.dev`).

4.2 System Architecture

The QH protocol is implemented in Go, primarily because QOTP was already written in Go. Additionally, Go has mature reference implementations of related protocols (HTTP/2, HTTP/3, QUIC) that served as valuable resources during development. The implementation requires Go 1.25.1 or later.

The system follows a layered architecture where QH operates as an application protocol on top of the QOTP transport layer. This separation of concerns allows QH to focus on HTTP semantics (methods, headers, status codes) while delegating connection management, encryption, and reliability to QOTP. The architecture mirrors the HTTP/3-over-QUIC model but with a simpler protocol design at both layers.

The implementation only uses three core external dependencies:

1. QOTP transport layer (`github.com/qo-proto/qotp`)
2. Compression libraries (`github.com/andybalholm/brotli` for Brotli and `github.com/klauspost/compress` for zstd)
3. Testify for testing utilities `github.com/stretchr/testify`

4.3 Core Implementation

4.3.1 Protocol Components

The protocol implementation is built around several core data structures defined in `protocol.go`:

Request and Response Types: The `Request` struct encapsulates all request data (method, host, path, headers, body), while the `Response` struct contains response data (status code, headers, body). Both use `map[string]string` for headers and `[]byte` for body content.

Method Encoding: QH methods are represented as a custom `Method` type (integer enum) that maps directly to the 3-bit wire format encoding.

Header Tables: The static header table is defined in `headers.go` with two lookup maps: one for complete header key-value pairs (Format 1) and one for header names with dynamic values (Format 2). This dual-map structure enables efficient O(1) lookup during encoding.

Varint Utilities: The `varint.go` file provides LEB128 encoding and decoding functions for variable-length integers used throughout the protocol. This includes functions for reading and writing varints.

```go
func ReadUvarint(buf []byte, offset int) (uint64, int, error)
func AppendUvarint(buf []byte, v uint64) []byte
```

Code Organization: To maintain a single source of truth for bidirectional mappings, the implementation leverages Go's `init()` function to automatically generate reverse lookup maps. Both `headers.go` and `status.go` follow this pattern: each file defines the forward mapping (such as header ID to header name, or HTTP status code to compact code), and an `init()` function programmatically populates the corresponding reverse map. This approach eliminates manual duplication and ensures consistency between encoding and decoding paths, reducing maintenance burden and preventing synchronization errors.

4.3.2 Encoding and Decoding

The protocol implements binary encoding that transforms high-level Request and Response objects into compact wire format.

Request Encoding: Requests are encoded by first writing a single byte containing version and method bits, followed by varint-prefixed host and path strings, then encoded headers, and finally the body.

Response Encoding: Responses follow a similar pattern but use a status code instead of method. The first byte contains version (2 bits) and status code (6 bits), followed by varint-prefixed headers and body.

Header Encoding Algorithm:

To encode a header:

1. Check if complete key-value pair exists in static table. If yes, write the 1-byte ID (Format 1) and return.
2. Check if header name exists in static table. If yes, write the 1-byte name ID, then varint value length, then value (Format 2) and return.
3. Otherwise, write 0x00, then varint key length, then key, then varint value length, then value (Format 3).

Header Decoding Algorithm:

To decode a header:

1. Read 1 byte as header ID.
2. If ID is 0x00, read varint key length, read key, read varint value length, read value (Format 3).
3. Otherwise, look up ID in static table. If table entry has a value, use complete pair (Format 1). If table entry has empty value, read varint value length and value (Format 2).

4.3.3 Client Architecture

The client implementation in `client.go` provides a stateful connection manager with support for DNS-based key discovery and automatic compression handling.

Connection Management: The `Client` struct maintains a single QOTP connection per instance, with an atomic stream ID counter for multiplexing multiple requests over a single connection.

DNS Key Discovery: When connecting, the client performs concurrent DNS lookups for both the A/AAAA record (IP address) and the TXT record at `_qotp.<hostname>` (server public key). If a valid key is found, the client establishes a 0-RTT connection; otherwise, it falls back to standard key exchange.

Request Methods: The client provides convenience methods for each QH method (GET, POST, PUT, etc.) that handle encoding, transmission, response parsing, decompression, and redirect following.

Compression Handling: The client automatically adds `Accept-Encoding: zstd, br, gzip` to requests and decompresses responses based on the `Content-Encoding` header.

4.3.4 Server Architecture

The server implementation in `server.go` uses a handler-based routing system similar to Go's `net/http` package.

Handler Registration: Handlers are registered per path and method using `HandleFunc`. The server maintains a nested map structure (`map[string]map[Method]Handler`) for O(1) handler lookup.

Request Processing: The server listens on a QOTP listener. Within each connection, it reads requests from streams, decodes them, looks up the appropriate handler, executes it, and encodes the response.

Compression Strategy: Before sending responses, the server checks if compression is appropriate based on client preferences (`Accept-Encoding` header), body size (must exceed `minCompressionSize`), and content type (skips binary data). It then compresses the response with the first mutually-supported algorithm.

4.3.5 Compression Integration

The protocol supports gzip, brotli, and zstd compression.

Algorithm Selection: The server parses the client's `Accept-Encoding` header to determine supported algorithms in preference order. It then selects the first algorithm that both client and server support.

Compression Decision: Compression is only applied when the response body exceeds the configured minimum size and when the compressed size is actually smaller than the original.

Decompression: The client automatically detects the `Content-Encoding` response header and applies the appropriate decompression algorithm.

4.3.6 QOTP Integration

The QH protocol is implemented as an application-layer protocol on top of QOTP, similar to how HTTP/3 runs on QUIC.

QH mainly uses QOTP's `Listener` and `Conn` types for connection management. The QOTP layer handles encryption, reliability, and congestion control, allowing QH to focus on application semantics. Each QH request-response exchange uses a separate QOTP stream, identified by an incrementing stream ID. This allows multiple concurrent requests over a single connection. QOTP provides mandatory encryption (curve25519/chacha20-poly1305), so QH does not need to implement its own TLS-like layer. The keylog mechanism (enabled with build tag `-tags keylog`) exports QOTP session keys for debugging with Wireshark.

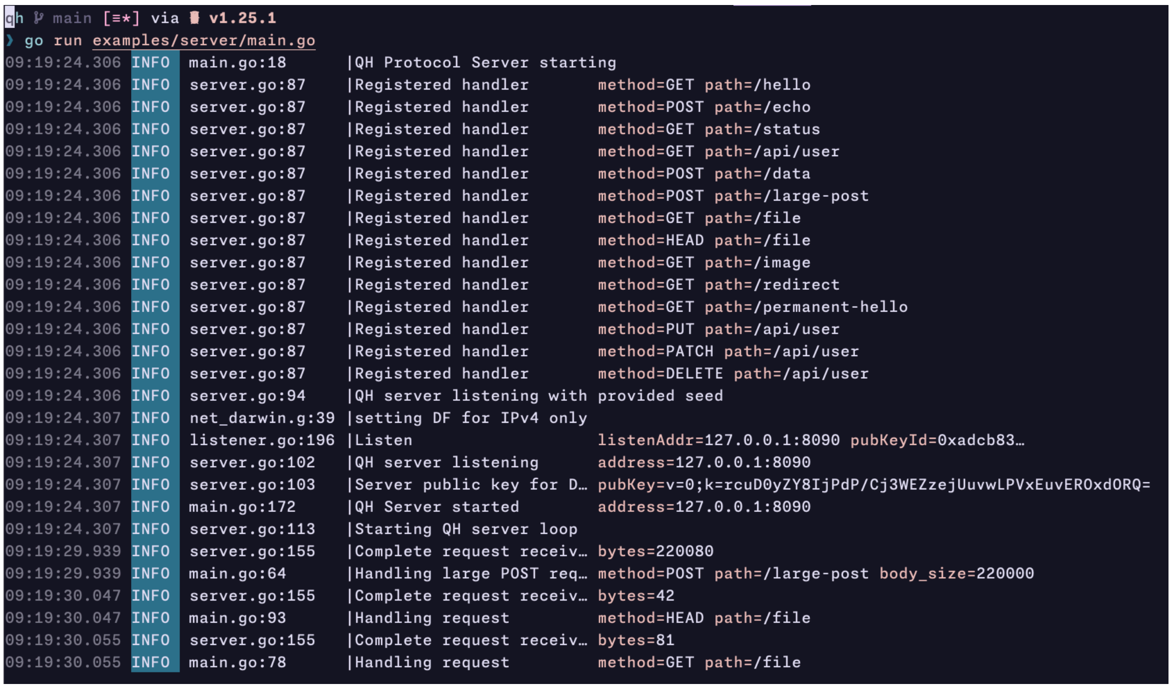
4.3.7 Example Applications

The implementation includes example applications that demonstrate protocol usage and serve as reference implementations for developers.

Basic Server: The `examples/server` application demonstrates a complete QH server with multiple endpoint types. It includes text responses (`/hello`, `/status`), JSON API endpoints (`/api/user`), file serving with different content types, and redirect handling. The server also shows how to use keylog integration for Wireshark debugging when built with the `keylog` tag.

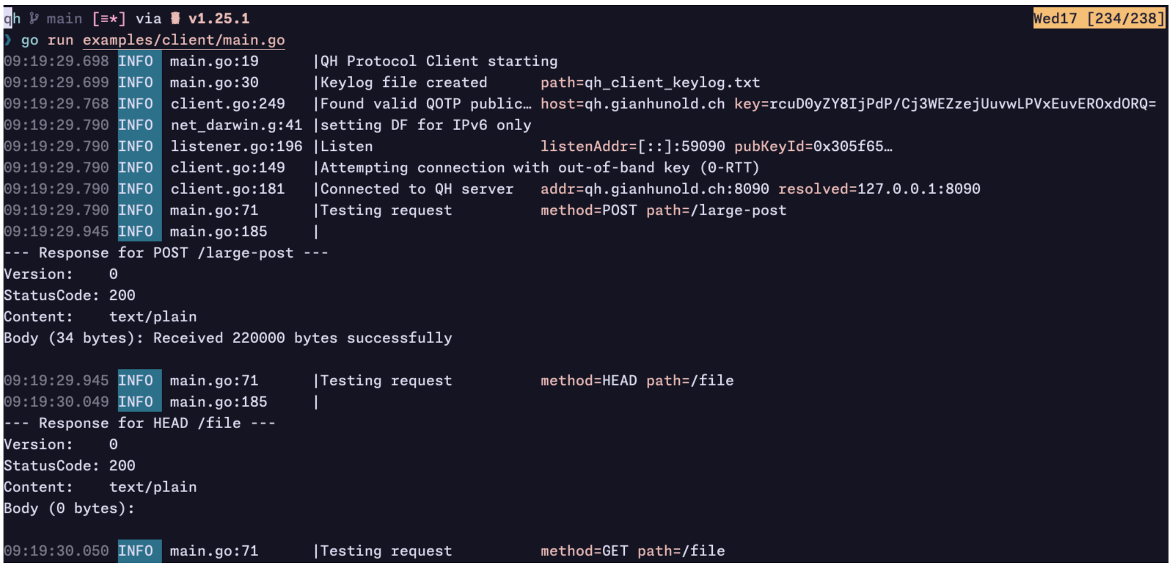
Basic Client: The `examples/client` application demonstrates all QH methods (GET, POST, PUT, PATCH, HEAD, DELETE) handling various scenarios: different payload sizes, JSON data, file downloads, error responses, and redirects.

Concurrent Examples: The `client-concurrent` and `server-concurrent` examples demonstrate stream multiplexing. The client launches multiple requests concurrently using goroutines over a single connection, while the server includes an artificial 200ms delay per request.



```
qh  main [≡*] via  v1.25.1
❯ go run examples/server/main.go
09:19:24.306 INFO  main.go:18       |QH Protocol Server starting
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/hello
09:19:24.306 INFO  server.go:87     |Registered handler       method=POST path=/echo
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/status
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/api/user
09:19:24.306 INFO  server.go:87     |Registered handler       method=POST path=/data
09:19:24.306 INFO  server.go:87     |Registered handler       method=POST path=/large-post
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/file
09:19:24.306 INFO  server.go:87     |Registered handler       method=HEAD path=/file
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/image
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/redirect
09:19:24.306 INFO  server.go:87     |Registered handler       method=GET path=/permanent-hello
09:19:24.306 INFO  server.go:87     |Registered handler       method=PUT path=/api/user
09:19:24.306 INFO  server.go:87     |Registered handler       method=PATCH path=/api/user
09:19:24.306 INFO  server.go:87     |Registered handler       method=DELETE path=/api/user
09:19:24.306 INFO  server.go:94     |QH server listening with provided seed
09:19:24.307 INFO  net_darwin.g:39  |setting DF for IPv4 only
09:19:24.307 INFO  listener.go:196  |Listen                   listenAddr=127.0.0.1:8090 pubKeyId=0xadcb83…
09:19:24.307 INFO  server.go:102    |QH server listening      address=127.0.0.1:8090
09:19:24.307 INFO  server.go:103    |Server public key for D… pubKey=v=0;k=rcuD0yZY8IjPdP/Cj3WEZzejUuvwLPVxEuvEROxdORQ=
09:19:24.307 INFO  main.go:172      |QH Server started        address=127.0.0.1:8090
09:19:24.307 INFO  server.go:113    |Starting QH server loop
09:19:29.939 INFO  server.go:155    |Complete request receiv… bytes=220080
09:19:29.939 INFO  main.go:64       |Handling large POST req… method=POST path=/large-post body_size=220000
09:19:30.047 INFO  server.go:155    |Complete request receiv… bytes=42
09:19:30.047 INFO  main.go:93       |Handling request         method=HEAD path=/file
09:19:30.055 INFO  server.go:155    |Complete request receiv… bytes=81
09:19:30.055 INFO  main.go:78       |Handling request         method=GET path=/file
```

Figure 11: Example QH Server CLI



```
qh  main [≡*] via  v1.25.1                                                                              Wed17 [234/238]
❯ go run examples/client/main.go
09:19:29.698 INFO  main.go:19      |QH Protocol Client starting
09:19:29.699 INFO  main.go:30      |Keylog file created      path=qh_client_keylog.txt
09:19:29.768 INFO  client.go:249   |Found valid QOTP public… host=qh.gianhunold.ch key=rcuD0yZY8IjPdP/Cj3WEZzejUuvwLPVxEuvEROxdORQ=
09:19:29.790 INFO  net_darwin.g:41 |setting DF for IPv6 only
09:19:29.790 INFO  listener.go:196 |Listen                   listenAddr=[::]:59090 pubKeyId=0x305f65…
09:19:29.790 INFO  client.go:149   |Attempting connection with out-of-band key (0-RTT)
09:19:29.790 INFO  client.go:181   |Connected to QH server   addr=qh.gianhunold.ch:8090 resolved=127.0.0.1:8090
09:19:29.790 INFO  main.go:71      |Testing request          method=POST path=/large-post
09:19:29.945 INFO  main.go:185     |
--- Response for POST /large-post ---
Version:    0
StatusCode: 200
Content:    text/plain
Body (34 bytes): Received 220000 bytes successfully

09:19:29.945 INFO  main.go:71      |Testing request          method=HEAD path=/file
09:19:30.049 INFO  main.go:185     |
--- Response for HEAD /file ---
Version:    0
StatusCode: 200
Content:    text/plain
Body (0 bytes):

09:19:30.050 INFO  main.go:71      |Testing request          method=GET path=/file
```

Figure 12: Example QH Client CLI

4.4 Testing Strategy

The QH protocol implementation is validated using four testing approaches.

4.4.0.1 Unit Tests

Unit tests verify the behavior of individual protocol components, including varint encoding and decoding, header processing, status code mappings, and compression. These tests ensure that core data structures and functions behave correctly under both normal and edge-case conditions.

4.4.0.2 Integration Tests

Integration tests validate complete client–server interaction. They spin up a server, connect a client, and simulate request–response cycles with different message types, header configurations, and payload sizes.

4.4.0.3 Fuzz Testing

Fuzz tests use randomly generated inputs to uncover edge cases and potential crashes in the parsing and message completeness logic. Fuzzing can reveal unexpected input handling issues that are difficult to anticipate manually.

4.4.0.4 Concurrency and Security Tests

All tests run with Go's race detector enabled to detect data races and concurrency bugs in the server's multi-goroutine execution model. Additional security-focused tests validate the protocol's behavior under adversarial conditions such as malformed messages, oversized payloads, and invalid header encodings.

4.4.0.5 Test Organization and Coverage

We organize tests by component, with shared helpers to ensure consistent setup. Coverage is measured using atomic mode to provide accurate results in concurrent code.

4.5 Code Quality and Tooling

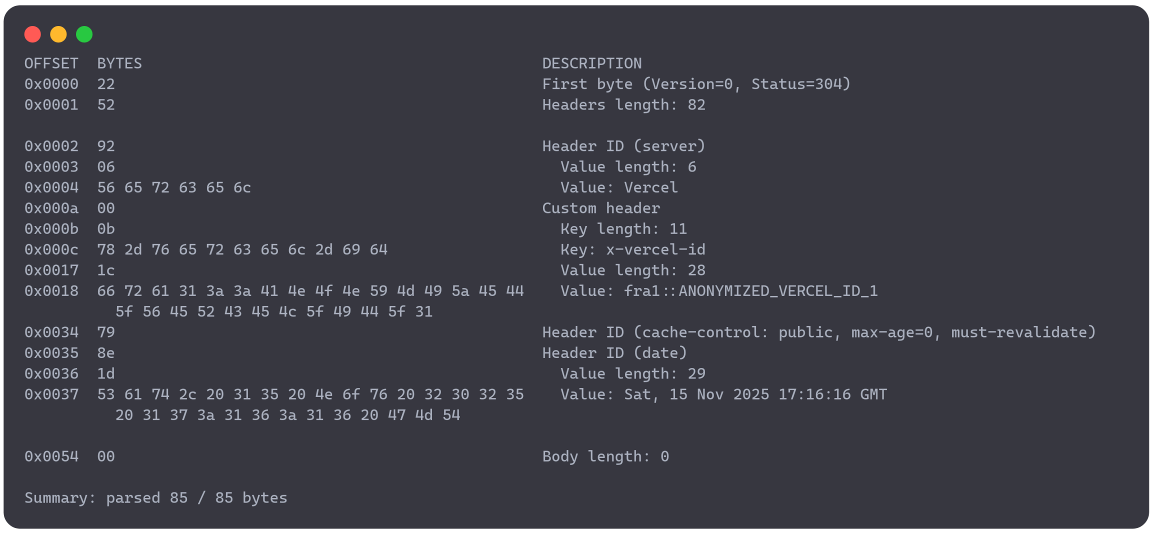
4.5.1 Development Process

The project follows standard Go development practices with `gofmt` for code formatting and `goimports` for automatic import management. Code quality is enforced through an automated CI/CD pipeline that runs various checks on every commit.

4.5.2 Wire Format Debugging Utilities

The implementation includes utilities in `debug.go` that generate human-readable annotated hexdumps of the binary wire format. The `DebugRequest()` and `DebugResponse()` functions parse encoded messages and display byte offsets, hex values, and decoded interpretations in a three-column format. The utilities provide byte-level inspection with field annotations, inline varint decoding, header format identification, and truncation for long fields. They were used to verify encoding correctness and compare wire format efficiency across protocols. The annotated format is also used in the benchmark report (`qh/docs/benchmarks/report.md`) to visualize message sizes.

Example output showing a QH response:

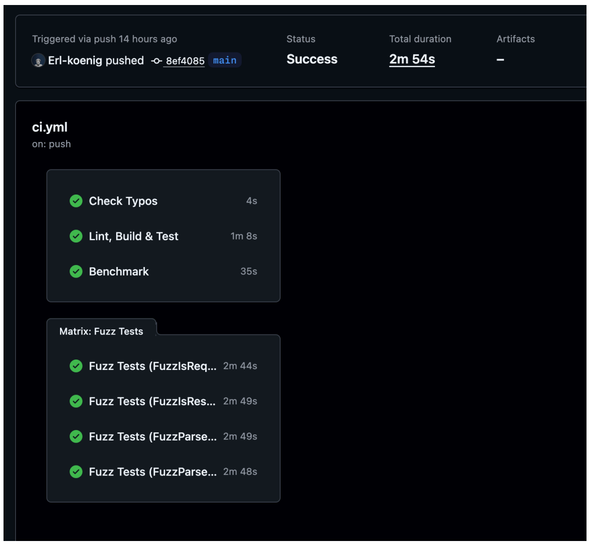


```
OFFSET  BYTES                                           DESCRIPTION
0x0000  22                                              First byte (Version=0, Status=304)
0x0001  52                                              Headers length: 82

0x0002  92                                              Header ID (server)
0x0003  06                                                Value length: 6
0x0004  56 65 72 63 65 6c                                 Value: Vercel
0x000a  00                                              Custom header
0x000b  0b                                                Key length: 11
0x000c  78 2d 76 65 72 63 65 6c 2d 69 64                  Key: x-vercel-id
0x0017  1c                                                Value length: 28
0x0018  66 72 61 31 3a 3a 41 4e 4f 4e 59 4d 49 5a 45 44   Value: fra1::ANONYMIZED_VERCEL_ID_1
          5f 56 45 52 43 45 4c 5f 49 44 5f 31
0x0034  79                                              Header ID (cache-control: public, max-age=0, must-revalidate)
0x0035  8e                                              Header ID (date)
0x0036  1d                                                Value length: 29
0x0037  53 61 74 2c 20 31 35 20 4e 6f 76 20 32 30 32 35   Value: Sat, 15 Nov 2025 17:16:16 GMT
          20 31 37 3a 31 36 3a 31 36 20 47 4d 54

0x0054  00                                              Body length: 0

Summary: parsed 85 / 85 bytes
```

Figure 13: Annotated hexdump of QH response wire format

4.5.3 CI/CD Pipeline

The CI/CD pipeline, defined in `.github/workflows/ci.yml`, runs on every pull request and push. It consists of the following jobs:



Triggered via push 14 hours ago
Erl-koenig pushed 8ef4085 main
Status: Success
Total duration: 2m 54s
Artifacts: –

ci.yml
on: push

Check Typos	4s
Lint, Build & Test	1m 8s
Benchmark	35s

Matrix: Fuzz Tests

Fuzz Tests (FuzzIsReq...	2m 44s
Fuzz Tests (FuzzIsRes...	2m 49s
Fuzz Tests (FuzzParse...	2m 49s
Fuzz Tests (FuzzParse...	2m 48s

Figure 14: GitHub Actions CI/CD Pipeline

- **typos**: Spell checking
- **lint-build-test**: Linting, build, tests, coverage
- **fuzz-tests**: Fuzzing on main branch
- **benchmark**: Runs benchmark suite

4.5.3.1 Typos

The `typos` tool performs spell checking, detecting spelling errors in code comments and documentation. This step runs on every pull request and push.

4.5.3.2 Linting

The project uses `golangci-lint`, a comprehensive aggregator of Go linters, to enforce consistent code quality. It detects a wide range of issues, including potential bugs, performance pitfalls, style inconsistencies, and common security problems. The linter runs on every pull request and push. If any issues are found, the pipeline fails.

4.5.3.3 Build

The build step compiles all example applications. This ensures the code can actually be built and that the examples work with the latest protocol changes. This step runs on every pull request and push.

4.5.3.4 Test

The full test suite (unit, integration, and concurrency tests) runs on every pull request and push with race detection enabled. Coverage results are automatically posted as pull request comments.

4.5.3.5 Fuzz Testing

Fuzzing tests run only on the main branch. These tests use random inputs to find edge cases and potential bugs that might not be found with unit tests. Each fuzz target runs for two minutes. We run the following four fuzz targets:

- `FuzzParseRequest`: Tests request parsing with random inputs.
- `FuzzParseResponse`: Tests response parsing robustness.
- `FuzzIsRequestComplete`: Tests request completeness detection.
- `FuzzIsResponseComplete`: Tests response completeness detection.

4.5.3.6 Benchmark

The benchmark job runs `make run` to validate that the benchmark code remains functional. The output is not saved, but visible in the actions run. This step runs on every pull request and push.

4.5.4 Wireshark Dissector for QOTP / QH

4.5.4.1 Description

To enable deep inspection and debugging of QH protocol traffic, we developed a custom Wireshark dissector that provides real-time decryption and visualization of encrypted QOTP packets. The dissector integrates seamlessly with Wireshark's packet analysis interface, presenting protocol fields in a human-readable format.

4.5.4.2 Motivation

Network protocol development requires visibility into protocol internals for debugging, performance optimization, and validation. While QOTP's encryption ensures security, it also obscures packet contents from network inspectors. A custom dissector bridges this gap, enabling developers to analyze encrypted traffic without compromising the protocol's security design.

4.5.4.3 Architecture

The dissector consists of three additional components and the key logging mechanism in the QOTP implementation that work together to provide live packet analysis:

Lua Dissector Core: Implemented as a Wireshark Lua plugin, the dissector registers with Wireshark to handle QOTP protocol packets. It parses the packet structure, extracts protocol fields, and coordinates with the decryption layer.

Native Decryption Library: The ChaCha20-Poly1305 decryption functionality is implemented as a shared library (DLL on Windows, SO on Linux/macOS) compiled from the QOTP Go codebase. This library exposes decryption functions through a C API.

C Bridge Layer: A thin C wrapper provides the interface between Lua and the Go decryption library. It handles dynamic library loading, function binding, and data marshaling between Lua and native code.

Key Logging Mechanism: The QOTP client implementation includes an optional keylog feature (enabled with build tag `-tags keylog`) that exports session keys to a keylog file. The dissector reads this file to obtain the keys necessary for decrypting captured traffic. The keylog includes both the session keys and the initial connection establishment keys derived from the key exchange.

4.5.4.4 Features

The dissector provides comprehensive protocol analysis capabilities:

- Real-time decryption of QOTP encrypted packets using session keys from the keylog file
- Hierarchical display of protocol layers (QOTP transport, QH application)
- Detailed field breakdown for QH requests and responses including methods, status codes, and headers
- Support for protocol-specific static tables with dynamic lookup from Go library exports

4.5.4.5 Build and Deployment

The dissector build process is fully automated through a GitHub Actions CI/CD pipeline that produces platform-specific releases:

Multi-Platform Build: Using Docker and cross-compilation toolchains (MinGW for Windows, osxcross for macOS), the pipeline builds binaries for Windows, Linux, and macOS from a single codebase.

Automated Releases: Each tagged version automatically generates release artifacts containing the Lua dissector script and platform-specific native libraries, ready for installation in Wireshark.

Dependency Management: The build process uses Wireshark's official Lua SDK for Windows and static Lua linking for Linux/macOS, ensuring compatibility across Wireshark versions.

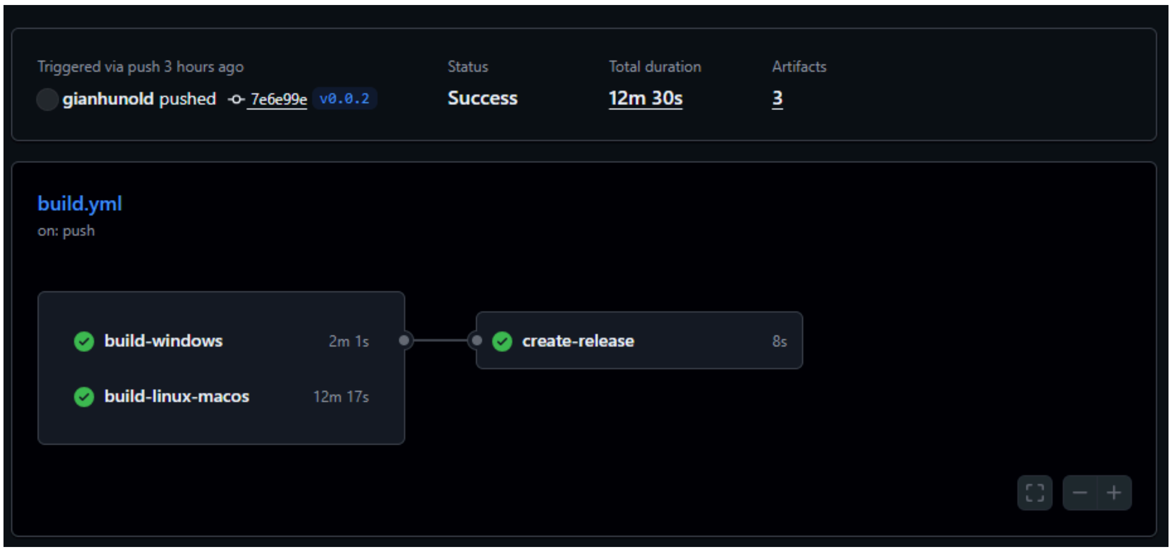
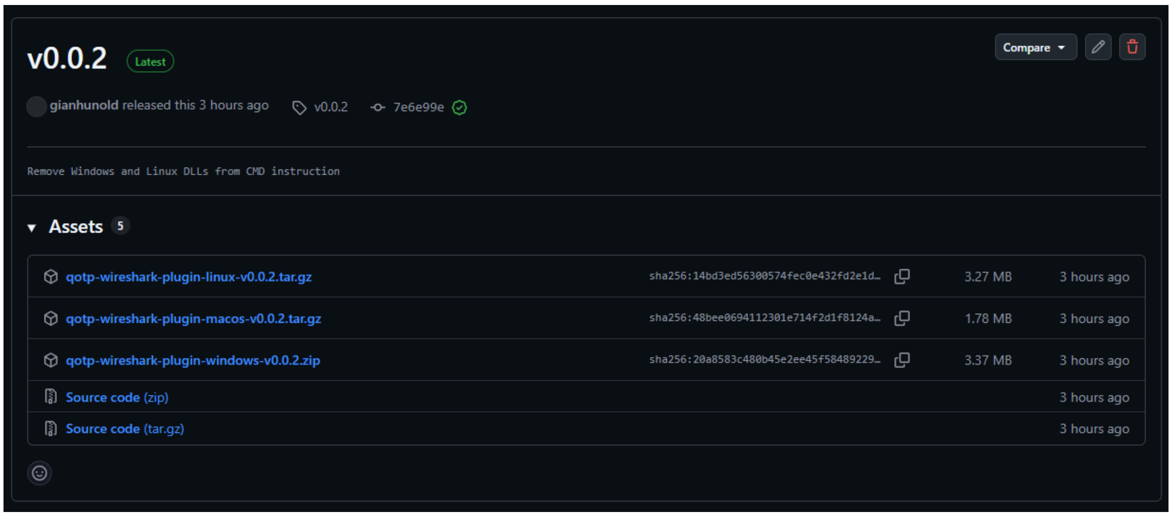


Figure 15: GitHub Actions CI/CD Pipeline for Wireshark Dissector



v0.0.2 Latest

gianhunold released this 3 hours ago v0.0.2 7e6e99e

```
Remove Windows and Linux DLLs from CMD instruction
```

Assets 5

qotp-wireshark-plugin-linux-v0.0.2.tar.gz	sha256:14bd3ed56300574fec0e432fd2e1d…	3.27 MB	3 hours ago
qotp-wireshark-plugin-macos-v0.0.2.tar.gz	sha256:48bee0694112301e714f2d1f8124a…	1.78 MB	3 hours ago
qotp-wireshark-plugin-windows-v0.0.2.zip	sha256:20a8583c480b45e2ee45f58489229…	3.37 MB	3 hours ago
Source code (zip)			3 hours ago
Source code (tar.gz)			3 hours ago

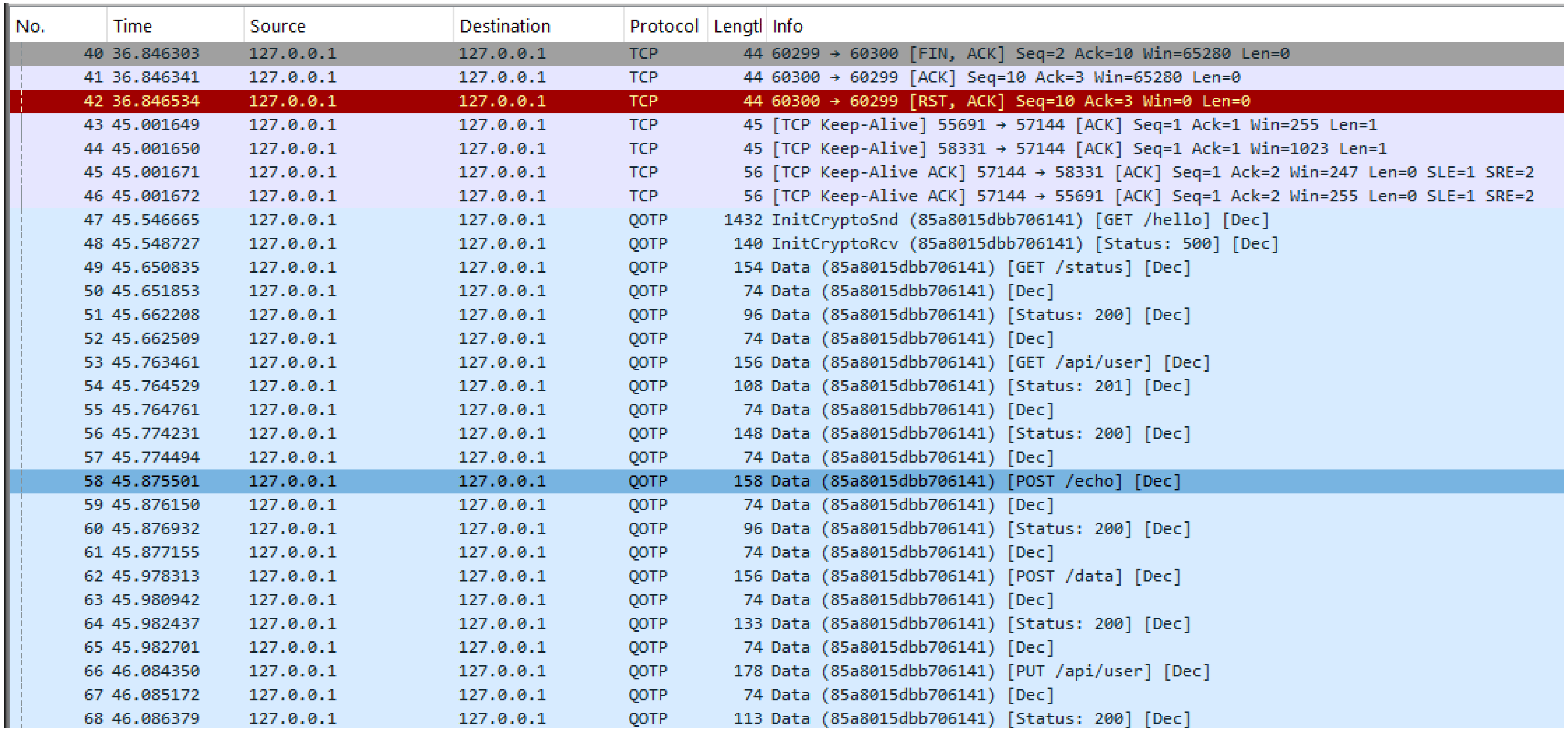
Figure 16: GitHub Releases Page with Dissector Artifacts

4.5.4.6 Impact

The dissector enhances the development workflow by enabling:

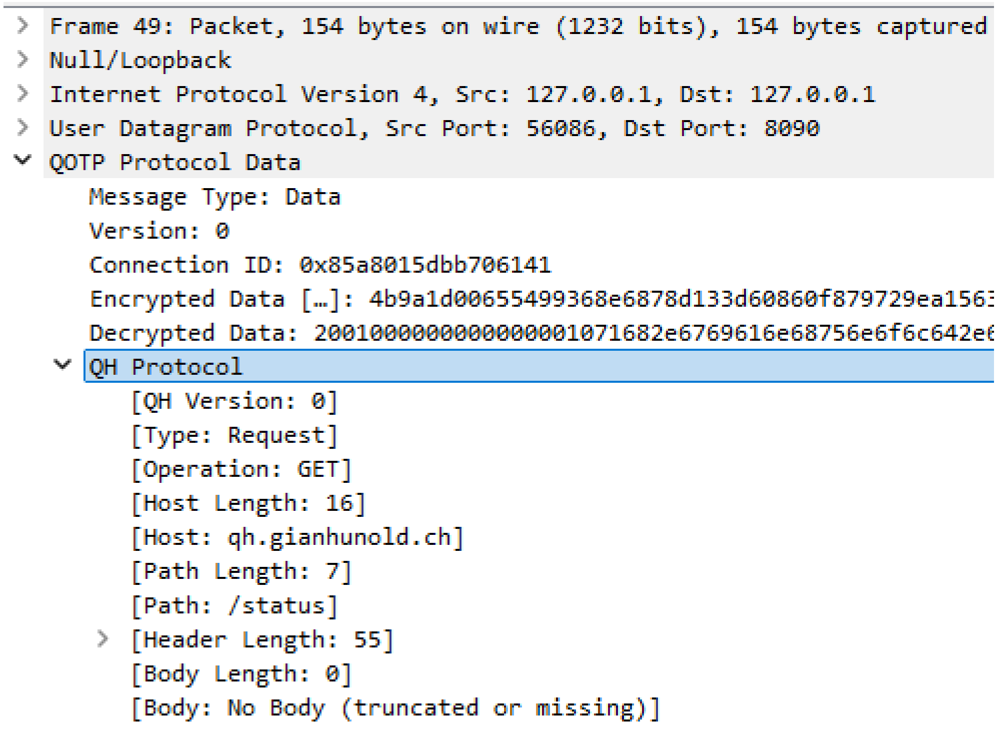
- Visualization of protocol messages and fields in real time
- Quick identification of malformed packets or protocol violations
- Validation of new protocol features and changes
- Efficient debugging of client and server interactions
- Knowledge sharing through visual protocol inspection

4.5.4.7 Impressions



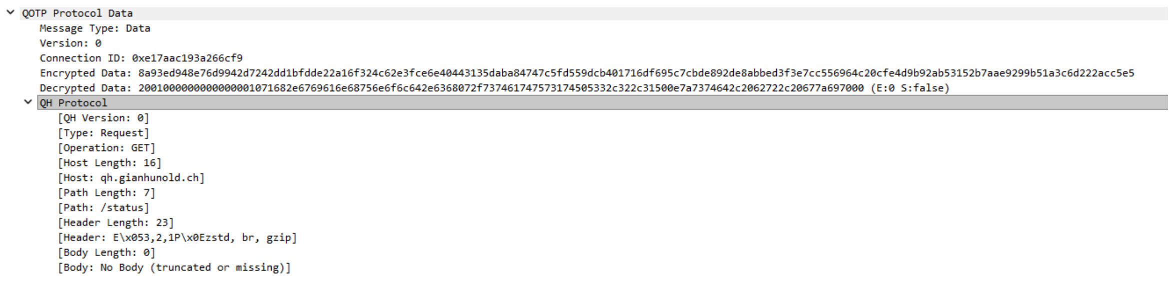
No.	Time	Source	Destination	Protocol	Lengtl	Info
40	36.846303	127.0.0.1	127.0.0.1	TCP	44	60299 → 60300 [FIN, ACK] Seq=2 Ack=10 Win=65280 Len=0
41	36.846341	127.0.0.1	127.0.0.1	TCP	44	60300 → 60299 [ACK] Seq=10 Ack=3 Win=65280 Len=0
42	36.846534	127.0.0.1	127.0.0.1	TCP	44	60300 → 60299 [RST, ACK] Seq=10 Ack=3 Win=0 Len=0
43	45.001649	127.0.0.1	127.0.0.1	TCP	45	[TCP Keep-Alive] 55691 → 57144 [ACK] Seq=1 Ack=1 Win=255 Len=1
44	45.001650	127.0.0.1	127.0.0.1	TCP	45	[TCP Keep-Alive] 58331 → 57144 [ACK] Seq=1 Ack=1 Win=1023 Len=1
45	45.001671	127.0.0.1	127.0.0.1	TCP	56	[TCP Keep-Alive ACK] 57144 → 58331 [ACK] Seq=1 Ack=2 Win=247 Len=0 SLE=1 SRE=2
46	45.001672	127.0.0.1	127.0.0.1	TCP	56	[TCP Keep-Alive ACK] 57144 → 55691 [ACK] Seq=1 Ack=2 Win=255 Len=0 SLE=1 SRE=2
47	45.546665	127.0.0.1	127.0.0.1	QOTP	1432	InitCryptoSnd (85a8015dbb706141) [GET /hello] [Dec]
48	45.548727	127.0.0.1	127.0.0.1	QOTP	140	InitCryptoRcv (85a8015dbb706141) [Status: 500] [Dec]
49	45.650835	127.0.0.1	127.0.0.1	QOTP	154	Data (85a8015dbb706141) [GET /status] [Dec]
50	45.651853	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
51	45.662208	127.0.0.1	127.0.0.1	QOTP	96	Data (85a8015dbb706141) [Status: 200] [Dec]
52	45.662509	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
53	45.763461	127.0.0.1	127.0.0.1	QOTP	156	Data (85a8015dbb706141) [GET /api/user] [Dec]
54	45.764529	127.0.0.1	127.0.0.1	QOTP	108	Data (85a8015dbb706141) [Status: 201] [Dec]
55	45.764761	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
56	45.774231	127.0.0.1	127.0.0.1	QOTP	148	Data (85a8015dbb706141) [Status: 200] [Dec]
57	45.774494	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
58	45.875501	127.0.0.1	127.0.0.1	QOTP	158	Data (85a8015dbb706141) [POST /echo] [Dec]
59	45.876150	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
60	45.876932	127.0.0.1	127.0.0.1	QOTP	96	Data (85a8015dbb706141) [Status: 200] [Dec]
61	45.877155	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
62	45.978313	127.0.0.1	127.0.0.1	QOTP	156	Data (85a8015dbb706141) [POST /data] [Dec]
63	45.980942	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
64	45.982437	127.0.0.1	127.0.0.1	QOTP	133	Data (85a8015dbb706141) [Status: 200] [Dec]
65	45.982701	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
66	46.084350	127.0.0.1	127.0.0.1	QOTP	178	Data (85a8015dbb706141) [PUT /api/user] [Dec]
67	46.085172	127.0.0.1	127.0.0.1	QOTP	74	Data (85a8015dbb706141) [Dec]
68	46.086379	127.0.0.1	127.0.0.1	QOTP	113	Data (85a8015dbb706141) [Status: 200] [Dec]

Figure 17: Wireshark Overview of QOTP Traffic



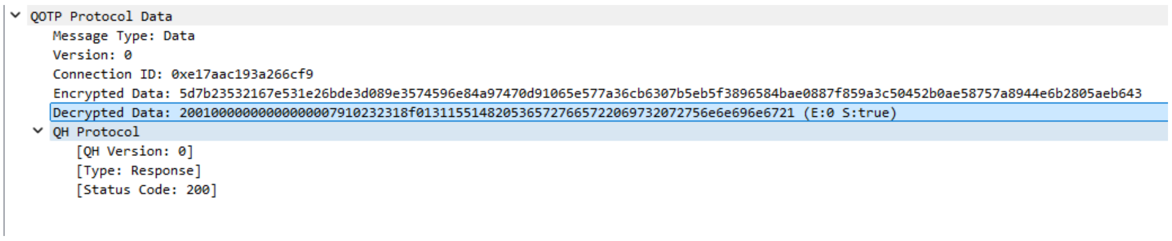
```
> Frame 49: Packet, 154 bytes on wire (1232 bits), 154 bytes captured
> Null/Loopback
> Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
> User Datagram Protocol, Src Port: 56086, Dst Port: 8090
v QOTP Protocol Data
    Message Type: Data
    Version: 0
    Connection ID: 0x85a8015dbb706141
    Encrypted Data […]: 4b9a1d00655499368e6878d133d60860f879729ea156
    Decrypted Data: 200100000000000000001071682e6769616e68756e6f6c642e
  v QH Protocol
      [QH Version: 0]
      [Type: Request]
      [Operation: GET]
      [Host Length: 16]
      [Host: qh.gianhunold.ch]
      [Path Length: 7]
      [Path: /status]
    > [Header Length: 55]
      [Body Length: 0]
      [Body: No Body (truncated or missing)]
```

Figure 18: Wireshark QH Protocol Structure



```
v QOTP Protocol Data
    Message Type: Data
    Version: 0
    Connection ID: 0xe17aac193a266cf9
    Encrypted Data: 8a93ed948e76d9942d7242dd1bfdde22a16f324c62e3fce6e40443135daba84747c5fd559dcb401716df695c7cbde892de8abbed3f3e7cc556964c20cfe4d9b92ab53152b7aae9299b51a3c6d222acc5e5
    Decrypted Data: 200100000000000000001071682e6769616e68756e6f6c642e6368072f737461747573174505332c322c31500e7a7374642c2062722c20677a697000 (E:0 S:false)
  v QH Protocol
      [QH Version: 0]
      [Type: Request]
      [Operation: GET]
      [Host Length: 16]
      [Host: qh.gianhunold.ch]
      [Path Length: 7]
      [Path: /status]
      [Header Length: 23]
      [Header: E\x053,2,1P\x0Ezstd, br, gzip]
      [Body Length: 0]
      [Body: No Body (truncated or missing)]
```

Figure 19: Wireshark Request View with Decrypted QH Protocol



```
v QOTP Protocol Data
    Message Type: Data
    Version: 0
    Connection ID: 0xe17aac193a266cf9
    Encrypted Data: 5d7b23532167e531e26bde3d089e3574596e84a97470d91065e577a36cb6307b5eb5f3896584bae0887f859a3c50452b0ae58757a8944e6b2805aeb643
    Decrypted Data: 2001000000000000000007910232318f0131155148205365727665722069732072756e6e696e6721 (E:0 S:true)
  v QH Protocol
      [QH Version: 0]
      [Type: Response]
      [Status Code: 200]
```

Figure 20: Wireshark Response View with Decrypted QH Protocol

4.5.5 DNS-Based Key Exchange

When QH clients connect to a server, they first resolve the server address to an IP address using DNS. To establish a secure connection, the client must obtain the server's public key, either through a QOTP key exchange handshake or by retrieving it in advance.

4.5.5.1 Motivation

We wanted to explore an alternative way of obtaining the server's public key without needing an additional round-trip for a key exchange handshake. This would reduce latency and improve performance.

4.5.5.2 Approach

We decided to use DNS to obtain the server's public key. DNS is a widely used protocol for resolving domain names to IP addresses, and it can also be used to store additional information such as public keys. Since we use DNS to resolve the server hostname to an IP address, DNS has to work anyway, so we can use DNS for the public key retrieval as well without adding additional round-trips.

We created a new TXT record on the DNS server that stores the server's public key. The TXT entry is called `_qotp`.hostname.domain . The value of this TXT record is the QOTP Version and the server's public key in base64 encoding.

Example:

```
_qotp.example.com. IN TXT "v=0;k=AAAAB3NzaC1yc2EAAAADAQABAAABAQC3..."
```

The client can then perform a DNS lookup for the `_qotp` TXT record then check the version and extract the public key for establishing a secure connection to the server.

The _ prefix is a convention used in DNS to indicate that the record is a service-specific record. This helps to avoid naming conflicts with other DNS records and makes it clear that the record is intended for a specific purpose. Inspired by dmarc records (e.g., `_dmarc.example.com`).

4.6 Implementation Challenges

Several technical challenges emerged during the development of the QH protocol implementation.

Code Duplication and Synchronization: Maintaining consistency across multiple artifacts proved challenging. The Wireshark dissector needed to mirror parts of the protocol implementation logic, requiring manual synchronization when the protocol evolved. The static header table collected via the browser extension needed to be available in multiple formats: `headers.go` for the Go implementation, `headers.md` for documentation, and `headers.json` for other language implementations. To address this, we developed two Python CLI tools as part of the browser extension tooling: `merge_headers.py` to consolidate header data from multiple browsing sessions, and `generate_outputs.py` to export the collected data into all required formats (Go, Markdown, JSON). This eliminated manual synchronization and ensured consistency across all artifacts.

Static Header Table Design: Selecting which headers to include in the static table required real-world traffic analysis using the browser extension. Analyzing and choosing which headers to include in the final table proved to be difficult, as neither Large Language Models nor

CLI scripts could do this reliably. For the analysis, multiple factors had to be considered. Deciding which headers should be included or excluded, detecting vendor-specific headers and adding missing headers key-value pairs manually. This was quite a manual process but the foundation and data stays now and is ready to be extended easily.

Data Anonymization: For both the browser extension and test cases, collecting real-world header data and creating benchmark test cases required careful anonymization to prevent leaking sensitive information. Header values often contain authentication tokens, API keys, session identifiers and other confidential data. All collected data had to be anonymized to replace sensitive header values with mock data.

5 Evaluation

This chapter evaluates QH's wire format efficiency through benchmarks comparing it against HTTP/1.1, HTTP/2, and HTTP/3 on real-world web traffic. Specifically, we measure:

Wire Format Efficiency: How does QH's binary encoding with static header compression compare to HTTP/1.1′s text format, HTTP/2′s HPACK, and HTTP/3′s QPACK in terms of total message size?

Static Table Effectiveness: Can the static header table achieve compression within 15% of dynamic schemes for real-world web traffic?

Best and Worst Case Performance: Under what conditions does QH excel, and where does it underperform relative to other protocols?

5.1 Benchmark Design

This section describes the methodology for comparing QH's wire format efficiency against HTTP/1.1, HTTP/2, and HTTP/3.

5.1.1 Methodology

The benchmarks measure wire format size by encoding identical HTTP semantics (method, path, headers, status) across all four protocols and comparing the resulting byte counts. This isolates the protocol encoding efficiency from transport and network factors.

Measurement Approach:

For each test case containing a request and response:

1. Encode the request using each protocol's wire format
2. Encode the response using each protocol's wire format
3. Sum request and response sizes for total message size
4. Compare sizes across protocols

Controlled Variables:

- Identical semantic content (same method, path, headers)
- Fresh encoder state for each test case (no dynamic table carryover)
- Header-only measurement (bodies excluded, see below)
- Consistent header normalization (lowercase names)

Body Exclusion:

Message bodies are intentionally excluded from the benchmark comparison. Including bodies would bloat message sizes and obscure the header compression differences we aim to measure. More importantly, body compression is protocol-independent: all protocols (QH, HTTP/1.1, HTTP/2, HTTP/3) apply the same content compression algorithms (gzip, brotli, zstd) to bodies through Content-Encoding negotiation. Body compression effectiveness depends on the compression algorithm and content type, not the protocol's wire format. By measuring headers only, we isolate the differentiating factor between protocols: how efficiently each encodes HTTP metadata.

Important Limitation:

The benchmarks use fresh encoders for each request response pair, meaning HTTP/2 and HTTP/3′s dynamic header tables start empty for every test case. This isolates per-message encoding efficiency but understates advantages in long-lived connections. Implications are discussed in Section 5.3.3

5.1.2 Test Scenarios

The benchmark suite comprises 110 test cases in two categories.

5.1.2.1 Test Case Structure

Each test case is defined in JSON format with the following structure:

```json
{
  "name": "Edge Case 1: QH Best Case - All Static Table Complete Pairs",
  "description": "Maximum QH efficiency: all headers use Format 1",
  "request": {
    "method": "GET",
    "host": "api.example.com",
    "path": "/resource",
    "headers": {
      "accept": "*/*",
      "accept-encoding": "gzip, deflate, br, zstd",
      "accept-language": "en-US,en;q=0.5",
      "content-type": "application/json"
    }
  },
  "response": {
    "statusCode": 200,
    "headers": {
      "content-type": "application/json; charset=utf-8",
      "content-encoding": "gzip",
      "cache-control": "public, max-age=7200"
    }
  }
}
```

Each test case is encoded into HTTP/1.1, HTTP/2, HTTP/3, and QH wire formats.

5.1.2.2 Edge Cases (10 test cases)

Manually constructed test cases designed to test protocol boundaries and highlight performance extremes:

Test Case	Purpose
QH Best Case	All headers match static table complete pairs (Format 1)
QH Worst Case	All custom headers requiring Format 3 encoding
HTTP/1.1 Best Case	Minimal headers favoring text encoding
QH Format 2	Static names with custom values
QH Mixed Formats	Combination of all three header encoding formats
Many Headers	Test with numerous headers
Minimal Response	204 No Content with few headers
Verbose Headers	Large header values
Large Single Header	Content-Security-Policy header
Empty Values	Headers present but with empty values

Table 1: Edge Case Test Scenarios

These edge cases intentionally favor different protocols to establish performance bounds.

5.1.2.3 Real HTTP Traffic (100 test cases)

Captured from actual web browsing sessions to represent realistic usage patterns. The test data was collected using Chrome DevTools' HAR (HTTP Archive) export feature during typical browsing activities.

Collection Details:

- Collection date: 15.11.2025

Total request/response pairs captured: 921

- Selected request-response pairs: 100

Data Sources:

- Video streaming (YouTube)
- Code hosting and collaboration (GitHub)
- Search engines (Google)
- Developer documentation sites
- Social media platforms

- Content delivery networks

Content Types Represented:

- JSON API responses
- JavaScript and CSS assets
- HTML documents
- Image resources
- Font files (filtered)
- Tracking requests (filtered)

Data Processing:

We processed the raw HAR files using a Python script (`benchmark/cmd/generate-testcases/main.py`) that performs stratified sampling to ensure representative content type distribution:

1. Filters non-essential resources (fonts, tracking pixels, ads)
2. Categorizes requests by content type (JSON 30%, JavaScript 25%, images 15%, HTML 10%, CSS 10%, other 10%)
3. Selects 100 request-response pairs using stratified sampling with fixed random seed (RANDOM_SEED=42) for reproducibility
4. Extracts headers only (bodies excluded for fair comparison)
5. Normalizes header names to lowercase
6. Outputs standardized JSON format

Since HAR files contain sensitive data such as authentication tokens, API keys, and session identifiers, and the benchmark data is included in the public repository, we manually anonymized all header values with realistic random values.

5.1.3 Compared Protocols

Each protocol is encoded using the following Go implementations:

Protocol	**Implementation**	**Compression**
HTTP/1.1	`net/http` standard library	None (text format)
HTTP/2	`golang.org/x/net/http2`	HPACK with empty dynamic table
HTTP/3	`github.com/quic-go/qpack`	QPACK with empty dynamic table
QH	`github.com/qo-proto/qh`	Static header table only

Table 2: Protocol Implementations Used for Benchmarking

All implementations use their default settings without custom tuning.

5.1.4 Metrics

The primary metric is **wire format size** measured in bytes:

- **Request size:** Encoded method, host, path, and headers

- **Response size:** Encoded status code and headers
- **Total size:** Request size + Response size

Secondary metrics include:

- **Compression ratio:** QH size relative to each comparison protocol (expressed as percentage)
- **Size category breakdown:** Performance grouped by total message size (tiny under 1KB, small 1-10KB)
- **Performance bounds:** Best and worst case ratios for each protocol pair

5.1.5 Environment

Benchmarks were executed with the following configuration:

Parameter	Value
Generated	December 4, 2025
Go Version	go1.25.1
Platform	Darwin/x86_64
Test Cases	110 (10 edge cases + 100 Real HTTP Traffic)

Table 3: Benchmark Environment Configuration

The benchmark tool (`qhbench`) is deterministic and produces identical results across runs given the same test data.

5.2 Results

This section presents the benchmark results organized by test category and analysis type.

5.2.1 Overall Summary

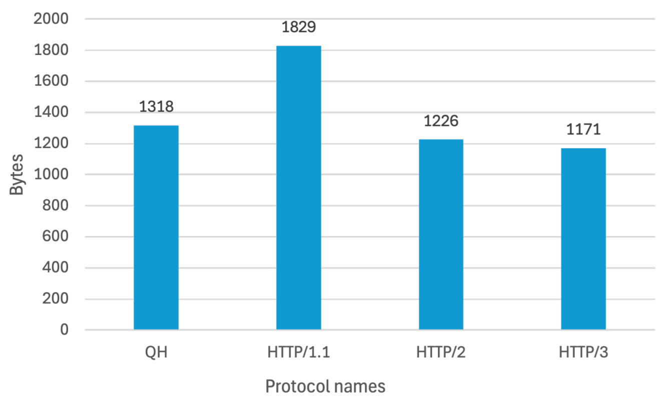


Figure 21: Average wire format size per test case across all protocols

QH achieves 38.8% size reduction compared to HTTP/1.1 while remaining within 11% of HTTP/3′s wire format efficiency. HTTP/2 and HTTP/3′s dynamic compression provides measurable benefits even with fresh encoder state.

5.2.2 Edge Case Analysis

The 10 edge cases establish performance bounds for each protocol. Table 4 shows the aggregate results.

Protocol	**Total Bytes**	**vs QH**	**Avg per Test**
QH	4,756	baseline	476 B
HTTP/1.1	7,044	48.1% larger	704 B
HTTP/2	4,442	6.6% smaller	444 B
HTTP/3	4,094	13.9% smaller	409 B

Table 4: Edge Case Wire Format Size Comparison (10 Test Cases)

5.2.2.1 Performance Bounds vs HTTP/1.1

- **Best case:** 92.0% smaller - QH: 49 B vs HTTP/1.1: 610 B (Edge Case 1: QH Best Case with all static table complete pairs)
- **Worst case:** 11.6% smaller - QH: 1,062 B vs HTTP/1.1: 1,201 B (Edge Case 8: Verbose Headers)

QH consistently outperforms HTTP/1.1 across all test cases.

5.2.2.2 Performance Bounds vs HTTP/2

- **Best case:** 85.5% smaller - QH: 49 B vs HTTP/2: 337 B (Edge Case 1: QH Best Case)
- **Worst case:** 29.2% larger - QH: 1,062 B vs HTTP/2: 822 B (Edge Case 8: Verbose Headers)

When static table coverage is high, QH outperforms HTTP/2 by up to 85%. With verbose or custom headers, HTTP/2′s Huffman encoding provides advantages.

5.2.2.3 Performance Bounds vs HTTP/3

- **Best case:** 82.1% smaller - QH: 49 B vs HTTP/3: 274 B (Edge Case 1: QH Best Case)
- **Worst case:** 35.5% larger - QH: 858 B vs HTTP/3: 633 B (Edge Case 9: Large Single Header)

Similar pattern to HTTP/2, with HTTP/3′s QPACK providing additional efficiency for large header values.

5.2.2.4 Detailed Edge Case Results

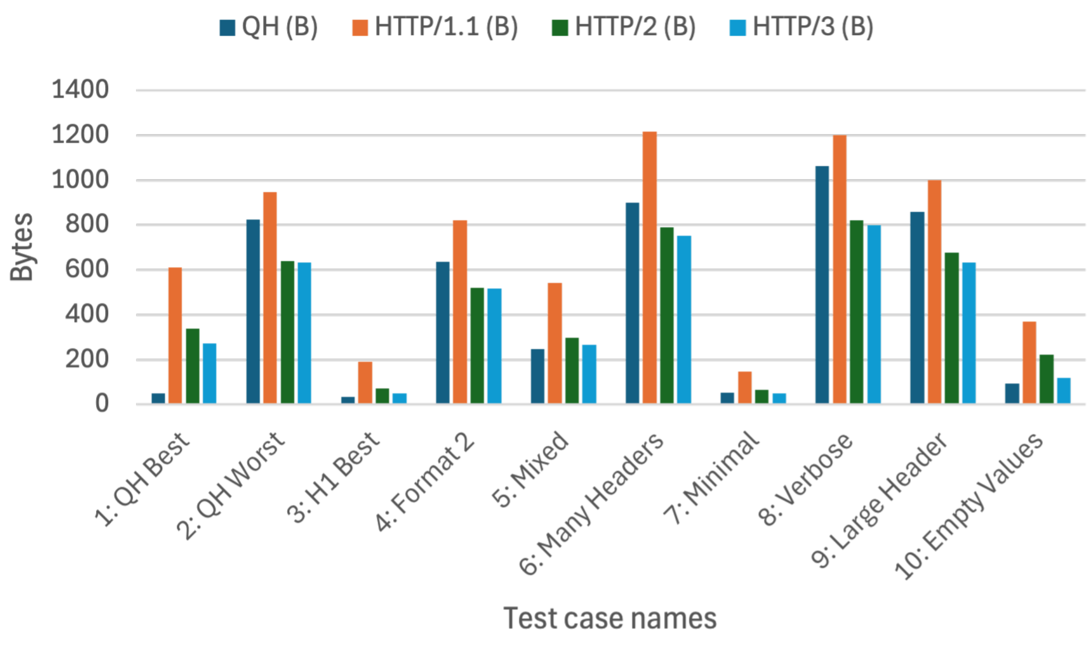


Figure 22: Edge case performance comparison across all protocols

Edge Cases 1, 3, 5, 7, and 10 show QH's strengths: high static table coverage with few custom headers. Edge Cases 2, 4, 8, and 9 reveal QH's weaknesses: custom headers and large header values where HTTP/2 and HTTP/3′s compression excels.

5.2.3 Real HTTP Traffic Analysis

The 100 real-world test cases provide insight into QH's performance on actual web traffic. Table 5 shows the aggregate results.

Protocol	Total Bytes	vs QH	Avg per Test
QH	140,195	baseline	1,402 B
HTTP/1.1	194,200	38.5% larger	1,942 B
HTTP/2	130,373	7.0% smaller	1,304 B
HTTP/3	124,714	11.0% smaller	1,247 B

Table 5: Real Traffic Wire Format Size Comparison (100 Test Cases)

5.2.3.1 Performance Bounds on Real Traffic

vs HTTP/1.1:

- Best case: 51.5% smaller (Request 93: OPTIONS /models)
- Worst case: 7.9% smaller (Request 1: GET /manifest.json)

vs HTTP/2:

- Best case: 23.6% smaller (Request 93: OPTIONS /models)
- Worst case: 33.7% larger (Request 1: GET /manifest.json)

vs HTTP/3:

- Best case: 18.2% smaller (Request 31: GET /)
- Worst case: 37.0% larger (Request 1: GET /manifest.json)

5.2.4 Size Category Analysis

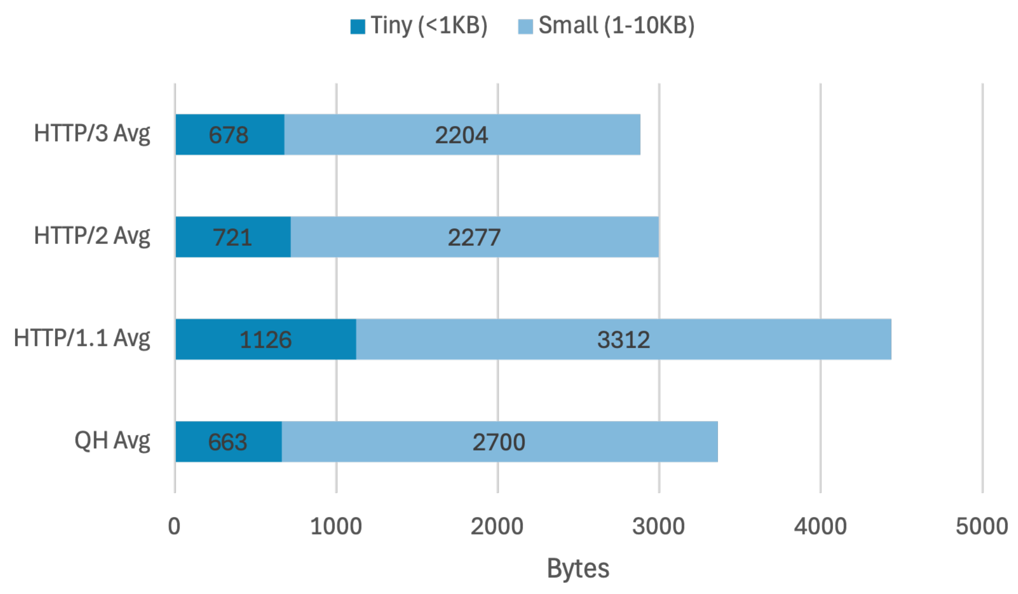


Figure 23: QH performance vs message size showing optimal performance for messages under 1KB

QH shows its strongest performance in the "Tiny" category, achieving 58.9% of HTTP/1.1′s size and 91.8% of HTTP/2′s size. For larger messages in the "Small" category, HTTP/2 and HTTP/3 pull ahead due to improved compression of repeated header patterns and large values.

5.2.5 Request vs Response Breakdown

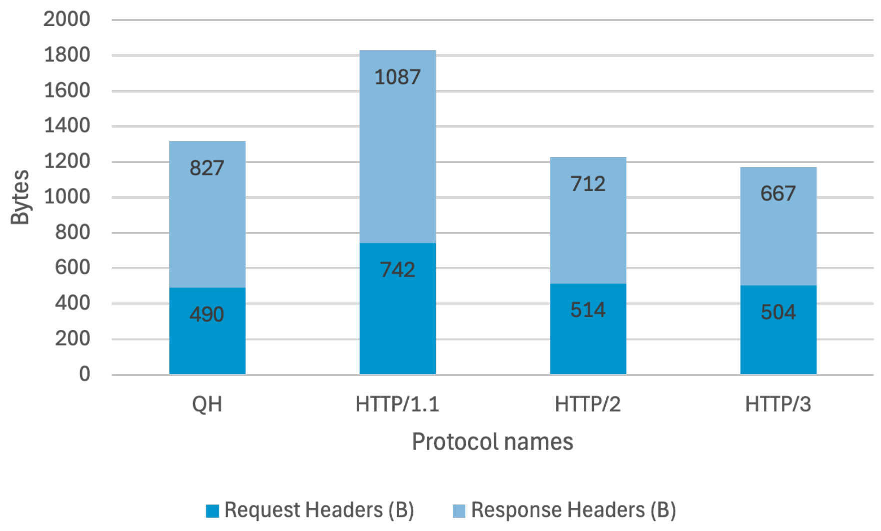


Figure 24: Request and response header size breakdown showing QH's differential performance

QH performs well on request headers, achieving 3-5% smaller encoding than HTTP/2 and HTTP/3. However, response headers show a larger gap: HTTP/2 achieves 14% smaller responses than QH, and HTTP/3 achieves 19.4% smaller responses. This is due to large, repetitive response headers (Content-Security-Policy, Cache-Control variants) that benefit from dynamic compression and Huffman encoding.

5.2.6 Wire Format Examples

To illustrate the encoding differences, we examine three representative test cases with annotated wire format breakdowns.

5.2.6.1 Example 1: GET /manifest.json (Request 1)

This GitHub request demonstrates a header-heavy scenario with a large Content-Security-Policy response header.

Component	QH	H1	H2	H3
Request	280 B	397 B	275 B	273 B
Response	3,916 B	4,160 B	2,864 B	2,789 B
Total	4,196 B	4,557 B	3,139 B	3,062 B

Table 6: Example 1: GET /manifest.json Size Comparison

The QH request encoding demonstrates efficient static table usage:

```
0x00                           Method byte (GET, version 0)
0x0A                           Host length: 10
```

```
github.com                    Host
0x0E                          Path length: 14
/manifest.json                Path
0xFA 0x01                     Headers length: 250 (varint)
0x44 0x41 "Chromium;v=..."    sec-ch-ua (Format 2)
0x41 0x75 "Mozilla/5.0..."    user-agent (Format 2)
0x01                          sec-ch-ua-mobile: ?0 (Format 1)
0x3D                          sec-ch-ua-platform: "macOS" (Format 1)
...
0x00                          Body length: 0
```

The response is dominated by a 3,511-byte Content-Security-Policy header, which QH encodes using Format 2 (static name, custom value). HTTP/2 and HTTP/3 achieve smaller encoding through Huffman compression.

5.2.6.2 Example 2: POST /github-copilot/chat/token (Request 2)

This API request shows mixed header formats with authentication-related custom headers.

Component	**QH**	**H1**	**H2**	**H3**
Request	443 B	792 B	534 B	514 B
Response	3,892 B	4,215 B	2,889 B	2,820 B
Total	4,335 B	5,007 B	3,423 B	3,334 B

Table 7: Example 2: POST /github-copilot/chat/token Size Comparison

The request uses several custom headers (github-verified-fetch, x-github-client-version, x-fetch-nonce) that require Format 3 encoding with both key and value length prefixes.

5.2.6.3 Example 3: Next.js Data Request (Request 3)

This Next.js data request demonstrates a 304 Not Modified response with minimal headers.

Component	**QH**	**H1**	**H2**	**H3**
Request	418 B	696 B	460 B	442 B
Response	85 B	176 B	103 B	101 B
Total	503 B	872 B	563 B	543 B

Table 8: Example 3: GET docs.json with 304 Response

The compact 304 response showcases QH’s efficiency for minimal responses:

```
0x22                          Status byte (304, version 0)
0x52                          Headers length: 82
0x92 0x06 "Vercel"            server (Format 2)
0x00 0x0B "x-vercel-id"...    Custom header (Format 3)
```

```
0x79                          cache-control (Format 1)
0x8E 0x1D "Sat, 15 Nov..."    date (Format 2)
0x00                          Body length: 0
```

5.3 Analysis and Interpretation

This section interprets the benchmark results in context of the research questions and design goals.

5.3.1 Performance Analysis

5.3.1.1 Where QH Excels

QH demonstrates strong performance when header patterns align with its static table design. The protocol achieves optimal results with **high static table coverage** (Edge Case 1: 49B vs 610B for HTTP/1.1) and **small API requests** in the “Tiny” category (58.9% of HTTP/1.1′s size, 91.8% of HTTP/2).

The key insight is that QH’s static table was empirically designed for client-generated headers, resulting in **request headers that are 3-5% smaller than HTTP/2 and HTTP/3**. This makes QH particularly suited for API-heavy applications where request headers dominate, such as REST and GraphQL services with minimal response metadata. Status-code-only responses (204 No Content, 304 Not Modified) further benefit from QH’s low per-header overhead.

5.3.1.2 Where QH Struggles

QH underperforms in scenarios favoring dynamic compression. **Large headers** like Content-Security-Policy (spanning thousands of bytes) benefit from HTTP/2 and HTTP/3′s Huffman encoding, while QH stores header values uncompressed. **Custom headers** not in the static table require Format 3 encoding with both key and value length prefixes, adding overhead that dynamic tables would eventually eliminate through learning. This creates a **14-19% gap on response headers**, which are dominated by site-specific security policies, cache configurations, and server headers that vary across deployments. The static table cannot anticipate this server-side diversity, whereas HTTP/2 and HTTP/3′s dynamic tables adapt to each server’s patterns.

5.3.2 Comparison to Research Questions

RQ1: Can a simplified protocol achieve wire format efficiency within 10-15% of HTTP/2 and HTTP/3?

Yes, with qualifications. QH achieves 38.8% smaller wire format than HTTP/1.1 across real traffic, validating that binary encoding with static header tables provides measurable benefits. Against HTTP/2 and HTTP/3, QH is 7-11% larger overall, a reasonable trade-off for reduced implementation complexity. The evaluation supports that QH’s wire format efficiency is sufficient for API traffic and small web requests, where the implementation simplicity benefits (stateless debugging, no synchronization, smaller codebase) may outweigh the modest efficiency loss for applications prioritizing maintainability.

RQ2: Does static header encoding achieve efficiency within 15% of dynamic compression?

For request headers, yes. QH outperforms HTTP/2 and HTTP/3. For response headers, the 14-19% gap suggests dynamic compression provides meaningful benefits for typical server response patterns. The typical case (real traffic average) shows QH within 7-11% of dynamic compression schemes, confirming that static header tables achieve efficiency within this range for common scenarios while accepting trade-offs in edge cases.

RQ3: Can standard network analysis tools like Wireshark provide sufficient protocol visibility, reducing reliance on protocol implementation logging for debugging?

Yes. The Wireshark dissector implementation (Section 4.5.4) demonstrates that network-level analysis can replace protocol implementation logging for debugging. The dissector decrypts QOTP transport layer packets and decodes QH application layer traffic in real-time, displaying connection state, methods, status codes, headers, and bodies. Unlike HTTP/2 and HTTP/3 where dynamic compression tables require connection-level state tracking, QH's stateless design allows packets to be decoded independently. Combined with keylog export for session keys, this provides complete visibility into encrypted traffic directly from network captures, eliminating the need for slog or printf-style debugging in the protocol implementation. Developers can analyze protocol behavior using familiar Wireshark workflows rather than parsing scattered log output.

5.3.3 Limitations

5.3.3.1 Internal Validity

Fresh Encoder State: Using fresh encoders for each test case prevents HTTP/2 and HTTP/3 from benefiting from dynamic table accumulation. The evaluation focuses on per-message encoding efficiency rather than connection-level optimization.

Test Case Selection: The 100 Real HTTP Traffic samples were collected from specific browsing sessions and may not represent all web traffic patterns. The samples span diverse sources including video streaming, code hosting, search engines, and documentation sites.

5.3.3.2 External Validity

Header-Only Measurement: Bodies are excluded from comparison. QH's body handling is identical to other protocols (length-prefixed bytes), so this focuses evaluation on the differentiating feature (header encoding). However, overall message efficiency depends on body size distribution.

Go Implementation Only: Results may differ with other language implementations due to library differences, encoder optimizations, or configuration options.

5.3.3.3 Construct Validity

Wire Size vs Performance: The evaluation measures wire format size, not encoding/ decoding speed, memory usage, or CPU cost. QH's simpler format likely provides encoding speed advantages not captured here.

5.4 Summary

The evaluation confirms that QH achieves its target wire format efficiency through static header compression. QH outperforms HTTP/1.1 by 38.8% while staying within 7-11% of HTTP/2 and HTTP/3 despite lacking dynamic compression. Static tables prove most effective for request headers and small API-style traffic, while response headers show the expected trade-off between simplicity and maximum compression.

6 Discussion

This chapter reflects on key findings from the evaluation and discusses implications for protocol design and future work.

6.1 Design Choices

6.1.1 Number of Headers Field

We considered two possible formats for the header field:

- Format 1 `Number of Headers`:

  `<varint:numHeaders> [header1] [header2] [header3] <varint:bodyLen> <body>`

The advantages are it is simple to parse sequentially and one knows exactly how many headers to expect. The disadvantages are that one must parse all headers before reaching the body and cannot start processing the body until all headers are complete.

- Format 2 `Total Header Length`:

  `<varint:totalHeaderLen> [all headers] <varint:bodyLen> <body>`

The advantages are that one can skip directly to the body after reading the total header length (offset = 1 + varintSize + totalHeaderLen), enabling parallel parsing of headers and body. The disadvantages are: more complex encoding, loses the "how many headers" information (could still be counted while parsing).

6.1.2 Common Prefixes

A further optimization could be to include common prefixes and patterns for custom header-names into the static table. The proposed optimization would be to map the `X-` prefix which appears quite often and other common patterns to an ID `0xNN`.

While this approach would save additional bytes, it does not fit the design philosophy of QH and its simplicity.

Prefix compression introduces complexity:

- Parser needs to handle prefix reconstruction
- Encoder needs prefix-matching logic
- The addition would be a step toward HPACK-/QPACK-style complexity (the goal was to remove it)

6.1.2.1 OpenZL Evaluation

We evaluated OpenZL [11], [12], [13], Meta's format-aware compression framework released in 2025, for potential application to QH header compression. OpenZL achieves lossless compression by treating data structure as an explicit parameter, applying transformations like structure-of-arrays conversion before compression. On the Silesia SAO benchmark (star catalog records), OpenZL demonstrates 2.06x compression ratio at 340 MB/s throughput, outperforming both zstd and xz.

However, OpenZL's design targets large-scale structured datasets rather than protocol headers. The framework excels with megabyte-scale data containing thousands of similar records, where structure-aware transformations provide compression benefits. QH headers typically span 20-200 bytes, orders of magnitude below OpenZL's effective operating range.

Several factors make OpenZL unsuitable for QH. First, the framework requires offline training on representative data to generate compression plans, adding deployment complexity. Second, compression plans must be distributed to all endpoints and updated as traffic patterns evolve, creating a management burden. Third, the transformation graph execution overhead likely exceeds compression benefits for sub-100-byte messages. Fourth, OpenZL's structure-of-arrays optimizations are most effective on batches of similar messages, whereas QH processes independent request-response pairs.

The static header table approach aligns with QH's design philosophy. It achieves comparable size reduction without runtime complexity, requires no training or distribution infrastructure, and provides deterministic encoding suitable for debugging. OpenZL solves an important problem for data warehouses and ML pipelines processing structured bulk data, but represents over-engineering for protocol header compression.

6.2 Challenges

In order to deploy QH in real-world scenarios, various challenges arise. HTTP benefits from decades of tooling and libraries. Integrating QH into current browsers like Chrome or Firefox requires substantial effort and is unlikely to be adopted.

6.3 Future Ideas

Working on the QH protocol surfaced several ideas for potential future features and enhancements that could further improve its performance, usability, and security.

Some ideas that sparked up during the project were already implemented (such as the Wireshark dissector) while others remain as Ideas for future exploration:

6.3.1 Protocol Implementation Improvements

The current protocol implementation could be extended in several areas.

- The implementation could be more robust by adding Go context support, allowing methods like `RequestWithContext(ctx context.Context, method Method, ...)`. This would enable proper request cancellation and timeout handling.
- In the current state of QH, the headers are encoded using a static table with 1-byte numeric IDs for common headers and content types. However, we do not need the full 256 values that one byte provides. We only need about 70 characters (a-z, A-Z, 0-9, and some special characters). If we encode the headers with an encoding not utilizing the whole byte, we could save additional bytes per header. For example we could encode them in base85.
- To mitigate abuse and denial-of-service attacks, a proof-of-work mechanism could be integrated into QH at the protocol level. Real-world examples like Cloudflare's challenge system [14], [15] demonstrate this approach: rather than having the web server issue challenges, clients would be required to solve a computationally intensive task before sending requests. This protocol-level approach would reduce server load and make it more difficult for attackers to launch denial-of-service attacks by deterring excessive traffic from malicious actors.

6.3.2 Ecosystem Development

Broader adoption would require client libraries in popular languages, developer tooling (wire format visualizers, compression calculators, benchmarking tools), and demonstration implementations. Browser support (potentially starting with smaller browsers like Ladybird or Servo) would enable real-world testing and lower the adoption barrier. Example applications (such as SPAs) could demonstrate QH's benefits in practical scenarios.

6.3.3 Latency Benchmarks

The current evaluation focuses on wire format efficiency (message size), but QH's simplified design likely provides advantages in encoding and decoding latency. Future benchmarks could measure the time required to encode and decode requests and responses across QH, HTTP/1.1, HTTP/2, and HTTP/3. This would show QH's potential strength: reduced computational overhead from eliminating dynamic compression tables, Huffman encoding, and complex state management. Such benchmarks would measure end-to-end processing latency to show message processing speed.

6.3.4 Real World Testing

Comprehensive real-world measurements across diverse network conditions would provide deeper insights into QH's practical performance. Testing under varying latencies (1ms to 500ms), packet loss rates (0% to 10%), and bandwidth constraints would reveal how QH responds to real network scenarios compared to HTTP/2 and HTTP/3. Additionally, parameter tuning based on empirical data (such as optimal stream window sizes, timeout values, and congestion control thresholds) could improve performance for specific network profiles.

Such measurements would validate that QH's simplified design maintains reasonable performance across real-world conditions rather than on idealized benchmarks, providing confidence for adoption in production scenarios.

6.4 Summary

QH demonstrates that static header tables achieve efficiency within 7-11% of dynamic compression schemes while eliminating state management complexity. The protocol suits API workloads with small, consistent requests but shows limitations for content-heavy scenarios.

7 Conclusion

Modern web protocols have evolved toward increasing complexity. HTTP/2 and HTTP/3 reduce wire format overhead through dynamic compression tables and complex state management. This thesis explored whether a deliberately simplified protocol can achieve performance within 10-15% of these protocols.

QH demonstrates that simplicity and performance are not mutually exclusive. Using static header compression designed through empirical analysis, QH achieves 38.8% smaller wire format than HTTP/1.1 while remaining within 7-11% of HTTP/2 and HTTP/3. This gap quantifies the cost of simplicity: modest efficiency loss for eliminating dynamic state synchronization.

The evaluation reveals nuanced trade-offs. QH matches HTTP/2 and HTTP/3 on request headers (within 5%) but falls behind on response headers (14-19%). Performance varies by workload: QH excels at small messages under 1KB, making it suitable for API workloads, but struggles with larger messages.

Static header compression remains viable when guided by empirical analysis of real-world traffic patterns. The core finding is that accepting modest efficiency loss eliminates implementation complexity, a worthwhile trade-off for many applications.

With the Wireshark dissector, QH enables straightforward debugging and analysis using standard network tools rather than protocol implementation logging. Combining keylog export with the dissector provides end-to-end visibility into encrypted traffic, making it practical to validate protocol behavior and detect interoperability issues directly from network captures.

Protocol design involves deliberate trade-offs rather than inevitable complexity. Data-driven choices enable simplified protocols.

As a research prototype, QH demonstrates these principles but requires further development for real-world use.

Glossary

***HPACK* – HTTP/2 Header Compression**: Stateful header compression for HTTP/2 using static/dynamic tables and Huffman encoding.

***HTTP* – Hypertext Transfer Protocol**: The foundational application protocol of the World Wide Web, used for transmitting hypermedia documents. Has evolved through multiple versions (HTTP/1.1, HTTP/2, HTTP/3).

Huffman encoding: A variable-length encoding algorithm that assigns shorter codes to more frequent symbols.

***LEB128* – Little Endian Base 128**: A variable-length encoding for integers where each byte uses 7 bits for data and 1 bit to indicate if more bytes follow.

***PKI* – Public Key Infrastructure**: A framework for managing digital certificates and public-key encryption, typically involving Certificate Authorities and certificate validation.

***QH* – Quite Ok HTTP Protocol**: A simplified HTTP-like application protocol designed for reduced complexity while maintaining practical performance. QH runs on top of QOTP and uses static header compression.

***QOTP* – Quite Ok Transport Protocol**: A UDP-based transport protocol that takes an "opinionated" approach, similar to QUIC but with a focus on providing reasonable defaults rather than many options. The goal is to have lower complexity, simplicity, and security, while still being reasonably performant.

***QPACK* – QUIC Header Compression**: Header compression for HTTP/3 adapted from HPACK to handle out-of-order stream delivery.

QUIC: A UDP-based transport protocol developed by Google and standardized as RFC 9000. Provides 0-RTT connection establishment, built-in encryption, and stream multiplexing.

***RTT* – Round Trip Time**: The time required for a network packet to travel from source to destination and back, measured in milliseconds.

Static header table: A predefined, fixed mapping of common headers to numeric identifiers. The table never changes during connection lifetime.

***TLS* – Transport Layer Security**: A cryptographic protocol for secure communication over networks. Commonly used with HTTP (HTTPS) and requires X.509 certificates issued by Certificate Authorities.

***Varint* – Variable-length Integer**: An encoding scheme that uses a variable number of bytes to represent integers, with smaller values using fewer bytes.

***0-RTT* – Zero Round-Trip Time**: A connection establishment technique where the client can send application data in the first packet without waiting for a handshake response.

Bibliography

[1] R. Fielding, M. Nottingham, and J. Reschke, “HTTP/1.1,” Jun. 2022. Accessed: Sep. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9112.html

[2] M. Thomson and C. Benfield, “HTTP/2,” Jun. 2022. Accessed: Sep. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9113.html

[3] R. Peon and H. Ruellan, “HPACK: Header Compression for HTTP/2,” May 2015. Accessed: Oct. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7541.html

[4] M. Bishop, “HTTP/3,” Jun. 2022. Accessed: Sep. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9114.html

[5] J. Iyengar and M. Thomson, “QUIC: A UDP-Based Multiplexed and Secure Transport,” May 2021. Accessed: Sep. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9000.html

[6] C. Krasic, M. Bishop, and A. Frindell, “QPACK: Field Compression for HTTP/3,” Jun. 2022. Accessed: Oct. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc9204.html

[7] D. Szablewski, “The Quite Ok Image Format.” Accessed: Oct. 2025. [Online]. Available: https://qoiformat.org/qoi-specification.pdf

[8] Google LLC, “gRPC: A high performance, open source universal RPC framework.” Accessed: Oct. 2025. [Online]. Available: https://grpc.io/

[9] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP),” Jun. 2014. Accessed: Sep. 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7252.html

[10] IETF QUIC Working Group, “QPACK Static Table.” Accessed: Nov. 2025. [Online]. Available: https://github.com/quicwg/base-drafts/wiki/QPACK-Static-Table

[11] Meta Engineering, “Introducing OpenZL: An Open Source Format-Aware Compression Framework.” Accessed: Oct. 2025. [Online]. Available: https://engineering.fb.com/2025/10/06/developer-tools/openzl-open-source-format-aware-compression-framework/

[12] Meta Platforms, Inc., “OpenZL: Format-Aware Compression Framework.” Accessed: Oct. 2025. [Online]. Available: https://openzl.org/

[13] Meta Platforms, Inc., “OpenZL: A novel data compression framework.” Accessed: Oct. 2025. [Online]. Available: https://github.com/facebook/openzl

[14] R. Tatoris and B. Wolters, “The end of the road for Cloudflare CAPTCHAs.” Accessed: Sep. 2025. [Online]. Available: https://blog.cloudflare.com/end-cloudflare-captcha/

[15] Cloudflare, “Cloudflare Challenges Documentation.” Accessed: Sep. 2025. [Online]. Available: https://developers.cloudflare.com/cloudflare-challenges/

List of Tools

Area	Tools
Research	Google, Perplexity, Claude
Data analysis	Excel
Diagrams	Mermaid.js, Draw.io
Text writing, Text optimization, Grammar, Spellcheck, Translations	DeepL, Typos, cspell, Claude, ChatGPT
Project Management	Github Projects & Issues, Teams, Clockify
Coding	Zed, Visual Studio Code, Claude, ChatGPT, golangci-lint
DevOps	Github Actions, Act