University of Zurich[UZH]

MASTER THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Design and Implementation of Novel Transport and Application Layer Measurement Techniques

Mete Polat
Zürich, Switzerland
Student ID: 18-932-129

Supervisor: Thomas Grübl, Daria Schumm
Date of Submission: November 26, 2025

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Master Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: https://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and Generative AI was used to improve the coherence of sentences and paragraphs. I am aware that I take full responsibility for the scientific character of the submitted text myself, even where AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 26. November 2025

Signature of student

Zusammenfassung

Die Verbreitung neuer Transport- und Anwendungsschichtprotokolle wie QUIC und HTTP/3 verläuft schneller als die Weiterentwicklung bestehender Internet-Messwerkzeuge, die häufig auf TCP sowie HTTP/1.1 beziehungsweise HTTP/2 zugeschnitten sind und keine Unterstützung für strukturierte Protokollaufzeichnung bieten. Diese Arbeit präsentiert die Konzeption, Implementierung und Evaluation eines modularen, erweiterbaren Frameworks für internetweite aktive Messungen moderner Transport- und Anwendungsschichtprotokolle. Das Framework, QUIC Lab, besteht aus drei Subsystemen: Einem Domain Extractor, der aus grossen Domainlisten reproduzierbare Zielmengen erzeugt, einer Probing-Engine, die konfigurierbare QUIC- und HTTP/3-basierte Messungen via plug-infähige Probes orchestriert, sowie einem Analyzer, der Recorder-Ausgaben und qlog-Traces (strukturierte QUIC-Protokollierung) einliest und zu aggregierten Statistiken aufbereitet.

QUIC Lab ist in Rust auf Basis der TQUIC-Bibliothek von Tencent implementiert und trennt klar zwischen probe-spezifischer Logik und gemeinsamen Diensten wie Konfigurationsmanagement, DNS-Auflösung, Lastbegrenzung, Transportabstraktion, Aufzeichnung und qlog-Multiplexing. Seine Skalierbarkeit und Robustheit werden anhand von zwei grossangelegten Scans von etwa 6.24 M Domains aus einer Tranco-Liste demonstriert, die von privat betriebenen Servern in der Schweiz sowie einer AWS EC2-Instanz in der Region US-East durchgeführt wurden. Bei über rund 5.48 M Verbindungsversuchen pro Standort erzeugte das Framework etwa 240 M qlog-Ereignisse bei einer konstanten Messrate von ungefähr 26 Domains pro Sekunde, bei gleichzeitig geringer Auslastung von CPU, Arbeitsspeicher und Netzwerk.

Die empirischen Ergebnisse zeigen, dass QUIC und HTTP/3 unter populären Domains weit verbreitet, aber nicht universell verfügbar sind: Erfolgreiche QUIC-Handshakes wurden bei 28–32% der Ziele beobachtet, wobei HTTP/3 die ausgehandelten ALPN-Werte klar dominiert. Die serverseitigen QUIC-Transportparameter weisen hoch konsistente Verteilungen über Standorte hinweg auf, was auf eine Konvergenz hin zu einigen wenigen betrieblichen Standardkonfigurationen hindeutet. In der untersuchten Population finden sich keine Hinweise auf eine produktive Nutzung von Multipath QUIC.

Abstract

The deployment of new transport and application-layer protocols such as QUIC and HTTP/3 has outpaced existing Internet measurement tooling, which is often tailored to TCP and HTTP/1.1 or HTTP/2 and lacks support for structured protocol logging. This thesis presents the design, implementation, and evaluation of a modular, extensible framework for Internet-wide active measurements of modern transport and application-layer protocols. The framework, termed QUIC Lab, comprises three subsystems: a Domain Extractor that constructs reproducible target sets from large domain lists, a probing engine that orchestrates configurable QUIC- and HTTP/3-based measurements via pluggable probes, and an Analyzer that ingests recorder outputs and qlog (structured QUIC logging) traces to derive aggregated statistics.

QUIC Lab is implemented in Rust on top of Tencent's TQUIC library, with a clear separation between probe-specific logic and shared services such as configuration management, DNS resolution, rate limiting, transport abstraction, recording, and qlog multiplexing. Its scalability and robustness are demonstrated through two large-scale scans of approximately 6.24 M domains derived from Tranco, executed from privately operated servers in Switzerland and an AWS EC2 instance in the US-East region. Across roughly 5.48 M connection attempts per vantage point, the framework produced about 240 M qlog events with sustained probing rates of approximately 26 domains per second, while maintaining low CPU, memory, and network utilization.

The empirical results show that QUIC and HTTP/3 are widely, but not universally, deployed among popular domains, with successful QUIC handshakes observed for 28–32% of targets and HTTP/3 dominating the negotiated ALPN values. Server-side QUIC transport parameters exhibit highly consistent distributions across vantage points, indicating convergence on a small set of operational defaults. No evidence of Multipath QUIC deployment was found in the examined population.

Acknowledgments

I am sincerely grateful to my supervisor, Thomas Grübl, whose consistent guidance, expertise, and thoughtful feedback were essential to the successful completion of this thesis. I also wish to express my appreciation to Prof. Burkhard Stiller and the Communication Systems Group at the University of Zurich for providing the opportunity and resources needed to carry out this work. Finally, I am thankful to my friends and family for their patience, encouragement, and constant support throughout this time.

Contents

Chapter 1

Introduction

1.1 Motivation

The widespread expansion of internet usage and increasingly sophisticated web applications has driven the demand for transport protocols that deliver higher performance, improved reliability, and enhanced security. Established protocols such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), although fundamental to the internet, are gradually revealing shortcomings when faced with the demands of contemporary web traffic [1], [2]. Challenges including elevated latency—particularly across long-distance links or mobile networks—frequent network transitions on mobile devices, and stricter security and privacy expectations highlight the constraints of these traditional protocols [3]. While TCP ensures reliable and ordered data transmission, it introduces connection establishment overhead through its three-way handshake and is susceptible to head-of-line blocking when multiple data streams are multiplexed over a single connection [4]. In contrast, UDP enables lightweight, connectionless communication but does not natively provide reliability or congestion control mechanisms [5]. As a result, fulfilling modern performance requirements for web browsing, video streaming, real-time communication, and other delay-sensitive services becomes increasingly difficult.

These challenges have prompted both researchers and practitioners to pursue transport solutions that are more efficient, secure, and adaptable [3], [6], [7]. This motivation has given rise to new transport protocols such as Quick UDP Internet Connections (QUIC) [1], along with advances at the application layer, including Hypertext Transfer Protocol Version 3 (HTTP/3) [2].

For researchers and security analysts, gaining insight into the real-world deployment and behavior of these protocol features is essential. Conventional scanning tools typically concentrate on identifying services or enumerating versions, but they often lack the capability to evaluate detailed protocol functionalities. Consequently, specialized tools are required to actively probe and examine the specific behaviors of modern transport and application-layer protocols.

1.2 Description of Work

First, a comprehensive review of existing research, protocol specifications, and ethical guidelines is performed to identify the methodological gaps in current Internet measurement approaches, with a particular focus on advanced QUIC features.

Based on the insights gained from this review, a modular scanning and fuzzing toolkit is designed, and then implemented. For QUIC, the toolkit must support full connection establishment, parsing of transport parameters, and protocol-compliant error handling. The design emphasizes extensibility, reproducibility, and adherence to ethical scanning constraints.

Finally, an evaluation is carried out using controlled deployments on multiple Virtual Private Servers in different geographic regions. The toolkit is used to assess feature support on enterprise-grade infrastructure, measure implementation variability, and analyze performance indicators such as scan speed, detection accuracy, and cross-location consistency. Ethical scanning practices guide target selection and experiment execution throughout all stages of the work.

1.3 Thesis Outline

Chapter 2 introduces key concepts such as QUIC, Multipath QUIC, and qvis, to establish a theoretical foundation. Chapter 3 discusses the current state of research in the field and shows the research gaps. In chapter 4, a technology-independent description of the solution is provided, followed by a concrete implementation in chapter 5. The evaluation is presented in chapter 6 and concluded in chapter 7, along with a summary of this thesis.

Chapter 2

Background

2.1 Internet Measurements

Internet measurement studies analyze public Internet traffic to track threats, user behavior, and the performance and security of systems. Telescopes and reactive vantage points observe unsolicited traffic and sometimes elicit limited application-layer responses. They also collect data from benign users. This creates ethical challenges: consent is infeasible at scale, anonymization can obscure key phenomena, and institutional review boards may miss network-level harms or misjudge risk-benefit trade-offs [8].

2.1.1 Active vs. Passive Measurements

While this research focuses on active probing, it is useful to note how passive measurements complement the process. Passive measurement involves observing real traffic (e.g., via network taps or server logs) to infer deployment and usage of protocols. For instance, [9] found that QUIC (in its pre-IETF Google variant) already accounted for 2.6%–9.1% of Internet traffic by 2017, with Google using QUIC for about 42% of its traffic. Similar adoption figures were reported by Google itself [10], highlighting QUIC's rapid deployment at Internet scale. Such insights reveal adoption trends in the wild. However, passive data alone cannot fully assess specific feature support, since many QUIC capabilities like 0-RTT or migration might be infrequently exercised or not visible without active triggers [11], [12]. Passive methods are also constrained by where the observer sits; they see what naturally occurs, which may omit edge cases. Active measurement, on the other hand, allows systematically probing features on demand (e.g., deliberately attempting a connection migration) across many targets [11]. Active and passive approaches are thus complementary: passive studies give a broad picture of real-world usage and performance, whereas active scans let researchers test capabilities and compliance (even for rarely used features) in a controlled way. In academic practice, it's common to use passive findings to motivate active tests (for example, noticing increasing QUIC traffic share passively, then actively scanning to see which servers support QUIC) [11].

2.1.2 Ethics in Internet Measurements

Active scanning must be conducted responsibly to avoid harming others. [13]. Key ethical practices include rate limiting the probe traffic to prevent overloads and progressive ramp-up of scan intensity. For example, one large-scale QUIC scan first tested locally and then gradually increased the number of targets and packet rate while monitoring for issues. It's also crucial to maintain a blocklist of networks or hosts that have requested not to be scanned [14]. Providing a clear opt-out mechanism (e.g., an email contact or a web page on the scanning host with instructions) demonstrates respect for network owners' preferences. Likewise, using informative reverse DNS names or User-Agent strings (for HTTP requests) that identify the measurement and provide contact information can help administrators understand and contact the researcher if needed. Data collected from purely network scanners is generally acceptable without consent, whereas data involving end-users should be anonymized or obtained with consent whenever possible. In all cases, potential harm should be minimized: avoiding sending payloads that could trigger crashes or large downloads, and sticking to innocuous requests (e.g., a small HEAD or GET request for HTTP). Active measurements that might inadvertently impact users (e.g., by inducing server load or alarms) should be carefully controlled or avoided [8].

2.1.3 Selecting Targets

Choosing an appropriate set of targets is essential when faced with large pools. A common strategy is to focus on popular domains using established "top lists". In the research community, the Alexa Top 1 Million list was previously prevalent, although it had reliability issues beyond the very top sites [13]. Modern studies favor the Tranco list, a research-oriented ranking that aggregates multiple sources over 30 days for stability and reproducibility. Using a stable list like Tranco (with a fixed snapshot ID) allows others to easily replicate the experiment [15]. If the research goal is broad Internet coverage (e.g., finding all QUIC-supporting hosts), combining multiple sources can help: for instance, one might take a top domains list, augment it with known QUIC host hitlists or data from DNS. Recent QUIC scans demonstrate using DNS records and alternative services to guide target selection. For example, [14] identified QUIC hosts via three methods: direct UDP scanning, HTTP Alt-Svc headers, and DNS SVCB/HTTPS records. Each method found some hosts the others missed. Notably, DNS HTTPS records revealed many Cloudflare-supported domains (but were biased toward Cloudflare). In practice, a hybrid approach can be used: starting with a large domain list (e.g., top 100k sites), resolve to IPs, and then optionally filter or augment using known QUIC indicators (Alt-Svc or HTTPS records) [14].

2.2 Transport Layer Security 1.3

Transport Layer Security (TLS) 1.3 is the most recent version of the widely deployed security protocol that secures HTTPS and, by extension, protocols such as HTTP/2,

HTTP/3, and QUIC encryption. Finalized in 2018, TLS 1.3 represents a substantial redesign focused on enhancing both security and performance. It removes obsolete and insecure cryptographic mechanisms (e.g., static RSA key exchange and legacy ciphers) and streamlines the protocol's state machine to mitigate historical vulnerabilities. A key improvement is reduced handshake latency: unlike TLS 1.2, which requires two round-trips, TLS 1.3 completes a full handshake in a single round-trip (1-RTT) and additionally supports zero-round-trip (0-RTT) mode for resumed sessions, allowing clients to send data immediately. This leads to faster establishment of secure connections. Furthermore, TLS 1.3 encrypts a larger portion of the handshake, improving privacy by concealing more metadata from passive observers [16]. Overall, TLS 1.3 enhances both privacy and connection setup time, making it a critical component of modern transport protocols like QUIC and HTTP/3, which integrate TLS 1.3 directly into their handshakes.

2.3 Quick UDP Internet Connections (QUIC)

QUIC is a modern transport protocol originally developed by Google and later standardized by the IETF as QUIC version 1. Operating over UDP, QUIC implements advanced transport-layer functionality in user space while providing a reliable, ordered byte-stream abstraction to upper-layer protocols. It was designed to address long-standing limitations of TCP, such as slow connection establishment, head-of-line blocking in multiplexed applications, and lack of native support for mobility and connection migration [1].

QUIC is message-oriented, multiplexed, and inherently secure: it enables multiple concurrently active streams within a single connection, each with independent flow control and ordering guarantees [1]. Encryption is integrated into the transport layer by mandating the use of TLS 1.3 [16] for all application data and most control information. The QUIC–TLS mapping, including key derivation, record protection, and the use of CRYPTO frames for the handshake, is specified in a separate document [17]. Loss detection and congestion control are defined independently of the transport mapping [18], which allows future versions of QUIC to evolve while reusing these algorithms.

The QUIC handshake is optimized for low latency by combining the transport and cryptographic handshakes into a single exchange and by supporting zero-round-trip (0-RTT) data for resumed connections. QUIC connections are further designed to be robust to changes in network paths. They are identified by opaque connection identifiers rather than the IP address and port tuple, which enables seamless migration across interfaces and networks [1], [17].

Sections 2.3.1 to 2.3.4 introduce key features of QUIC that are particularly relevant to transport and application-layer measurement.

2.3.1 Connection Migration

In QUIC, a connection is identified by an opaque connection ID chosen by the endpoints rather than by the 4-tuple of source and destination IP addresses and ports. This decou-

pling of the connection identity from the underlying network path enables a connection to survive changes in the path, such as Network Address Translator (NAT) rebinding, interface changes, or handovers between access networks. A *path* is defined as a specific 4-tuple over which packets are exchanged [1].

Connection migration mechanisms distinguish between implicit migration due to NAT rebinding and explicit, active migration initiated by an endpoint. In the NAT rebinding case, packets from the peer appear to originate from a different 4-tuple while the connection ID remains unchanged. The peer can silently migrate the connection state to the new path once basic validation succeeds [1]. In contrast, during active migration, typically initiated by a client that switches from one local interface to another, an endpoint deliberately starts sending non-probing packets from a new local address and port.

To prevent off-path attackers from redirecting traffic or amplifying traffic towards a victim, QUIC performs *path validation* before using a new path for non-probing traffic. Path validation uses PATH_CHALLENGE/PATH_RESPONSE frame pairs: the probing endpoint sends a PATH_CHALLENGE on the candidate path and requires a matching PATH_RESPONSE from its peer before considering the path valid [1]. Until validation completes, an endpoint is subject to anti-amplification limits and typically restricts itself to sending only probing frames on the new path [1], [18].

Servers can influence migration behavior via transport parameters. For instance, the `disable_active_migration` transport parameter signals that the client must not actively migrate away from the path used during the handshake [1]. Conversely, the `preferred_address` transport parameter allows a server to advertise an alternative address to which the client is encouraged to migrate after the handshake completes, for example to move from a load balancer front-end to a backend server [1].

From a transport-measurement perspective, QUIC connection migration introduces additional degrees of freedom: flows can change path while maintaining a stable connection identifier, and validation traffic (PATH_CHALLENGE/PATH_RESPONSE) can be observed as an explicit signal of path changes.

2.3.2 0-RTT Resumption

QUIC supports the transmission of application data in 0-RTT, i.e., before completion of the handshake, for resumed connections. This mechanism builds on TLS 1.3 early data and the pre-shared key (PSK) mode of TLS 1.3 [16]. During an initial connection, the server may issue a resumption ticket bound to a PSK and associated configuration state. In subsequent connections, the client can present this ticket and immediately send application data protected under keys derived from the PSK, without waiting for the server's first flight [17].

Use of 0-RTT in QUIC requires that the client and server agree on a set of remembered parameters from the original connection. These include the application protocol (via ALPN), the QUIC version, the cipher suite, and a subset of the server's transport parameters, including flow control limits and other configuration relevant to transport

behavior [1], [17]. The server must verify that its current configuration is compatible with the stored parameters before accepting 0-RTT data; otherwise it has to reject early data and force the client to fall back to a 1-RTT handshake [17].

A central limitation of 0-RTT is that early data is not protected against replay at the TLS layer [16], [17]. QUIC therefore places responsibility on applications and deployments to ensure that operations triggered by 0-RTT data are either idempotent, tolerant to replay, or otherwise protected by additional mechanisms (such as application-level anti-replay state). Servers are permitted to reject early data for policy reasons, for example when they cannot provide strong replay protection or when configuration differences prevent safe reuse of stored transport parameters [17].

2.3.3 Version Negotiation

QUIC is explicitly versioned at the transport layer. Each QUIC long-header packet carries a version field that specifies the protocol version used for the connection [1]. To enable evolution of the protocol, endpoints must cope with unsupported versions and, where possible, select a mutually supported version.

The base QUIC specification defines a *Version Negotiation* packet with version value 0, which servers use to indicate that the version chosen by the client is not supported [1]. Upon receiving a client Initial packet with an unknown version, a server may respond with a Version Negotiation packet that echoes the client's connection IDs and contains a list of versions that the server is willing to accept [1]. The client can then abandon the current attempt and initiate a new connection using one of the advertised versions. QUIC version 1 describes how endpoints handle Version Negotiation packets but intentionally leaves the detailed version-selection logic and downgrade protection to future specifications [1].

The version-invariant properties of QUIC, including the format of Version Negotiation packets, are defined separately [19]. Building on these invariants, a complete, downgrade-resistant version negotiation mechanism has been specified in a dedicated document [20]. That mechanism introduces the concept of the client's *Original Version* and *Chosen Version*, as well as the final *Negotiated Version*, and defines two complementary procedures:

- *Incompatible version negotiation*, in which the server cannot parse the client's first flight and responds with a Version Negotiation packet listing its offered versions, causing the client to start a new connection with a different version [20].
- *Compatible version negotiation*, in which the server can parse the client's first flight and the client's first-flight format is compatible with another version. In this case, the server can switch to a different negotiated version without incurring an extra round trip by conceptually converting the first flight into that version [20].

To support these procedures, the version negotiation mechanism introduces a `version_information` transport parameter that carries the client's and server's view of available and chosen versions and is authenticated as part of the QUIC–TLS handshake [17], [20]. The client validates that the server's chosen version is consistent with

the versions it offered; if validation fails, the client aborts the connection with a dedicated version negotiation error [20]. This design prevents on-path attackers from forcing a downgrade by forging Version Negotiation packets or manipulating the Version field in long headers.

2.3.4 Flow Control

QUIC employs a limit-based, receiver-driven flow control scheme to prevent a sender from overwhelming the receiver's buffers. Flow control in QUIC operates at two levels: per-connection and per-stream. Both are expressed as limits on the cumulative number of bytes that a peer is allowed to send [1].

At connection establishment, each endpoint advertises initial flow control limits via transport parameters: `initial_max_data` defines the maximum number of bytes the peer may send in total across all streams, while `initial_max_stream_data_bidi_local`, `initial_max_stream_data_bidi_remote`, and `initial_max_stream_data_uni` define per-stream limits for the different stream types [1]. These initial limits bound the amount of data that can be sent before the receiver has processed any application data. During the connection, the receiver can increase these limits by sending MAX_DATA and MAX_STREAM_DATA frames, respectively, thereby granting additional credit to the sender [1].

The sender is required to track, for each stream and for the connection as a whole, the highest byte offset sent and to ensure that no data is transmitted beyond the advertised limits. If either the per-stream or the connection-level credit is exhausted, further sending on the affected stream (or on any stream, for the connection limit) becomes flow-control blocked until new credit is advertised. The sender can signal this condition to the receiver using BLOCKED, STREAM_DATA_BLOCKED, or analogous frames, which can be useful hints for tuning flow control limits [1].

Flow control is distinct from, but interacts with, congestion control as specified in the QUIC loss detection and congestion control specification [18]. Congestion control protects the network from overload by limiting the volume of in-flight data based on observed loss and delay, whereas flow control protects endpoints from resource exhaustion by limiting the total amount of data a peer may send. In practice, the effective sending rate is constrained by the minimum of the congestion window and the available flow control credit.

2.4 Managing Multiple Paths for a QUIC Connection (Multipath QUIC)

While QUIC version 1 supports connection migration by switching a connection between different network paths, only a single path is active at any point in time [1]. Connection migration primarily targets robustness against path changes, such as NAT rebinding

or moving between Wi-Fi and cellular networks, but it does not exploit the aggregate capacity or redundancy of multiple paths simultaneously. To address this limitation, a multipath extension for QUIC is currently being standardized in the IETF QUIC Working Group [21]. This extension enables a single QUIC connection to use multiple network paths in parallel, using the same or different 4-tuples of IP addresses and UDP ports, while preserving QUIC's security properties and application semantics.

2.4.1 Design Overview and Goals

The multipath extension introduces the notion of a *path identifier* (path ID) to manage multiple simultaneous paths within a QUIC connection [21]. Each path corresponds to a network path in the sense of RFC 9000 and is associated with its own packet number space and connection identifiers (CIDs). Path IDs are monotonically increasing 32-bit integers; path ID 0 denotes the initial path and additional paths are assigned consecutive identifiers that are never reused for the lifetime of the connection.

A key design objective is to reuse the existing QUIC handshake and cryptographic mechanisms defined in RFC 9001 without requiring changes to the TLS 1.3 key exchange [17]. The multipath extension is negotiated via a new transport parameter and only becomes active once the QUIC handshake has completed successfully; all multipath-specific signaling is carried in 1-RTT packets and remains protected by QUIC-TLS [21]. As a result, the extension retains the confidentiality and integrity guarantees of QUIC while adding the ability to create, use, and remove multiple paths in a controlled fashion.

Figure 2.1 illustrates the high-level message flow for enabling multipath on a QUIC connection. The client and server first perform the standard QUIC + TLS 1.3 handshake while advertising support for the multipath extension via the `initial_max_path_id` transport parameter. After the handshake has completed, application data is initially sent on path ID 0. The client then activates an additional path (path ID 1) using per-path connection identifiers, after which both paths can concurrently carry application data.

2.4.2 Path Identifiers and Transport Parameter Negotiation

Support for multipath QUIC is indicated during the handshake using the `initial_max_path_id` transport parameter. This parameter is a variable-length integer that specifies the maximum path ID an endpoint is willing to maintain at connection establishment; its value is limited to $2^{32} - 1$ to ensure uniqueness of the AEAD nonces that incorporate the path ID. If either endpoint omits this parameter, the multipath extension is disabled and the connection behaves as a regular single-path QUIC connection. When both endpoints advertise `initial_max_path_id`, the extension is enabled after the handshake completes and additional paths can be created up to the negotiated maximum [21].

The connection ID of each packet binds that packet to a specific path ID and thus to a particular packet number space. Each CID is associated with exactly one path ID, but

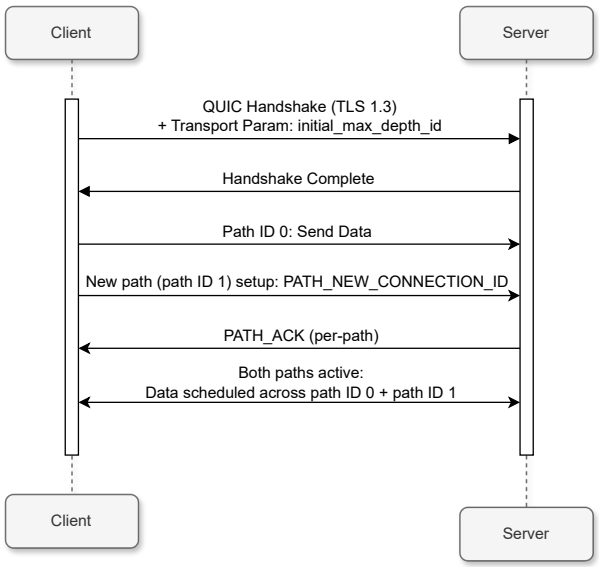

Figure 2.1: High-Level Message Flow for Enabling Multipath on a QUIC Connection

multiple CIDs may be issued for a single path, for example to support CID rotation or migration within that path. The same path ID is used in both directions of the connection, and each path maintains its own sender and receiver packet number state starting from 0 [21]. This design enables direct reuse of QUIC's loss detection and congestion control mechanisms on a per-path basis, while requiring non-zero CIDs and a modified AEAD nonce construction that includes the path ID [18], [21].

2.4.3 Path Lifecycle and Path Management

After the handshake has indicated multipath support, endpoints can manage multiple paths using a set of dedicated frames and procedures [21]. Path management covers four aspects: path initiation and validation, per-path connection ID handling, preferred path signaling, and explicit path closure.

Path initiation and validation

To open a new path, an endpoint selects an unused path ID and uses a connection ID associated with that path ID when sending packets on the new 4-tuple. The peer performs path validation using QUIC's existing PATH_CHALLENGE/PATH_RESPONSE mechanism before the path can be used for application data, as in single-path connection migration [1], [21]. Endpoints are encouraged to allocate additional per-path resources only after validation succeeds to limit exposure to resource-exhaustion attacks.

Per-path connection identifiers

Each path relies on a set of CIDs that are specific to its path ID. The extension defines frames such as PATH_NEW_CONNECTION_ID and PATH_RETIRE_CONNECTION_ID, which mirror the semantics of NEW_CONNECTION_ID and RETIRE_CONNECTION_ID in RFC 9000 but operate on a per-path basis. Together with the MAX_PATH_ID, PATHS_BLOCKED, and PATH_CIDS_BLOCKED frames, these mechanisms allow endpoints to bound the number of active paths and associated CIDs for resource control [21].

Preferred path signaling and path closure

Applications may prefer some paths over others (for example, cellular versus Wi-Fi). The extension therefore specifies PATH_STATUS_AVAILABLE and PATH_STATUS_BACKUP frames, which allow endpoints to signal which paths are primary and which are considered backups. Paths can be explicitly removed using PATH_ABANDON frames, accompanied by error codes that indicate why a path was closed (e.g., resource limits or poor performance). The specification does not define per-path idle timeouts; instead, endpoints are free to decide when to abandon unused paths, potentially using keep-alive traffic if long-lived backup paths are desired [1], [21].

2.4.4 Scheduling, Congestion Control, and Transport Dynamics

When multipath is enabled, senders maintain separate congestion control state for each path [1], [18], [21]. Each path has its own congestion window, round-trip time estimator, and probe timeout (PTO), and a sender must not transmit more data on a path than allowed by that path's congestion window. This independence allows different paths to react appropriately to heterogeneous characteristics such as bandwidth, delay, and loss.

The extension deliberately does not standardize any packet scheduling algorithm across paths. Instead, it provides only basic implementation guidance and leaves the choice of scheduler to the application or implementation, ranging from simple failover strategies to algorithms that exploit the aggregate capacity of all active paths [21]. For example, a video streaming application might distribute traffic across paths with similar RTTs, whereas an interactive application might restrict traffic to a low-latency path and keep another path as a hot standby. The draft also discusses practical aspects such as RTT estimation per path, handling paths with different path maximum transmission units (PMTUs), and managing idle paths and keep-alives [21].

2.5 qlog: Structured Logging for Network Protocols

Modern transport protocols such as QUIC encrypt most of their wire image, which makes passive measurement and debugging substantially more difficult than for TCP [1], [16],

[22]. Endpoint-based logging is therefore required to obtain protocol semantics such as packet and frame types, state-machine transitions, and congestion-control events. The qlog main schema Internet-Draft defines an extensible structured logging format for network protocols that standardizes how such information is captured and shared across implementations and tools [23]. By providing common schemas for log files, traces, events, and protocol-specific extensions, qlog enables interoperable analysis pipelines that are particularly suitable for large-scale QUIC and HTTP/3 measurements.

2.5.1 Design Goals and Overall Structure

qlog is designed around several core principles: streamable, event-based logging; low overhead for log producers; extensibility; and aggregation-friendly structure for consumers and tools. A qlog artefact is organized in a three-level hierarchy of *log file*, *trace*, and *event*. The abstract `LogFile` class defines fields that are common to all log files, in particular a `file_schema` URI that identifies the concrete schema and a `serialization_format` media type [23]. Concrete log-file schemas derive from this definition and may add additional metadata.

A *trace* corresponds conceptually to a single logical data flow observed at one vantage point, for example one QUIC connection as seen by the client, server, or a network observer. Each trace contains optional descriptive metadata, information about its vantage point, a set of event schema URIs that describe the event namespace(s) used, and a sequence of events. This structure allows several related traces—for example client-side, server-side, and on-path logs of the same connection—to be aggregated into a single file while preserving their provenance [23].

2.5.2 Traces, Vantage Points, and Grouping

The `Trace` and `TraceSeq` structures describe per-flow metadata and provide context for events. They include an optional `vantage_point` field that identifies whether the trace originates from a client, server, or network observer, and can additionally encode the flow direction when traces are generated from packet captures [23]. This information allows analysis tools to interpret events such as *packet sent* or *packet received* correctly even when identifying information from the wire image (for example IP addresses) has been removed for privacy reasons.

Qlog introduces the `group_id` field to support grouping of events belonging to the same logical entity within a trace or across traces. Typical examples include using a QUIC connection identifier to group all events for one connection, or tagging events with a quality-of-service class. The `group_id` can be stored either directly in each event or once in the `common_fields` section of a trace if it is constant for all events [23]. This mechanism is particularly useful in server deployments that log many connections in a single streamed log file and later split them per connection during post-processing.

The `tuple` field provides a similar abstraction for network paths. It associates events with an identifier representing a specific IP-address and port four-tuple, but leaves the

concrete encoding of this identifier to protocol-specific schemas. This abstraction allows logs to represent connection migration or multipath operation, where several tuples can be active concurrently for the same logical connection, without exposing raw addressing information directly [23].

2.5.3 Events and Event Schemas

At the lowest level, qlog represents protocol behavior as a sequence of events. The abstract `Event` type comprises a timestamp, a `name`, and a protocol-specific `data` object, with optional fields for time formatting, grouping, system information, and custom extensions. Event names are globally unique strings of the form `namespace:event_type`, for example `quic:packet_sent`, and are defined by *event schemas* [23].

An event schema defines a namespace and the set of event types and data structures it contains. The main schema document specifies generic namespaces such as `loglevel` for free-form logging (errors, warnings, and informational messages) and `simulation` for recording test scenarios and markers. Protocol-specific schemas, for example for QUIC and HTTP/3, define detailed events such as packet transmission, frame parsing, and connection-state updates. Event schemas are identified by URIs and listed in the `event_schemas` field of each trace, which enables tools to determine the expected event vocabulary for a given log [23].

Qlog is explicitly extensible. The `data` field uses a CDDL type extension point that permits additional events and new fields to be added by later schemas without breaking existing tooling. Similarly, each event data definition can contain an extension socket that allows future documents to add optional properties [23]. This extension mechanism is essential for evolving protocols such as QUIC, where new transport parameters or frames may be introduced without changing the base logging schema.

To reduce redundancy, qlog offers the `common_fields` mechanism: fields that have the same value for all events in a trace, such as time format, reference time, group identifier, and tuple, can be moved from individual events into a shared dictionary. This significantly reduces log size and simplifies logging implementations while keeping the event model conceptually uniform [23].

2.5.4 Serialization Formats and Streaming

Although schema definitions are serialization-independent, the main schema document specifies mappings to JSON and JSON Text Sequences, which are recommended default formats due to their interoperability and ease of tooling [24], [25]. For non-streaming use cases, the `QlogFile` schema aggregates one or more traces into a single JSON object, with media type `application/qlog+json` and typically the `.qlog` file extension. This format is well suited for offline analysis but requires the file to be closed properly before it can be parsed.

For streaming scenarios, qlog defines the `QlogFileSeq` schema, which is serialized using JSON Text Sequences. A `QlogFileSeq` file contains a single `TraceSeq` header record followed by an unbounded sequence of event records, each prefixed by a record-separator character and terminated by a newline [23], [25]. The corresponding media type is `application/qlog+json-seq`, and the conventional file extension is `.sqlog`. This stream-oriented design enables low-overhead continuous logging in high-volume deployments and is especially suitable for long-running QUIC servers and large-scale Internet scans.

Qlog also provides guidelines for optimization and interoperability, including support for I-JSON environments with restricted integer ranges and for truncating raw byte fields while retaining length metadata [23], [26]. These features are important when logs are processed by browser-based tools or must be stored efficiently at scale.

2.5.5 Security and Privacy Considerations

Because qlog can expose detailed packet and frame-level information, connection identifiers, addressing information, and even decrypted application data, its use has significant security and privacy implications. The main schema therefore includes a dedicated discussion of data-at-risk categories, such as IP addresses, QUIC connection IDs, TLS session secrets, and raw payloads, and recommends that operators carefully control which fields are logged and who can access the resulting files [23], [27]. Techniques such as data minimization, anonymization, truncation of raw payloads, and encryption of log files at rest and in transit are explicitly encouraged.

Qlog further suggests operational best practices including access-control and auditing for both log generation and log consumption, as well as environment variables (`QLOGDIR`, `QLOGFILE`) that allow operators to direct log output in a controlled fashion. These mechanisms are essential when qlog is deployed in production systems that process sensitive user traffic.

2.5.6 Role in Transport and Application-Layer Measurement

For the purposes of transport- and application-layer measurement, qlog provides a protocol-independent but QUIC-aware logging substrate that combines high semantic richness with a well-specified, machine-readable format. In contrast to pcap-based measurements, which operate solely on the wire image, qlog exposes internal protocol state such as loss-detection timers, congestion-control variables, stream-level flow control updates, and connection identifiers [23]. When combined across multiple vantage points, qlog traces enable reconstruction of end-to-end behavior and correlation with external data sources such as DNS measurements or HTTP performance metrics.

The event-based, streaming design of qlog, together with its grouping and tuple abstractions, makes it particularly well suited for large-scale experiments involving many concurrent QUIC connections, connection migration, or multipath operation. As a consequence, qlog is a central building block for the measurement techniques developed later in this thesis.

Chapter 3

Related Work

This chapter provides an overview of the state of the art in transport and application layer measurement techniques, thereby establishing the necessary context for the novel contributions of this thesis. The initial step in this study is a review of the measurement challenges of legacy and transitional protocols, such as Multipath TCP (MPTCP). These challenges directly motivate the design of modern encrypted protocols. A thorough examination of measurement methodologies developed for the Quick UDP Internet Connections (QUIC) protocol is subsequently conducted, with a particular emphasis on the time period from 2019 to the present. The recent work is categorized as follows: large-scale active discovery, server implementation fingerprinting, passive and in-band analysis, performance benchmarking, and measurement of specific protocol mechanisms. In conclusion, a comprehensive review is presented on the evolution from MPTCP to Multipath QUIC, as this subject is a primary focus of the present thesis.

3.1 The Evolving Landscape of Transport Protocol Measurement

For decades, transport-layer measurement techniques were inextricably linked to the design of the Transmission Control Protocol (TCP). Network monitors, middleboxes, and researchers relied on their ability to passively observe unencrypted TCP headers. By tracking TCP sequence (SEQ) and acknowledgment (ACK) numbers, one could easily infer network round-trip time (RTT), packet loss, and reordering. This provided the foundation for network diagnostics and management. Active measurements were similarly straightforward and often involved simple probes, such as TCP SYN packets, to determine service availability [28].

This measurement paradigm was first and most significantly challenged by transport protocols that extended TCP, notably Multipath TCP (MPTCP). First standardized as an experimental protocol (MPTCPv0) in 2013 and later as MPTCPv1 in 2020, MPTCP extends TCP to utilize multiple network paths (e.g., Wi-Fi and cellular) simultaneously by introducing new TCP options, such as `MP_CAPABLE`, during the initial handshake [29].

This reliance on TCP options proved to be a critical flaw for both deployment and measurement. Longitudinal studies conducting Internet-wide scans for the `MP_CAPABLE` option revealed that naive active measurements produced overwhelmingly false-positive results. This measurement failure was a direct consequence of middlebox interference. Middleboxes (e.g., firewalls, NATs) on the path, not understanding the new options, would either strip them from the packet, making the server appear as a legacy TCP endpoint, or, more deceptively, mirror the unknown options back to the sender [30]. [29] conducted a comprehensive longitudinal study and found that this mirroring artifact affected a "substantial share" of seemingly MPTCP-capable hosts; on port 80, 80–90% of all positive `MP_CAPABLE` responses were artifacts of mirroring middleboxes.

This fundamental challenge forced the network measurement community to develop a more sophisticated, multi-stage active measurement methodology to filter these false positives. The state-of-the-art technique for MPTCP discovery involves: (1) a high-speed, stateless scan using ZMap to probe for the `MP_CAPABLE` option; (2) a filtering stage to discard all responses that merely "echo" the sender's MPTCP key; and (3) a robust validation stage using tools like Tracebox [29]. Tracebox sends MPTCP packets and observes hop-by-hop which device on the path, if any, modifies or strips the options, thereby isolating true server-side support from on-path interference [30].

Using this robust methodology, longitudinal studies from 2020-2022 revealed the true, sparse state of MPTCP deployment. True MPTCPv0 deployment, while growing, was limited to approximately 13,000 IPv4 addresses, and MPTCPv1 deployment was "comparatively low" or "almost non-existent," with approximately 100 hosts, driven almost entirely by Apple. Passive traffic analysis from CAIDA and MAWI confirmed this, showing that the MPTCP traffic share, while increasing 20 times, remained below 0.4% of TCP traffic and was "almost all" attributable to Apple [29].

The failure of MPTCP, due to its reliance on clear-text TCP options being ossified by middleboxes, serves as the primary technical motivation for the design of QUIC. QUIC was designed from the ground up to avoid this problem by running over UDP and encrypting all of its transport-layer headers [31]. This design creates a new measurement paradox: by successfully hiding transport semantics from interfering middleboxes, QUIC also hides them from legitimate passive network monitors. This paradox establishes the central theme of modern transport measurement: techniques must evolve from implicit passive inference (as with TCP) to advanced active probing and the analysis of explicit, protocol-defined signals (as with QUIC) [32].

3.2 Active Measurement and Scanning of QUIC Deployments

The first challenge in QUIC measurement is discovery. Given that QUIC runs on UDP, a simple TCP SYN-style liveness check is not possible.

The foundational technique for large-scale, stateless discovery was developed by [7] and [14]. This method uses the ZMap high-speed scanner 1 to send a minimal QUIC `Initial`

packet. The `version` field of this packet is intentionally set to a reserved value (e.g. `0x?a?a?a?a`). A compliant server, upon receiving this, is expected to reply statelessly with a Version Negotiation (VN) packet, which confirms QUIC support without requiring the server to commit state or complete a handshake [33].

However, this technique was found to be incomplete. Recent work demonstrated that this simple VN probe fails to detect major QUIC deployments. Using a controlled testbed, they showed that implementations such as Amazon's `s2n-quic` and `lsquic` would attempt to parse the entire `Initial` packet. Finding it malformed (e.g., missing a complete Transport Layer Security (TLS) Client Hello), these implementations would simply drop the packet rather than sending a VN reply [33].

This discovery led to a novel, evolved stateless discovery technique. The improved methodology involves crafting a fully valid QUIC `Initial` packet, including a syntactically correct Client Hello and the required 1200-byte padding, but still setting a reserved version number. This valid-but-wrong-version packet successfully passes the initial parsing steps of all major implementations and correctly triggers the VN packet response. This single change in measurement methodology was profound: it "found 2.6 M more IPv4 and 7.4 M IPv6 targets", primarily operated by Amazon [33]. This demonstrates that accurate, large-scale discovery is highly sensitive to implementation-specific parser behavior.

Beyond simple discovery, stateful scanners like QScanner 1, which is based on the `quic-go` library, are required to perform a full QUIC handshake. This stateful technique is necessary to collect richer data, such as supported QUIC versions, negotiated TLS parameters, QUIC transport parameters, and application-layer support (e.g., HTTP/3 (H3)) [14], [33].

This stateful scanning methodology revealed another critical component of modern measurement: the necessity of Server Name Indication (SNI). While the new ZMap probe successfully identified millions of QUIC-enabled IPs at Content Delivery Networks (CDNs) like Amazon, stateful QScanner probes to those same IPs without an SNI value failed, typically resulting in a timeout. However, probing the exact same IPs with a valid SNI (e.g., "a.cloudfront.net") resulted in a 97.7% handshake success rate. This demonstrates that for large-scale, multi-tenant CDN deployments, the server requires a valid SNI to select a certificate and will not complete a handshake (or even send an error) without one [33].

This finding invalidates naive IP-based stateful scanning. A correct, large-scale QUIC measurement methodology must be a multi-stage process: (1) stateless IP discovery (e.g., ZMap with the evolved probe), (2) a mapping phase to find a valid domain name for that IP (e.g., via reverse-DNS or large-scale DNS datasets), and (3) a stateful probe (e.g., QScanner) that uses the discovered domain name in the SNI field of its Client Hello [33].

Alternative discovery methods have also been analyzed, such as parsing HTTP Alternative Service (ALT-SVC) headers or the Domain Name System (DNS) HTTPS Resource Record (RR). However, these studies found that such methods were, at the time, heavily biased towards specific early adopters (e.g., Cloudflare was a dominant user of the HTTPS RR) and thus did not provide a comprehensive, unbiased view of Internet-wide deployment [14].

3.3 QUIC Server Implementation Fingerprinting

A more advanced measurement technique goes beyond discovery (is QUIC present?) to identification (which QUIC library is running?). Given the diversity of QUIC implementations, each with different performance and security characteristics, fingerprinting the server-side library is a key goal for network measurement [33].

The "QUIC Hunter" methodology, provides a novel, two-pronged approach for active QUIC library fingerprinting. The technique was developed by creating a local testbed using Docker containers to run over 18 different QUIC server implementations and observing their unique, non-configurable responses to specific probes [33].

The first technique, `CONNECTION_CLOSE` Frame Analysis, is a 1-RTT method that does not require a successful handshake. The scanner sends a QUIC `Initial` packet containing an invalid Application-Layer Protocol Negotiation (ALPN) value (e.g., the string "invalid"). A compliant server must reject this negotiation and reply with a `CONNECTION_CLOSE` frame. The QUIC specification permits this frame to contain an arbitrary, human-readable "reason phrase" string. This string is frequently hard-coded into the server library and is unique to the implementation. For example, Cloudflare's `quiche` library replies with "tls: no application protocol," whereas `lsquic` replies with "no suitable application protocol" [33]. This provides a fast and stable fingerprint.

The second technique, Transport Parameter (TP) Order Analysis, is used if a handshake is successful or the error string is not unique. The scanner completes the handshake and inspects the `quic_transport_parameters` extension returned by the server. The IETF standard does not specify the order in which these parameters must be encoded. Consequently, the order is an incidental artifact of the server's source code (e.g., the declaration order of fields in a struct). This order is static for most implementations; for example, Amazon's `s2n-quic` was found to always use the order `4-6-7-8-0-f`. Some libraries, such as Google Quiche and Akamai QUIC, explicitly randomize their TP order as a security feature. This behavior itself becomes a fingerprint; a re-probe of a server running Google Quiche will yield a different TP order, distinguishing it from a server running HAProxy, which has a static order that collides with one of Google's permutations [33].

This two-pronged active measurement technique proved highly effective. It successfully identified 18 distinct QUIC libraries deployed on over 8.0 million IPv4 and 2.5 million IPv6 addresses. This revealed a diverse QUIC ecosystem, with up to 12 different libraries found operating within a single Autonomous System (AS). This methodology repurposes protocol metadata and error-handling logic as new, robust fingerprinting vectors in an encrypted world [33].

3.4 Passive Measurement Techniques for QUIC

The encryption of transport headers in QUIC fundamentally breaks traditional passive measurement [32]. In response, two new categories of measurement have emerged: analysis of explicit in-band signals and analysis of unencrypted metadata.

3.4.1 Explicit In-Band Signals: The Spin Bit

QUIC (RFC 9000) introduced the optional spin bit as an explicit, in-band signal for passive RTT measurement, intended to replace the functionality of TCP timestamps. An on-path observer can measure the time between state transitions of this bit (0→1 and 1→0) as it is "spun" by the client and "reflected" by the server, with one full cycle corresponding to one RTT [32].

While this mechanism was proposed as a solution, its real-world utility was unknown until [32] conducted a novel study to measure the accuracy of the spin bit itself. The methodology involved actively establishing connections to spin-bit-enabled servers while simultaneously capturing detailed `qlog` traces. This log data provided both the RTT as perceived by a passive observer (measuring the spin edge timing) and the ground truth network RTT (as measured by the client's QUIC stack using packet-ACK timing) [32].

The study's key finding is that the spin bit is a flawed mechanism for measuring network RTT. The spin is applied by the endpoint's application layer upon processing a packet, meaning the measured RTT includes not only the network RTT but also all end-host processing delays. In modern, complex server environments, this application-level delay can be substantial. The study found that for 51.7% of connections, the spin bit drastically overestimated the actual network RTT by more than a factor of three. This identifies a significant limitation of this first-generation explicit measurement signal, as it is confounded by end-host performance [32].

3.4.2 Passive Backscatter Analysis

A second passive technique, explored by [11], leverages QUIC backscatter traffic. This methodology uses data from network telescopes (e.g., CAIDA's /9 telescope) to analyze unsolicited QUIC packets. These packets are often replies from servers to spoofed-IP addresses, typically generated during DDoS reflection attacks.

While QUIC payloads are encrypted, the public header (specifically the short header) contains the Source Connection ID (SCID) in cleartext. The researchers discovered that hypergiants like Facebook and Cloudflare use structured SCIDs, encoding internal infrastructure information—such as Worker ID, Host ID, and Process ID—directly into this public, unencrypted value. This creates a stable, passive fingerprint for their infrastructure, including off-net deployments.

Furthermore, by observing the inter-arrival time of retransmitted `Initial` packets within the backscatter traffic, this technique allows for the passive measurement of provider-specific Retransmission Timeout (RTO) strategies. This methodology provides a powerful, non-intrusive way to measure and map QUIC deployments by leveraging unencrypted metadata and protocol retransmission behavior.

3.5 Performance Benchmarking of QUIC Implementations

Understanding the performance of QUIC requires a reproducible, high-fidelity benchmarking methodology capable of isolating bottlenecks across the protocol stack. A novel framework for this purpose was developed by [34]. This work extends the existing QUIC Interop Runner (QIR), which is typically used for functional correctness testing with network emulation via `ns-3`. The new methodology adapts the QIR to orchestrate client and server containers on dedicated bare-metal hardware connected by a 10 Gbit/s link. This high-speed testbed, combined with detailed metric collection from the kernel (ethtool, netstat) and CPU (perf), allows for a precise analysis of performance bottlenecks [34].

This high-rate benchmarking methodology yielded several key findings:

1. Implementation Asymmetry: QUIC performance is not monolithic. The goodput varies dramatically based on the specific pair of client and server implementations, ranging from as low as 90 Mbit/s to 4900 Mbit/s on the same 10G link [34].
2. Bottleneck Identification: On modern CPUs with hardware-accelerated encryption (AES-NI), the primary CPU bottleneck is not cryptography. Instead, perf profiles consistently identified Packet I/O—the cost of sendmsg and recvmsg system calls to move data between the user-space QUIC application and the kernel's UDP socket—as the main performance limiter [34].
3. S-Level Tuning: The default Linux UDP Receive Buffer (`RCVBUF`) size (208 KiB) is far too small for high-speed (10G) traffic, leading to high packet drop rates at the receiver's socket. A critical finding was that increasing this buffer by at least an order of magnitude (>16x) was necessary to mitigate these drops and achieve maximum goodput [34].
4. Offloading Ineffectiveness: Because QUIC implementations operate in user-space over UDP, they do not benefit from kernel-level TCP Segmentation Offload (TSO) or Generic Segmentation Offload (GSO). In contrast, standard TCP/TLS on the same hardware achieved 8000 Mbit/s, largely due to TSO [34].

Collectively, these studies show that "QUIC benchmarking" is a non-trivial measurement technique. A robust methodology must use bare-metal hardware, control for OS-level tuning (especially RCVBUF), test a matrix of $N \times M$ client/server pairs, and measure at both the raw transport (e.g., H0.9) and application (H3) layers to obtain a complete performance profile [34].

3.6 Measurement of Specific QUIC Protocol Mechanisms

This thesis focuses on novel measurement techniques; therefore, a review of techniques for QUIC's unique mechanisms is essential.

3.6.1 Connection Migration

A key feature of QUIC is the ability to migrate a connection (e.g., from Wi-Fi to cellular) by changing the 4-tuple while retaining the Connection ID (CID), providing session persistence. Measuring server-side support for this feature, however, proved challenging [35].

[35] found that existing stateful scanners, such as QScanner, were unable to test this feature because their underlying library (`quic-go`) did not implement client-side migration support. This required the development of a new, custom stateful scanner based on Cloudflare's `quiche` library. The methodology of this new scanner is as follows: (1) Establish a successful H3 connection. (2) Verify that the server provides at least one additional CID, a prerequisite for migration. (3) Send a `PATH_CHALLENGE` frame from a new source port (simulating a path change) to trigger the server's migration validation mechanism. (4) A successful `PATH_RESPONSE` from the server confirms support for connection migration [35].

Using this novel scanner, the study found that despite rapid and widespread QUIC deployment, "some of the most popular destinations do not support connection migration yet". This demonstrates a significant gap between protocol specification and real-world feature deployment, highlighting the need for custom measurement tools to track specific protocol extensions [35].

3.6.2 Address Validation and Security Measurement

QUIC's handshake, being UDP-based, is vulnerable to amplification/reflection DDoS attacks, where an attacker sends a small, spoofed-IP packet to a server, inducing a large reply to a victim. QUIC's primary defense is the 1200-byte padding requirement for `Initial` packets, designed to ensure the client's first packet is larger than the server's potential first response [14].

[33] developed QUICforge, a security measurement framework to test the compliance of server implementations with these anti-amplification guarantees [33]. This methodology involves actively probing servers with spoofed-source-IP packets designed to trigger specific server responses:

- Version Negotiation Request Forgery (VNRF): A spoofed `Initial` packet with an invalid version number triggers a `Version Negotiation` packet to the victim.
- Connection Migration Request Forgery (CMRF): A spoofed packet initiating a connection migration triggers a `PATH_CHALLENGE` packet to the victim.

This security measurement technique found that 9 of the 13 open-source implementations tested were non-compliant with QUIC's anti-amplification limits, often due to ambiguities in the specification regarding packet retransmissions. This allowed for significant amplification factors, with a Bandwidth Amplification Factor (BAF) as high as 374x for CMRF. This work provides a formal methodology for measuring the security compliance of transport protocol implementations, a critical area of measurement.

3.6.3 0-RTT Connection Establishment

QUIC's 0-RTT (Zero Round-Trip Time) connection establishment feature allows a client to send application data in its very first flight of packets upon resuming a session, eliminating connection setup latency [36], [37]. Measurement studies have confirmed its significant impact on latency-sensitive applications. In CDN environments, 0-RTT is the mechanism that enables connection resumption across consecutive page visits, allowing H3 to "skip the connection phase". This provides a key performance benefit over H2+TCP and is critical for improving metrics like Page Load Time (PLT) and video start-up time [37]. While the benefits of 0-RTT are well-measured, the deployment rates and security implications (e.g., vulnerability to replay attacks) of 0-RTT specifically are less studied, representing a measurement gap [36].

3.7 Multipath Transport Protocols: From MPTCP to MP-QUIC

This section synthesizes the 10-year evolution of multipath transport, providing the direct context for the user's thesis on Multipath QUIC.

3.7.1 MPTCP (The 10-Year Context)

As established in Section A, the story of MPTCP measurement is defined by the challenge of middlebox interference. The required measurement technique is a complex, multi-stage active probing and filtering process using ZMap and Tracebox [29], [30]. The finding from this methodology is that MPTCPv1 (RFC 8684) deployment remains "almost non-existent" as of 2022, limited almost exclusively to Apple's infrastructure [29].

3.7.2 Multipath QUIC (The 5-Year Focus)

Multipath QUIC (MP-QUIC) was designed as the direct solution to MPTCP's fundamental flaws. Its design—running over UDP and encrypting all transport semantics—makes it immune to the TCP Option-stripping and -mirroring that plagued MPTCP [31].

The core mechanisms of MP-QUIC, as specified in the IETF draft draft-ietf-quic-multipath-17, are the targets for any novel measurement technique:

1. Path ID: An explicit, unencrypted identifier for each path, used to manage connection IDs and packet number spaces [31].
2. Per-Path Packet Number (PN) Space: This is the most critical design choice. Each Path ID is associated with its own independent PN space. This resolves a major ambiguity from MPTCP, as loss detection and congestion control state are now managed on a per-path basis [31].

3. Encrypted Path Management Frames: The protocol specifies new, encrypted frames for path management, such as `PATH_NEW_CONNECTION_ID`, `PATH_ABANDON`, and path status frames [31].

4. 0-RTT Path Establishment: Unlike MPTCP, which requires a 3-way handshake (1-RTT) to add a new subflow, MP-QUIC can immediately send data on a new path after its address is validated (e.g., via `PATH_CHALLENGE`), eliminating new-path handshake latency [31].

3.7.3 Measurement and Performance Evaluation of MP-QUIC

The foundational evaluation of MP-QUIC (De Coninck & Bonaventure, 2017) used Mininet emulation. This methodology compared an extended `quic-go` implementation against the Linux kernel's MPTCP implementation. This study established that MP-QUIC maintains MPTCP's core benefits (bandwidth aggregation, network handover) while outperforming it in lossy and low-BDP scenarios, thanks to QUIC's superior loss recovery and the lack of handshake latency on new paths [31].

More recent work has focused on measuring MP-QUIC performance for aggregating cellular networks (LTE/5G). This research found that default packet schedulers (e.g., Round-Robin) are insufficient. A key finding was that "it is not worth communicating on all the available links," as a single low-quality link can stall the entire connection (Head-of-Line blocking), making performance worse than single-path. This highlights that MP-QUIC's performance is entirely dependent on the intelligence of its packet scheduler [31].

This leads to the most advanced measurement methodology: Application-Aware Scheduling. Because MP-QUIC is a user-space protocol, the application itself can provide input to the transport-layer scheduler. A landmark study by [31], an MP-QUIC solution deployed in the Taobao short-video app, measured this concept at scale. The measurement technique was a large-scale A/B test (over 100,000 users) comparing the XLINK MP-QUIC scheduler against single-path QUIC.

The XLINK scheduler's logic was QoE-driven: it reads the client's video buffer level (an application QoE metric). If the buffer was full ($> T_{th2}$), it used only the cheap (Wi-Fi) path to save cost. If the buffer was low ($< T_{th1}$), it aggressively re-injected packets on the fast (5G) path to prevent a stall. This application-transport co-design resulted in a 23-67% reduction in video re-buffering at the cost of only 2.1% redundant traffic. This study demonstrates that the state-of-the-art in multipath transport measurement is no longer just about measuring goodput; it is about measuring application-specific QoE as a function of novel, cross-layer scheduling algorithms [31].

3.8 Summary and Research Gaps

This review of related work reveals a clear 10-year trajectory in transport protocol measurement. The field has been forced to evolve from passively inferring state from clear-text

TCP headers (a methodology that is now obsolete) to actively filtering middlebox interference (the MPTCP era). We are now in a new paradigm defined by QUIC, where encryption is the default.

This new landscape requires a new generation of measurement techniques, which this chapter has surveyed:

1. Implementation-Aware Active Discovery: Using full, valid `Initial` packets to discover deployments hidden by simple probes.
2. Stateful SNI-Aware Scanning: Combining DNS and IP-based probing to bypass CDN certificate-selection requirements.
3. Protocol-Artifact Fingerprinting: Using `CONNECTION_CLOSE` error strings and Transport Parameter ordering to identify server implementations.
4. Explicit Signal Utility Measurement: Quantifying the limitations of in-band signals like the Spin Bit, which is confounded by end-host delay.
5. Passive Metadata Analysis: Using clear-text SCIDs and RTOs from backscatter traffic to map infrastructure.
6. High-Fidelity Benchmarking: Using bare-metal frameworks to identify user-space I/O bottlenecks and the critical role of OS tuning.
7. Security Compliance Measurement: Using active "fuzzing" frameworks like QUICforge to validate implementation security against amplification attacks.
8. Application-Aware Transport Measurement: Moving beyond goodput to measure QoE as a function of cross-layer (application-to-transport) schedulers, as seen in MP-QUIC.

This comprehensive review identifies several clear research gaps that motivate the work of this thesis:

1. MP-QUIC Deployment Measurement: The user's specific focus, Multipath QUIC (defined in draft-ietf-quic-multipath-17) 27, has not yet been measured at scale in the wild. All existing studies are confined to emulated (Mininet), cellular testbed, or controlled A/B test (XLINK) 30 environments. A major gap exists to design and implement a measurement technique (e.g., an "MP-QUIC Hunter") to discover and "fingerprint" its deployment as it becomes available from major providers.
2. MP-QUIC Scheduler Inference: The IETF draft 27 intentionally does not specify a packet scheduler, leaving it to the implementation. As studies 9 clearly show, performance is entirely dependent on the scheduler's logic. This is a critical research gap. A novel measurement technique would be one that can remotely and externally infer the scheduling logic (e.g., is it RTT-based, loss-based, or QoE-aware like XLINK?) of a target MP-QUIC server.

3. Robust In-Band RTT Measurement: The Spin Bit is a "first-generation" explicit signal, and studies 1 prove it is significantly flawed by end-host delay. A novel design could propose an improved in-band signal or, more relevantly, a measurement technique that can actively probe and calibrate this end-host delay component to correct the spin bit's RTT estimate.

4. Security Measurement of Multipath: The "QUICforge" 11 work provides a methodology for single-path QUIC. This methodology can be extended. A novel measurement technique would be to design and implement probes to test the security compliance of Multipath QUIC implementations, particularly their vulnerability to resource exhaustion attacks via the path management frames (e.g., `PATH_CHALLENGE` flooding on multiple paths).

Chapter 4

Architecture and Design

This chapter presents the overall architecture of the probing framework and motivates the main design decisions. The goal is to provide a technology-independent description that can, in principle, be instantiated in any programming language or runtime. The prober is designed as a modular scanning and fuzzing framework for advanced transport and application layer protocols, with a particular focus on QUIC, Multipath QUIC, HTTP/3 and related mechanisms.

The architecture is structured around three logically separated subsystems:

1. **Domain Extractor**: a configurable pipeline that cleans, filters, and canonicalizes input domains into a well-defined target set.
2. **QUIC Lab**: a general, protocol-independent probing framework that orchestrates large-scale measurements against the target set using pluggable probes.
3. **QUIC Lab Analyzer**: an analysis layer that ingests the raw measurement outputs, aggregates them, and derives interpretable statistics and visualizations.

Figure 4.1 provides a high-level overview of the architecture and visualizes the data flow between these three systems. It is important to note that all three systems are independent and can also be run independently of each other. The subsequent subchapters will provide a comprehensive overview of all subsystems.

4.1 High-Level Architecture

At a high level, the measurement process is organized as a unidirectional pipeline:

1. The **Domain Extractor** ingests one or more raw domain sources and produces a sanitized, reproducible list of targets in a canonical format.

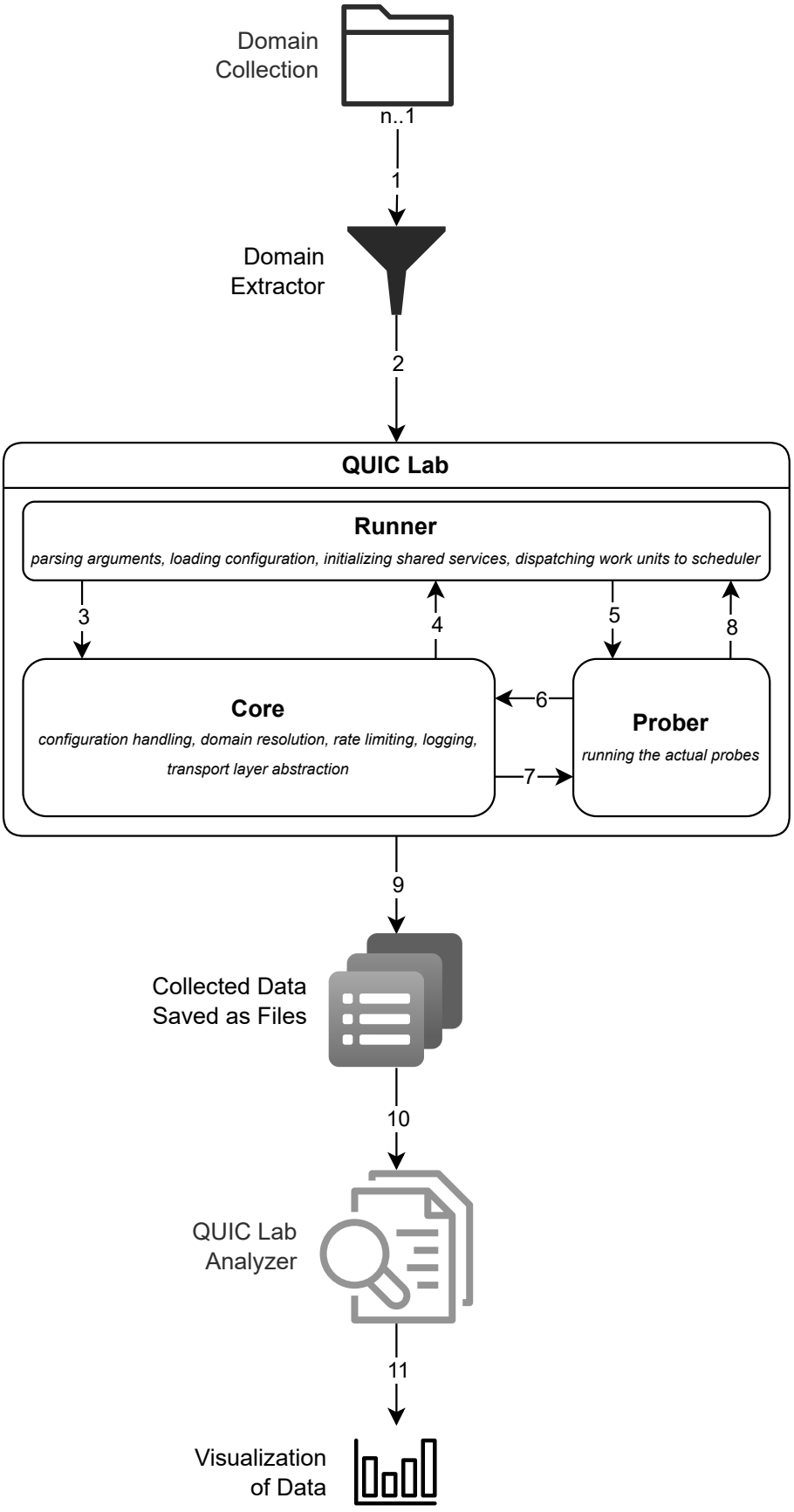

Figure 4.1: High-Level Overview of the Project

2. The **QUIC Lab Probing Framework** consumes this canonical list and executes a configurable set of probes against each target, under rate limiting and concurrency constraints to respect ethical requirements.

3. The **QUIC Lab Analyzer** consumes the outputs produced by QUIC Lab, joins them with metadata, and derives feature-level statistics.

The three subsystems are decoupled via clearly defined, file-based interfaces rather than shared internal data structures. This decoupling serves several purposes:

- **Implementation independence**: each subsystem can be implemented or reimplemented in a different language or technology stack without affecting the others, as long as the external interface contracts are preserved.
- **Extensibility**: additional domain sources, probe types, or analysis routines can be integrated without modifications to the existing components.
- **Reproducibility**: each subsystem can be re-executed independently (e.g., re-running the analyzer on the same raw outputs, or re-running QUIC Lab with a previously stored domain list).

In the following subchapters, the three subsystems and the cross-cutting design aspects—ethical measurement, portability, and measurability—are described in more detail.

4.2 Domain Extractor

4.2.1 Design Goals

The Domain Extractor is designed to solve a recurring precondition for Internet measurements: the construction of a high-quality, ethically acceptable, and reproducible set of domains. The following goals guided its design:

- **Input flexibility**: support for multiple heterogeneous domain sources (e.g., popularity rankings, manually curated lists, or top level domain zone files).
- **Sanitization and safety**: systematic removal of malformed, harmful, or inappropriate domains before they enter the probing phase.
- **Deduplication and normalization**: canonical representation of domains to avoid redundant measurements and to simplify downstream processing.
- **Configurability**: the ability to selectively enable or disable stages (e.g., filter only) without modifying code.
- **Reproducibility**: the ability to regenerate the exact same target list at a later point in time, given the same input configuration and data.

4.2.2 Pipeline

Conceptually, the Domain Extractor is modeled as a linear pipeline of transformation stages, each of which consumes a stream of domain identifiers and outputs a transformed stream. The main stages are:

1. **Source ingestion**: adapters import domains from various sources. Typical examples include:
   - Research-oriented popularity rankings such as Tranco [15].
   - Domain lists published by other measurement studies or operational communities.
   - Zone Files
2. **Normalization and validation**: each candidate domain is transformed into a canonical representation, for example by:
   - Lowercasing, trimming, and removing protocol prefixes.
   - Rejecting syntactically invalid domain names and internationalized domain names that cannot be normalized reliably.
   - Optionally mapping subdomains to a canonical parent, depending on the measurement question.
3. **Deduplication**: domains are de-duplicated across all sources to ensure that each canonical target appears at most once in the final list.
4. **Safety and blacklist filtering**: the pipeline applies blacklists to exclude domains that are known to be malicious, inappropriate, or otherwise out of scope. This stage reflects ethical constraints and reduces the risk of probing harmful infrastructure.

The output of the pipeline is a canonical domain list that serves as the sole input to the probing framework. This list is stored in a stable, machine-readable format (e.g., line-based) and is accompanied by a metrics file that tracks the amount of domains dropped per TLD and total.

4.3 QUIC Lab

The probing framework QUIC Lab forms the core of the architecture. It provides the generic functionality required to execute large-scale transport and application layer measurements against the domain list produced by the Domain Extractor. QUIC Lab is intentionally designed to be protocol-independent and extensible, with QUIC and HTTP/3 probes representing specific instantiations rather than hard-coded assumptions.

4.3.1 Design Goals

The following design goals guided the architecture of QUIC Lab:

- **Modularity**: separation of concerns between orchestration, shared infrastructure services, and probe-specific logic.
- **Protocol-independent transport abstraction**: support for multiple transport protocols (e.g., QUIC, TCP) through a common interface.
- **Probe extensibility**: simple integration of new probes without changes to the core, enabling future experimentation with additional protocols.
- **Ethical-by-design operation**: built-in rate limiting and scheduling mechanisms that enable conservative scanning behavior and prevent overload of remote systems.
- **Portability**: minimal assumptions about the execution environment, enabling deployment on different operating systems and vantage points.
- **Observability and measurability**: detailed logging and structured outputs that allow downstream analysis and validation of experimental results.

To achieve these goals, QUIC Lab is decomposed into three main architectural elements: the **Runner**, the **Core**, and the **Probes**.

4.3.2 Runner

The **Runner** acts as the entry point of the probing framework. Its responsibilities are purely orchestration-related:

- Parsing command-line arguments or other external parameters.
- Loading configuration files and constructing an effective configuration (defaults overridden by user-specified values).
- Initializing shared services (e.g., logging, scheduling, transport abstractions) provided by the Core.
- Discovering and instantiating the set of probes to be executed.
- Dispatching work units (individual domains or domain–probe combinations) to the scheduler.

The Runner is intentionally kept free of measurement-specific logic. This separation ensures that high-level orchestration can evolve independently of the probe implementations or core service internals.

4.3.3 Core Services

The **Core** offers reusable services that implement the generic cross-cutting functionality required by all probes. Conceptually, the Core comprises the following modules:

- **Configuration module**: defines the configuration schema (e.g., concurrency limits, timeouts, feature toggles) and provides typed access to configuration values. Each parameter has a documented default value and can be overridden externally without code changes.
- **Domain resolution module**: translates domains into IP addresses. This module encapsulates name resolution strategies and error handling.
- **Logging module**: provides structured logging facilities for all components. Logs are written to persistent storage rather than standard output, enabling offline analysis and post-mortem debugging. Log records include timestamps, severity levels, component identifiers, and vantage-point metadata.
- **Scheduling and rate-limiting module**: orchestrates concurrent probe execution while enforcing resource limits and ethical constraints. The module manages:
  - A configurable **concurrency** limit, bounding the number of simultaneous connections or probe executions.
  - A **requests-per-second (RPS)** parameter that caps the average rate at which new probe attempts are initiated.
  - A **burst** limit, effectively modeling a token-bucket capacity that allows short-term bursts while still enforcing the long-term RPS constraint.
  - Per-target backoff rules, specifying minimum waiting times before repeated connection attempts toward the same endpoint.

  Conceptually, the scheduler maintains queues of pending probe tasks and uses the token-bucket parameters (RPS and burst) to decide when new tasks may be admitted, while the concurrency limit bounds the number of active tasks at any given moment. This combination allows fine-grained control over both instantaneous load and long-term scanning rate.
- **Recording module**: offers an abstraction for recording measurement-specific events and custom metrics that are not captured by general-purpose logs. Probes use this module to emit structured records such as transport parameters, handshake outcomes, error codes, and feature-detection flags.
- **Transport abstraction module**: exposes a generic transport interface that hides protocol-specific details. For example, QUIC connections are managed through a transport-layer abstraction that supports connection establishment, stream management, and graceful teardown. This design permits later integration of additional transports (e.g., different QUIC stacks, TCP) without modifying probe logic.

- **Key logging module**: optionally records cryptographic session keys (e.g., TLS 1.3 keys) in a standardized format to support offline decryption of captured traffic in external tools. This capability is particularly relevant when detailed packet-level analysis with tools such as Wireshark[1] is required.

- **File handling and rotation module**: manages the persistent storage of measurement outputs. To avoid the scalability problems of maintaining millions of tiny files, the design aggregates records for multiple connections into larger files up to a configurable size limit. Once a file reaches this threshold, it is rotated: a new file is created, and the old file is sealed and renamed using a monotonically increasing sequence number. This approach balances the need for append-only, fault-tolerant logging with the operational constraints of file systems.

- **Shared types module**: defines common data types and schemas used across the framework (e.g., identifiers for probes, connections, or targets), thereby ensuring type consistency between the Core and probes.

These modules encapsulate all non-probe-specific functionality, enabling probes to focus exclusively on protocol logic and measurement strategy.

4.3.4 Probes

The **Probe** layer is the extensibility point of QUIC Lab. A probe encapsulates a specific measurement procedure—for example, testing support for QUIC connection migration, evaluating 0-RTT capability, or assessing HTTP/3 QPACK behavior.

Each probe is modeled as a state machine that progresses through a sequence of steps, such as:

1. Obtaining a domain and resolving it to one or more endpoints via the Core's resolution module.

2. Establishing a transport-layer connection using the Core's transport abstraction.

3. Conducting a protocol-specific interaction pattern (e.g., sending application-layer requests, triggering migration events, or injecting edge-case inputs).

4. Observing responses and behavior, including timeouts, error codes, and advertised protocol parameters.

5. Emitting structured records via the Recording module and logging any unexpected events.

[1] `https://wiki.wireshark.org/TLS#using-the-pre-master-secret`

Probes interact with the Core exclusively through stable interfaces, including configuration access, logging facilities, transport primitives, and recording APIs. They do not implement their own concurrency mechanisms or rate limiting; these responsibilities are handled centrally by the scheduler. This separation of concerns allows probe authors to focus solely on measurement logic without engaging with orchestration complexity, guarantees that system-wide constraints such as ethical limits and timeouts are uniformly enforced, and ensures that probes remain portable and reusable across different deployments.

4.4 QUIC Lab Analyzer

The third major subsystem, the QUIC Lab Analyzer, addresses the challenge of interpreting the large volumes of data generated by QUIC Lab. Its role is to convert raw per-connection and per-probe records into aggregated, human-interpretable results that directly address the research questions.

4.4.1 Design Goals

The design of the Analyzer is driven by the following goals:

- **Scalability**: the ability to process measurement campaigns comprising millions of connections and large volumes of structured logs.
- **Modularity**: clear separation between low-level parsing, aggregation, and presentation layers.
- **Reusability**: the capability to reuse analysis components across different experiments and vantage points.
- **Reproducibility**: deterministic analysis pipelines that can be rerun on the same input to reproduce published results.

4.4.2 Analysis Pipeline

Conceptually, the Analyzer is organized as a sequence of stages:

1. **Ingestion and parsing**: raw output files produced by QUIC Lab are read and parsed into internal data structures. The parser understands the schemas emitted by the Recording and logging modules (e.g., connection identifiers, transport parameters, error codes, feature flags).
2. **Normalization and enrichment**: records are augmented with contextual metadata, such as:

- The originating vantage point and time window.
- The domain and endpoint group (e.g., cloud provider vs. other hosts).
- The probe type and version.

3. **Aggregation and metric computation**: the Analyzer computes aggregate statistics such as:
   - Fraction of targets supporting specific protocol versions or features (e.g., QUIC version negotiation outcomes, presence of connection migration support).
   - Distributions of handshake success, error types, and performance-related metrics.
4. **Visualization and reporting**: finally, the Analyzer produces outputs in formats suitable for integration into the evaluation chapter, including:
   - Tables summarizing feature support across domains.
   - Plots and diagrams highlighting distributions and correlations.
   - Intermediate artifacts (e.g., CSV files or serialized data structures) for further manual inspection.

The Analyzer is designed to be decoupled from QUIC Lab's implementation details beyond the output schema. This decoupling allows the analysis pipeline to be reused even if the underlying probing framework is reimplemented, as long as the schema remains stable or is appropriately versioned.

4.5 Ethical Measurement and Scheduling

Large-scale Internet measurements raise ethical concerns related to potential service disruption, misinterpretation of probing activity, and unintended interaction with vulnerable systems. In accordance with the established guidelines on active measurements, ethical considerations are integrated directly into the architecture rather than treated as an afterthought.

Key design elements include:

- **Global rate limiting**: the RPS and burst parameters are chosen conservatively to avoid overwhelming remote systems. The token-bucket model bounds both the long-term average rate and the short-term peak rate of connection attempts.
- **Bounded concurrency**: the concurrency limit prevents excessive simultaneous connections from a single vantage point, which could otherwise resemble a denial-of-service attempt.
- **Per-target backoff**: repeated failures toward a specific target (e.g., connection refusals, timeouts) trigger exponential or fixed backoff intervals before further attempts, thereby reducing the risk of persistent unwanted traffic.

- **Configurable safety margins**: all ethical parameters (concurrency, RPS, burst, backoff intervals) are configurable, allowing stricter settings in more sensitive environments (e.g., when probing outside well-known cloud providers).
- **Transparent logging and opt-out capability**: logs contain sufficient detail to reconstruct what traffic was sent to which targets. This transparency facilitates incident response, should operators inquire about measurement traffic, and permits future integration of opt-out mechanisms.

By embedding these mechanisms into the scheduler and configuration system of QUIC Lab, the framework ensures that any probe executed within it automatically inherits the same ethical safeguards.

4.6 Portability and Deployment Model

The architecture assumes deployment across multiple vantage points, such as private servers in different geographic regions. To accommodate heterogeneous environments, the design follows the principle of self-contained deployment units:

- The **Runner** exposes a single entry point that can be invoked by standard process managers or scheduling systems.
- All adjustable behavior is externalized in configuration files (or equivalent configuration mechanisms), removing the need to recompile or repackage the framework for different experiments.
- Input and output are mediated via well-defined file system locations or streams, which can be mapped to different storage backends depending on the environment (e.g., local disks, network file systems, or object storage).

This approach enables straightforward packaging into containers or virtual machines when desired, without tying the architecture to a specific container or orchestration technology.

4.7 Measurability and Reproducibility

A central requirement of this thesis is the ability to quantify protocol feature support and to reproduce the measurements at a later date. To this end, the architecture incorporates several mechanisms:

- **Complete input preservation**: the canonical domain list produced by the Domain Extractor, together with its configuration manifest and references to external sources (e.g., Tranco list identifiers), is archived for each campaign.

- **Configuration snapshotting**: QUIC Lab stores a copy of the effective configuration (including default values and derived parameters) alongside the measurement outputs, ensuring that all operational parameters are documented.
- **Deterministic analysis pipelines**: the QUIC Lab Analyzer is constructed as a deterministic pipeline without hidden randomness, so that rerunning it on the same input produces the same outputs.

These design decisions collectively ensure that the toolkit does not only produce answers for a single point in time, but also supports transparent and reproducible scientific analysis.

4.8 Summary

The architecture described in this chapter decomposes the overall measurement system into three major subsystems—Domain Extractor, QUIC Lab, and QUIC Lab Analyzer—connected through stable, implementation-independent interfaces. The Domain Extractor constructs a safe and reproducible target set, leveraging research-oriented rankings such as Tranco and configurable filters. QUIC Lab provides a modular, protocol-independent probing core with strong ethical safeguards and clear extensibility points for new probes and transport protocols. The QUIC Lab Analyzer transforms raw output into aggregated, interpretable results suitable for scientific evaluation.

This separation of concerns, combined with explicit attention to ethical operation, portability, and reproducibility, yields a flexible measurement toolkit that matches the goals of the thesis and can be extended to future transport and application layer protocols beyond QUIC and HTTP/3.

Chapter 5

Implementation

This chapter describes the concrete implementation of the measurement framework, focusing on the main prober that performs large-scale QUIC scans. The implementation is written in Rust (edition 2024) and organized as a multi-crate workspace, with a clear separation between reusable core components, protocol-specific probes, and the command-line runner. Throughout the implementation, the need for tools to extract domains from various sources and later interpret the raw data created by the measurement framework arose. Therefore, in addition to QUIC Lab, the side projects Domain Extractor and QUIC Lab Analyzer were created. The source code for the entire project can be found on GitHub[1]. Section 5.1 shows the implementation of QUIC Lab in detail, followed by sections 5.2 and 5.3, which then give a detailed overview of Domain Extractor and QUIC Lab Analyzer, respectively.

Figure 5.1 provides an overview of the implementation of the entire project, including the side projects. In the following chapters, the explanations of the implementation implicitly refer to this figure, which serves as a general reference.

5.1 QUIC Lab

The Rust code is structured into three crates:

- `core`: reusable building blocks shared by all probes. This crate encapsulates configuration handling, DNS resolution, rate limiting, transport abstractions on top of TQUIC, logging, QLOG and TLS key log handling, and a generic JSONL recorder.
- `probes`: protocol-specific measurement logic. In the current prototype, it contains an HTTP/3 probe that drives TQUIC's `Http3Connection` on top of the QUIC transport exposed by `core`. The probes crate was designed to be easily extendable with more probes.

[1] `https://github.com/QUIC-Lab/quic-lab`

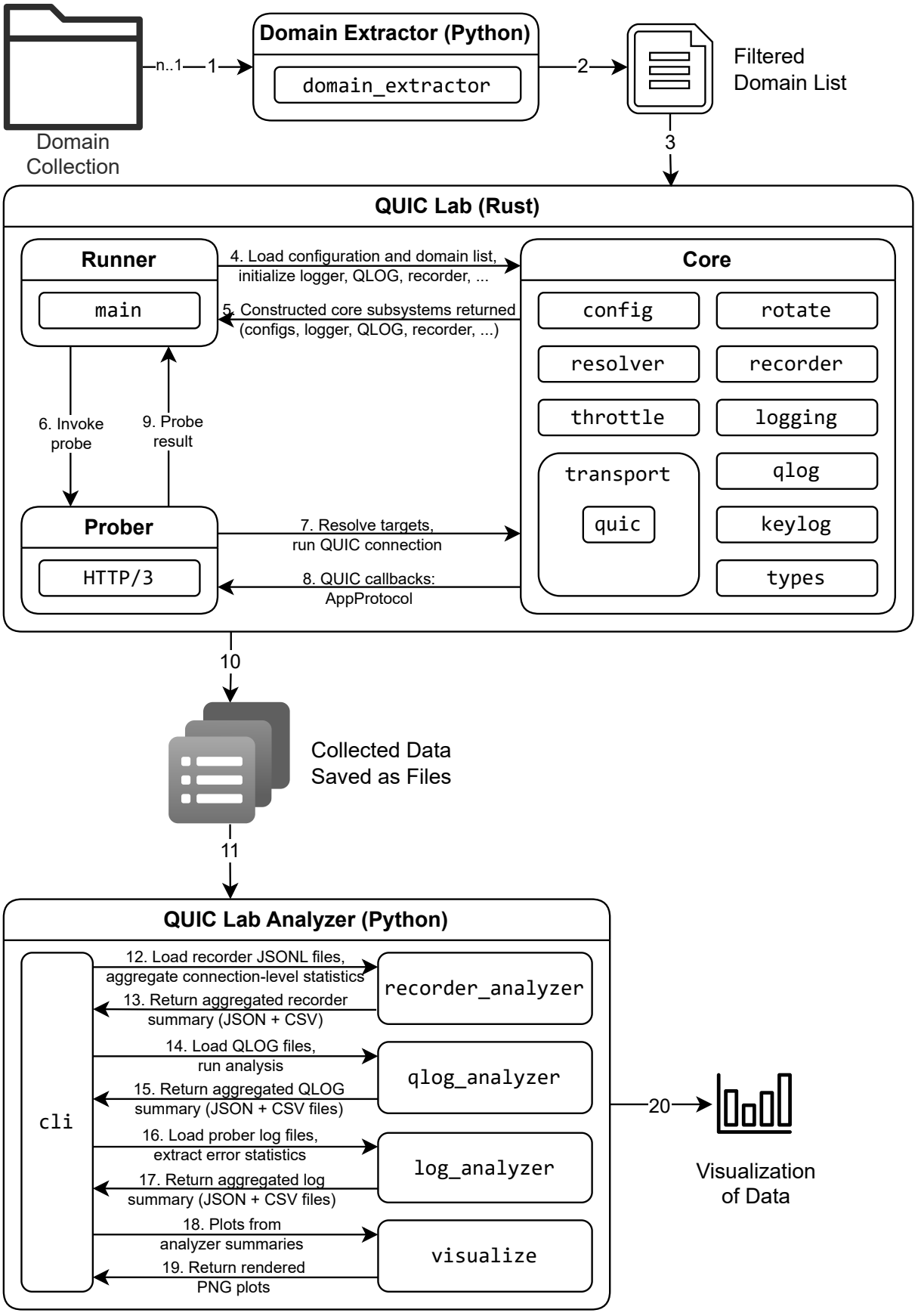

Figure 5.1: Implementation Overview of the entire Project

- `runner`: the main executable. It loads configuration and domain lists, configures the global logging and output sinks, instantiates the rate limiter and thread pool, and drives the probe in parallel over all domains.

The code uses the following external libraries in addition to the Rust standard library:

- `tquic`[2]: QUIC and HTTP/3 implementation used as the underlying transport engine. `tquic` also supports an early draft of the Multipath Extension for QUIC (draft-ietf-quic-multipath-05).[3]
- `mio`[4]: low-level I/O event loop and UDP socket abstraction for driving the TQUIC endpoint.
- `rayon`[5]: data-parallel thread pool used to process domains in parallel.
- `governor`[6]: rate limiter, used for global requests-per-second throttling.
- `tracing-subscriber`, `tracing-appender`, and `tracing-log`[7]: structured, rotating logging.
- `serde`[8], `serde_json`[9], `toml`[10]: configuration and recorder file serialization.
- `indicatif`[11]: user-friendly progress bar for interactive runs.

Packaging and CI/CD are handled by a GitHub Actions workflow that builds multi-architecture Docker images and publishes them to the GitHub Container Registry. The utilization of multi-architecture images is a strategy employed to ensure the availability on an even a broader range of systems. `Dependabot` is configured to monitor both Cargo dependencies and GitHub Actions versions and to open weekly update suggestions.

5.1.1 Core

The `core` crate collects all functionality that is independent of a specific probe. This section describes its main components.

[2] https://github.com/Tencent/tquic
[3] https://github.com/Tencent/tquic/discussions/379
[4] https://github.com/tokio-rs/mio
[5] https://github.com/rayon-rs/rayon
[6] https://github.com/antifuchs/governor
[7] https://github.com/tokio-rs/tracing
[8] https://github.com/serde-rs/serde
[9] https://github.com/serde-rs/json
[10] https://github.com/toml-rs/toml
[11] https://github.com/console-rs/indicatif

Configuration Management

Configuration is modelled by the `RootConfig` structure in `config.rs`. It is deserialized from a TOML file via `serde`:

- **SchedulerConfig**: controls concurrency and throttling.
  - `concurrency`: desired number of worker threads. A value of zero selects an automatic mode where the runner derives a thread count from the number of available CPU cores.
  - `requests_per_second`: global rate limit (RPS) across all probes. A value of zero disables throttling.
  - `burst`: short-term burst capacity for the rate limiter, expressed as additional tokens.
  - `inter_attempt_delay_ms`: delay between two consecutive connection attempts to the same domain when multiple `ConnectionConfig` profiles are tried in sequence.
- **IOConfig**: controls file system layout for input and output.
  - `in_dir`, `domains_file_name`: directory and file name for the input domain list.
  - `out_dir`: base directory for all outputs (logs, recorder files, QLOG, keylog, session files).
- **GeneralConfig**: enables or disables various output channels.
  - `log_level`: global log level.
  - `save_log_files`,
    `save_recorder_files`,
    `save_qlog_files`,
    `save_keylog_files`,
    `save_session_files`: boolean switches that control whether the corresponding sub-systems are initialized.
- **ConnectionConfig**: encodes all parameters for one connection attempt:
  - Application-level knobs: `port`, `path`, `user_agent`, `alpn`.
  - TLS behavior: `verify_peer`.
  - IP family preference: `ip_version` (`auto`, `ipv4`, `ipv6`).
  - QUIC transport parameters:
    - `max_idle_timeout_ms`
    - flow control limits for streams and connection
    - `max_ack_delay`
    - `active_connection_id_limit`

    * send_udp_payload_size
    * max_receive_buffer_size
  - Multipath flags for TQUIC: enable_multipath and multipath_algorithm (minrtt, roundrobin, or redundant).

Defaults are provided via #[serde(default)] and small helper functions for each field. This ensures that even a minimal configuration file yields a valid RootConfig. If no connection attempt is specified explicitly, read_config inserts a default ConnectionConfig so that the system always has at least one profile to try.

The domain list is consumed lazily via read_domains_iter, which returns an iterator over trimmed, non-empty lines while ignoring everything after a # comment character. This allows the domain extractor to annotate the list while keeping it readable.

DNS Resolution and IP Version Handling

The resolver.rs module encapsulates address resolution and IP family selection via the IpVersion enum defined in types.rs. It implements three functions:

- resolve_peer: resolves a single (host,port) pair and filters the result to the requested IP family (IPv4 or IPv6).
- resolve_peers_for_both: resolves both A and AAAA records and returns at most one IPv4 and one IPv6 address each.
- resolve_targets: orchestrates the above and returns a vector of (IpVersion, SocketAddr) pairs depending on ip_version:
  - For auto, it attempts to resolve both IPv4 and IPv6, returning the available families, which allows the probe to try both sequentially.
  - For an explicit family, it resolves a single address and normalizes the family based on the actual result.

This abstraction decouples the probe logic from low-level DNS APIs and keeps the family selection logic in one place.

Rate Limiting

Global throttling is implemented by the RateLimit wrapper in throttle.rs. Internally, it wraps a governor::DefaultDirectRateLimiter with a per-second quota and an explicit burst capacity:

- RateLimit::per_second(rps, burst) constructs a shared limiter with a per-second quota of rps tokens and an additional burst capacity of burst tokens. A minimum burst of 1 is enforced to avoid edge cases. Passing rps = 0 yields a disabled limiter.

- `until_ready` blocks the caller until a token is available. The prober invokes this before each connection attempt, so the global rate is enforced across all worker threads and all domains.

By decoupling *concurrency* (number of worker threads) from *RPS* (global rate), the implementation can exploit parallelism without overloading the network or remote servers.

Transport Layer Abstraction

Given the central focus of this thesis on novel transport layer protocols, as of 2025 the implementation is limited to QUIC. However, the application was designed to be modular and extendable. Consequently, the framework can be extended by other transport layer protocols, such as TCP and UDP. The current transport-level interactions with TQUIC are implemented in `transport/quic`, which splits into:

- **QuicSocket** (`mod.rs`): a thin wrapper around `mio::net::UdpSocket` that handles UDP socket creation, registration with `mio`'s `Registry`, and `send_to`/`recv_from` operations. It uses a `Slab` and a hash map to associate local addresses with socket identifiers, allowing TQUIC to bind multiple local addresses if needed. `QuicSocket` implements TQUIC's `PacketSendHandler` trait, so the QUIC endpoint can offload packet sending directly to it.
- **Client and event loop** (`quic.rs`): the `Client` struct bundles the TQUIC `Endpoint`, the `mio::Poll` object, the `QuicSocket`, a shared `ClientContext`, and a receive buffer. The event loop repeatedly:
  1. calls `endpoint.process_connections()` to let TQUIC handle timers and state,
  2. polls `mio` for readable UDP events,
  3. feeds received datagrams back into TQUIC via `endpoint.recv`, and
  4. invokes `endpoint.on_timeout` to handle timer expirations.

  The loop terminates when the `ClientContext` marks the connection as finished, which happens when TQUIC signals that the QUIC connection has been closed.

The connection life-cycle is driven by a `ClientHandler` that implements TQUIC's `TransportHandler` trait. It receives callbacks such as `on_conn_created`, `on_conn_established`, `on_stream_readable`, and `on_conn_closed`. Instead of hard-coding any specific application protocol, `ClientHandler` delegates application-level behavior to an implementation of the `AppProtocol` trait, as shown in listing 5.1.

This separation allows multiple probes (e.g., HTTP/3, MASQUE, or protocol-independent QUIC measurements) to reuse the same transport engine by providing different `AppProtocol` implementations.

The function `run_probe` is a small convenience wrapper around `open_connection`, which constructs a `Client` with the given configuration and application protocol and runs its event loop until the connection terminates.

```rust
pub trait AppProtocol {
 fn on_connected(&mut self, _conn: &mut Connection){}
 fn on_stream_readable(&mut self, _conn: &mut Connection, _stream_id: u64){}
 fn on_stream_writable(&mut self, _conn: &mut Connection, _stream_id: u64){}
 fn on_stream_closed(&mut self, _conn: &mut Connection, _stream_id: u64){}
 fn on_conn_closed(&mut self, _conn: &mut Connection){}
}
```

Listing 5.1: Implementation of AppProtocol Trait

Session Resumption and Sharding

To enable TLS session resumption across runs and avoid reperforming costly handshakes, the implementation stores and reloads session tickets:

- On connection creation, `ClientHandler` attempts to locate a session file for the current host in `session_files/`, using a small sharding helper `shard2` that hashes the host name and maps it into a two-level directory tree (two bytes of the hash, printed as hexadecimal). This keeps any single directory from containing too many session files.
- On connection establishment or closure, if sessions are enabled, TQUIC's `conn.session()` is serialized to `{host}.session` in the corresponding shard directory.

This mechanism is fully independent of the probe: it can be enabled or disabled via configuration and transparently accelerates repeated scans against the same set of domains.

Rotating File Writer

The `rotate.rs` module provides a generic `RotatingWriter<H>` for size-bounded log files. It maintains a current active file `base` and renames it to `base.1`, `base.2`, etc., once the configured size threshold is exceeded. Each writer can be associated with a `NewFileHook` that runs exactly once on newly created empty files. This hook is used, for example, to prepend a JSON-SEQ header to each QLOG file.

All higher-level output subsystems (QLOG, key log, recorder, and the main log) are implemented on top of `RotatingWriter`, which centralises the rotation logic and ensures that individual records are not split across files: each logical record is serialized into a contiguous buffer that is passed as a single `write` call.

Measurement Metadata and Recorder

The `types.rs` module defines small serialisable data structures used for measurement metadata, in particular `MetaRecord` and `BasicStats`. When a connection closes, the `ClientHandler` extracts statistics from TQUIC via `conn.stats()` and stores them as a JSON object through the `Recorder`. A `MetaRecord` instance contains the host and peer address, the negotiated ALPN, indicators for handshake success, any local or peer close reasons, a flag denoting whether multipath was enabled, and an embedded `BasicStats` object. The `BasicStats` structure records the number of bytes and packets sent, received, and lost.

The `Recorder` in `recorder.rs` writes one JSON record per line into a size-rotating file `quic-lab-recorder.jsonl`. Each record is of the form:

```
{"key": "<trace_id>", "value": { ... MetaRecord ... }}
```

where `key` is the TQUIC trace identifier of the connection. The recorder is optional and can be disabled via configuration; in that case, calls become no-ops.

Logging

The `logging.rs` module configures the global logging pipeline. At start-up, the runner calls `init_file_logger`, which creates the directory `<out_dir>/log_files` if it does not already exist, constructs a `RotatingWriter` for the main log file `quic-lab.log` with a size limit of 128 MiB, wraps this writer in a thread-safe adapter backed by a non-blocking `tracing_appender` channel, and installs a `tracing_subscriber` registry that respects the configured log level (or a `RUST_LOG` environment override) while suppressing verbose dependencies such as TQUIC. Using a non-blocking writer ensures that logging cannot become a bottleneck or add noticeable latency to the measurement process.

Global QLOG Aggregation and Minimization

The `qlog.rs` module implements a two-stage QLOG pipeline:

1. **Per-connection writer** (`PerConnSqlog`): Each TQUIC connection is configured with a `PerConnSqlog` instance as its QLOG sink in `on_conn_created`. This object implements `Write` and receives a bytestream in JSON-SEQ format from TQUIC (each record is preceded by the ASCII Record Separator 0x1E and terminated by a newline).
    - It buffers bytes, extracts full `RS...LF` frames, and JSON-parses the payload into a `serde_json::Value`.
    - If the QLOG event has no `group_id`, it injects the trace identifier of the connection.

- It enforces monotonically increasing timestamps per connection: if the incoming `time` field is not strictly larger than the previous one, it is adjusted by a small epsilon.
- Depending on the compile-time constant `MINIMIZE_QLOG`, it passes the event through a minimiser that drops unnecessary fields and entire event types.

2. **Global multiplexer** (`QlogMux`): The global `QlogMux` instance, initialized once per run, writes all events into a single JSON-SEQ file `quic-lab.sqlog` (again managed by `RotatingWriter`). A `QlogHeaderHook` ensures that each new file begins with a QLOG header that sets the format to `"JSON-SEQ"`, defines a vantage point, and establishes a reference time.
   - The mux maintains its own per-group timestamp map to ensure monotonically increasing times across file rotation.
   - It drops any incoming frames that look like QLOG file headers from TQUIC (to avoid nested headers) by scanning for known keys (`"qlog_format"` or `"file_schema"`).
   - For each event, it writes a complete JSON object framed as `RS JSON LF`.

The minimization function `qvis_minimize_in_place` is tuned for compatibility with qvis and custom statistics scripts. It preserves:

- all `meta:*` and `loglevel:*` events (with heavy `raw` fields removed),
- parameter events (`*:parameters_set`),
- error-related and connection loss events (with payload trimming),
- `"recovery:packet_lost"` events from the recovery namespace, and
- packet events (`"quic:packet_sent"`/`"quic:packet_received"`) with a reduced header (only packet type, number, and CID lengths), raw length and payload length, and simplified frame descriptors (`frame_type` and optional `stream_id` only).

Noisy events such as `"quic:stream_data_moved"` are dropped entirely. All other events are pruned to remove nested `raw` fields and excessive per-frame metadata. This significantly reduces QLOG volume when scanning large domain sets while keeping the information needed for most analyzes.

TLS Key Logging

TLS key logging is implemented in `keylog.rs`. A process-wide `KeylogSink` is initialized as a rotated writer `quic-lab.keylog` under `keylog_files/`. Each connection receives a `PerConnKeylog` writer, which buffers bytes from TQUIC's keylog callback and forwards complete lines to the global sink. Again, writes are buffered and flushed periodically to limit overhead.

Exported key logs can be used to decrypt QUIC traffic in external tools such as Wireshark[12], if needed for offline debugging or validation.

5.1.2 Probe

The `probes` crate contains an HTTP/3 probe that is implemented as a thin application layer on top of the generic QUIC transport described above.

Application Protocol Hook

The main component is the `H3App` struct in `h3.rs`, which implements the `AppProtocol` trait:

- It stores the target host, path, and user agent string.
- It maintains an optional `Http3Connection` object from TQUIC, the identifier of the request stream, and simple state variables such as the observed HTTP status code and whether response headers have been seen.

Upon `on_connected`, `H3App` performs the following steps:

1. Constructs a default `Http3Config`.
2. Creates an HTTP/3 connection on top of the already established QUIC connection via `Http3Connection::new_with_quic_conn`.
3. Opens a new unidirectional HTTP/3 stream via `stream_new`.
4. Sends a minimal `GET` request with the following headers:
   - `:method = GET`
   - `:scheme = https`
   - `:authority = <host>`
   - `:path = <path>`
   - `user-agent = <configured user-agent>`
   - `accept = */*`

   The headers are sent with the FIN flag set, so no request body follows.

In `on_stream_readable`, the application drives the HTTP/3 state machine by repeatedly calling `h3.poll(conn)` until TQUIC signals that no more events are available (via an internal `Done` error). For each event:

[12] `https://wiki.wireshark.org/TLS#using-the-pre-master-secret`

- On `Http3Event::Headers`, it iterates over the received headers and extracts the `:status` pseudo-header if present, parsing it into a `u16` status code. If the header event carries the FIN flag, the probe closes the HTTP/3 stream and initiates a graceful QUIC connection close with a zero error code.
- On `Http3Event::Data`, it drains and discards the response body by repeatedly calling `recv_body`.
- On `Http3Event::Finished`, it closes the HTTP/3 stream and closes the QUIC connection as above.

The implementation deliberately keeps the HTTP/3 logic minimal: it retrieves only the status code and validates that a syntactically correct HTTP/3 response is received. The status code is stored in memory and logged via the debug logger upon connection close; it can be persisted later via the recorder if needed.

Probe Orchestration and Connection Attempts

The public entry point for the HTTP/3 probe is visible in listing 5.2.

```
pub fn probe(
    host: &str,
    scheduler_config: &SchedulerConfig,
    io_config: &IOConfig,
    general_config: &GeneralConfig,
    connection_configs: &[ConnectionConfig],
    rl: &RateLimit,
    recorder: &Recorder,
) -> Result<()>
```

Listing 5.2: Public Entry Point of HTTP/3 Probe

For each `ConnectionConfig` profile, it performs the following steps:

1. Resolves the host to one or more target addresses via `resolve_targets`, taking into account the configured IP family preference.
2. Iterates over all target addresses of this attempt:
   (a) Calls `rl.until_ready()` to respect the global rate limit.
   (b) Constructs a new `H3App` with the host, path, and user agent of the current profile.
   (c) Invokes `run_probe`, which establishes a QUIC connection, performs the HTTP/3 request/response exchange, and runs until the QUIC connection is closed.
   (d) If `run_probe` returns without error, the attempt is considered successful and the loop over addresses is terminated.

3. If all addresses for this profile fail and more profiles are configured, the probe sleeps for `inter_attempt_delay_ms` before proceeding to the next profile.

This pattern allows the probe to implement flexible fallback strategies, for example trying IPv6 and IPv4 in sequence, or trying different transport parameter sets or multipath configurations, while still enforcing a single global RPS limit.

A template for generic QUIC-based probes is provided in `template.rs`. It illustrates how to combine `resolve_targets`, `RateLimit`, `run_probe`, and the recorder with a minimal `AppProtocol` implementation that operates directly on QUIC streams. This template serves as a blueprint for future probes beyond HTTP/3.

5.1.3 Runner

The `runner` crate provides the binary entry point in `main.rs` and is responsible for orchestrating the entire measurement execution. It loads the configuration and the domain list, configures global logging, QLOG, keylog, and recorder sinks according to the `GeneralConfig` settings, instantiates the global rate limiter, and sets up a Rayon thread pool. Once initialization is complete, it drives the probes over all domains in parallel and manages progress reporting throughout the scan.

Start-Up and Configuration

On start-up, the runner expects an optional command-line argument specifying the path to the TOML configuration file; if omitted, it defaults to `in/config.toml`. After loading `RootConfig` via `read_config`, it initialises:

- file logging (if enabled),
- TLS key logging (if enabled),
- the global QLOG multiplexer (if enabled), and
- a `Recorder` instance for `quic-lab-recorder.jsonl` (if enabled).

The domain list is read from `<in_dir>/<domains_file_name>` via `read_domains_iter` and collected into a `Vec<String>`. An empty list is treated as a configuration error.

Concurrency and Global Rate Limiting

The runner derives the effective thread pool size from `SchedulerConfig::concurrency`:

- If `concurrency > 0`, this value is used directly as the number of worker threads.

- If `concurrency = 0`, an automatic mode determines the number of hardware threads (via `available_parallelism`) and multiplies it by a factor of ten. This heuristic reflects the fact that many probes are I/O-bound and benefit from a larger number of in-flight tasks than there are CPU cores.

Rayon's global thread pool is then configured accordingly. The global rate limiter is constructed as `RateLimit::per_second(requests_per_second, burst)` and passed by reference to all probes.

The set of domains is processed via `domains.par_iter().for_each(\dots)`. For each host, the runner invokes the probes with the shared configuration, rate limiter, and recorder. Errors from the probe are counted via an atomic counter and logged; successful probes simply increment the `processed` counter.

Progress Reporting

To provide user feedback during long-running scans, the runner chooses between two reporting modes based on whether standard output or error is attached to a terminal:

- On Teletypewriters (TTYs), it uses `indicatif`'s progress bar, showing the number of processed domains, percentage, elapsed time, estimated remaining time, processing rate (domains per second), and the current error count.
- On non-TTY environments (e.g., batch runs or CI), it spawns a dedicated reporter thread that prints a summary line every ten seconds, including processed domains, percentage, elapsed time, estimated remaining time, processing rate, and cumulative error count. Once the parallel loop completes, a final summary line is printed.

Both modes rely on atomic counters for processed domains and errors, ensuring that reporting remains lock-free with minimal overhead.

5.1.4 Continuous Integration and Containerization

To ease deployment and reproducible execution, the repository contains an automated build pipeline in `.github/workflows/docker-publish-latest.yml`. On each push to the `main` branch (or on manual trigger), GitHub Actions:

1. checks out the repository,
2. sets up QEMU and Docker Buildx for multi-architecture builds,
3. logs into the GitHub Container Registry, and
4. builds and pushes a Docker image tagged `ghcr.io/<owner>/<repo>:latest`.

Build arguments allow parameterising the project name and entry point crate (here: `PROJECT_ENTRYPOINT=runner`, `PROJECT_NAME=quic-lab`). The resulting container embeds the compiled prober together with its runtime dependencies, enabling consistent execution across different measurement vantage points (e.g., private servers and cloud VMs).

Taken together, these components realize a modular and extensible measurement engine. The `core` crate abstracts all transport, logging, and persistence concerns; the `probes` crate encapsulates protocol-specific logic on top of the generic `AppProtocol` interface; and the `runner` crate ties everything together into a scalable, rate-limited domain scanner that can be deployed as a single Docker container.

5.2 Domain Extractor

The `domain_extractor.py` script is a stand-alone tool that reduces full TLD zone files to a sorted list of unique second-level domains (SLDs) and, optionally, applies host blacklists to remove obvious unwanted targets. The implementation explicitly restricts itself to the Python standard library, which simplifies deployment on arbitrary measurement vantage points.

5.2.1 Streaming Zone File Parsing and SLD Extraction

The extractor processes very large zone files in a fully streaming manner. Input files are discovered in the specified folder based on their extension (`.txt`, `.gz`, or `.txt.gz`), and the TLD is inferred heuristically from the filename (for example, `com.txt.gz` implies the TLD `com`). Each zone file is then opened as a text stream: compressed files are read via `gzip.open`, while plain text files use standard file I/O, ensuring that no file is ever loaded into memory in its entirety. The core parsing routine `parse_slds_from_lines(lines, tld)` implements a lightweight zone-file reader. It keeps track of the current `$ORIGIN` directive and honours the special owner `"@"` representing apex records, skips comments and `$TTL` directives, constructs the fully qualified domain name for each owner, normalizes it, and reduces it to an SLD using `fqdn_to_sld(fqdn, tld)`. The latter enforces that the name ends with the expected TLD and returns only the immediate child domain (for example, transforming `www.api.example.com` into `example.com`).

To provide precise progress reporting even for multi-gigabyte files, the extractor uses specialized byte-based iterators (`_iter_lines_bytes_progress_txt` and `_iter_lines_bytes_progress_gz`). These iterators read text lines while periodically sampling the underlying byte offset using `tell()`, and they forward the measured byte deltas to a shared `Progress` object. The progress bar thus displays a throttled, compact line showing percentage completion, domains processed per second, and elapsed time. Because progress is derived directly from byte positions rather than estimated line counts, the resulting per-file progress is deterministic and accurate.

5.2.2 Deduplication via Temporary SQLite Store

Because different zone files may contain overlapping SLDs, the extractor performs deduplication in a separate layer backed by SQLite. A context manager `domain_db()` creates a temporary SQLite database in the system's temporary directory with a single table `domains(domain TEXT PRIMARY KEY, tld TEXT)`. Parsed SLDs are first collected in a Python list and then periodically flushed into this table using batched `INSERT OR IGNORE` operations, which keeps per-row overhead low. Several SQLite pragmas—`journal_mode=WAL`, `synchronous=OFF`, and `temp_store=MEMORY`—are applied to maximize throughput at the cost of durability, meaning a crash may lose the most recent batch but a re-run is always safe. Once all zone files have been processed, `dump_domains(conn, out_path)` retrieves all distinct domains in sorted order and writes them sequentially to the output text file. This design effectively turns SQLite into a disk-backed set abstraction that scales to hundreds of millions of domains while keeping the Python process's memory usage low.

5.2.3 Blacklist Integration and Suffix-Based Filtering

To reduce the risk of probing domains that are clearly unwanted—such as advertising, tracking, or otherwise inappropriate hosts—the extractor can optionally apply blocklists obtained from Firebog[13]. This filtering step is fully optional and controlled by the `--mode` flag. Blacklist acquisition is performed by downloading Firebog's index pages (for example, `type=nocross` or `type=adult`), extracting the listed URLs, and retrieving each referenced hosts file using only `urllib.request`, with a byte-level progress bar indicating download progress. The downloaded hosts files are then parsed by `parse_hosts_line`, which supports common formats such as `"0.0.0.0 example.com"` or simply `"example.com"`, while ignoring comments and IP address entries. All valid domains are normalized and added to a Python `set`, which forms the blacklist. Filtering is based on suffix matching: the function `suffix_blacklisted(domain, blacklist)` checks whether the domain itself appears in the blacklist or whether any of its parent domain suffixes—obtained by progressively removing labels from the left—match an entry. Consequently, an entry such as `"example.com"` blocks both `"example.com"` and any of its subdomains, such as `"www.example.com"`. In *extract+filter* mode, this suffix-based filtering is applied during zone-file parsing, ensuring that only non-blacklisted SLDs are inserted into the SQLite store. In *filter-only* mode, the extractor reads a pre-existing `domains.txt`, applies the same blacklist evaluation, and produces a filtered `domains_filtered.txt`.

5.2.4 CLI Modes and Metrics Generation

The extractor offers three modes, implemented by `mode_extract_and_filter`, `mode_extract_only`, and `mode_filter_only`:

[13] `https://firebog.net`

- **Extract + filter (default):** read all zone files, derive SLDs, apply blacklists, deduplicate via SQLite, and write `domains.txt`.
- **Extract only:** same as above, but without any blacklist; still deduplicates and writes `domains.txt`.
- **Filter only:** assume `domains.txt` already exists in the given folder; apply blacklists and write `domains_filtered.txt`.

Each operating mode records detailed metrics that capture both per-TLD and global behavior. For every TLD, the extractor tracks how many SLDs were `extracted`, how many were `kept`, and—when filtering is enabled—how many were `filtered`. It also measures the processing time per TLD and derives throughput values, such as extracted-per-second and kept-per-second rates. After processing all TLDs, the script computes global totals, including the overall runtime. All collected metrics are written as a JSON document named `metrics.json` in the input directory (or another user-specified location), ensuring reproducibility and enabling detailed post-hoc analysis. The command-line interface is provided by `parse_args()` and exposed via `main()`, allowing the tool to be invoked directly with `python3 domain_extractor` together with the required arguments.

5.3 QUIC Lab Analyzer

The second Python component is a modular analysis toolbox that processes the measurement artefacts produced by QUIC Lab: recorder JSONL files, QLOG streams, and rotating log files. The project is organized into four main modules plus a CLI entry point:

- `recorder_analyzer.py`: aggregates connection-level statistics from recorder JSONL files.
- `qlog_analyzer.py`: performs event-level and transport-parameter analysis on QLOG files.
- `log_analyzer.py`: extracts error statistics from textual prober logs.
- `visualize.py`: renders selected metrics as static PNG plots using Matplotlib.
- `cli.py`: command-line orchestration and output directory management.

The toolbox is designed to be independent of the prober implementation language. It assumes only a file layout (recorder files, qlog files, and log files in separate directories) and specific output formats as defined in the architecture.

5.3.1 CLI Orchestration

The main entry point `cli.py` coordinates the complete analysis:

- It accepts a `--root` directory (defaulting to the current directory of the analysis project) that must contain:
  - `recorder_files/` with `quic-lab-recorder.jsonl*`,
  - `qlog_files/` with `quic-lab.sqlog*`,
  - `log_files/` with `quic-lab.log*`.
- It creates an `--out` directory (default `analysis_output/`) for derived summaries and plots.
- It delegates to the analyzers in a fixed order:
  1. process recorder files,
  2. process QLOG files (possibly using multiple worker processes),
  3. process log files,
  4. compute a cross-set summary of connection identifiers present in recorder and QLOG,
  5. optionally generate plots (unless `--no-plots` is specified).

The CLI configures Python's `logging` module with timestamps and log levels, so all steps are traceable.

5.3.2 Recorder Analysis

The `RecorderAnalyzer` ingests the JSONL output generated by QUIC Labs recorder component. Each line is expected to be a JSON object with a `key` (connection identifier) and a `value` containing metadata such as handshake success, ALPN, and QUIC close reasons.

Its design is intentionally simple and streaming-based:

- All matching files `quic-lab-recorder.jsonl*` are processed sequentially. Lines are parsed one by one; invalid JSON lines are skipped.
- For each record, the analyzer:
  - increments a global `total_records` counter,
  - collects all unique `key` values into a `group_ids` set (used later for cross-correlation with QLOG),
  - updates `handshake_ok_counts` (number of successful vs. failed handshakes),

  - updates `enable_multipath_counts` to quantify how often multipath was activated,
  - records the distribution of ALPN values in `alpn_counts`, treating missing ALPN as `"<none>"`,
  - extracts QUIC close error codes (from human-readable strings such as `"error_code=0x15"`) via a small regular expression and tallies them for both peer-initiated and local closes.

- At the end, the analyzer writes:
  - a JSON summary `recorder_summary.json` with all aggregated counters and the number of unique group identifiers; and
  - CSV files for ALPN distributions and close error-code histograms, which are straightforward to import into external tools.

5.3.3 QLOG Analysis

The `QlogAnalyzer` handles the potentially very large QLOG streams generated by the prober's QLOG multiplexer. These are stored as JSON-SEQ files `quic-lab.sqlog*`, where each event is encoded as a JSON object prefixed by the ASCII record separator.

Key design aspects include:

- **Parallel processing:** depending on the `--workers` setting, QLOG files are processed either sequentially or via a `ProcessPoolExecutor`. Each worker runs the helper function `_process_qlog_file(path)`, which returns a picklable dictionary of aggregated counters.
- **Streaming event loop:** `_process_qlog_file` reads each file line by line, strips the record separator, and attempts to parse JSON. Invalid lines are counted in `invalid_events` but otherwise ignored.
- **Event-level aggregation:**
  - `total_events` counts all valid events.
  - `event_name_counts` records the frequency of each event name.
  - `error_event_counts` focuses on events whose name suggests an error, closed connection, or connection loss.
  - `path_event_counts` focuses on `"quic:path_..."` events, which are relevant for multipath behavior.
- **Packet- and frame-level analysis:**
  - For `"quic:packet_sent"` and `"quic:packet_received"` events, the analyzer extracts direction, packet type, and a best-effort packet size (from `packet_size`, `payload_length`, or `length` fields).

  - Packet types are counted by direction in `packet_type_counts`, and cumulative sent and received bytes are tracked in `total_bytes_sent` and `total_bytes_received`.
  - For each packet, the analyzer iterates through any attached `frames` list and tallies frame types in `frame_type_counts`.

- **Transport-parameter distributions:**
  - For `"quic:parameters_set"` events with `owner="remote"`, the analyzer iterates over all parameters and counts the frequency of each value in `transport_param_counts`, using a `defaultdict(Counter)`.
  - Values are kept as scalars where possible; complex values are stringified.
- **Connection identifiers:** any string-valued `group_id` field is added to a set of group identifiers for later cross-correlation with the recorder.

After processing all files, `QlogAnalyzer` merges the worker-level results and writes:

- a JSON summary `qlog_summary.json` with aggregate event counts, packet and frame distributions, transport parameter distributions, total bytes, and the number of unique group identifiers; and
- a series of CSV files:
  - global event name counts,
  - frame-type counts,
  - packet-type counts (with direction), and
  - per-parameter distributions (`qlog_transport_param_<name>.csv`).

Experiments with qvis for Visual Inspection of Individual Connections

To complement the large-scale, machine-driven QLOG analysis described above, several experiments were conducted using qvis, an interactive visualization toolkit specifically designed for inspecting the behavior of QUIC and HTTP/3 connections. qvis is part of the broader qlog ecosystem and provides a rich set of visual inspection tools, including sequence diagrams, multiplexing diagrams, congestion graphs, and packetization diagrams. These tools translate qlog event streams into domain-specific visualizations that reveal handshake progress, stream scheduling, packetization efficiency, congestion-control dynamics, packet loss, and reordering patterns [38].

For exploratory experiments, selected per-connection qlog traces generated by the prober were transformed into qvis-compatible JSON using a custom conversion script (`qlog2qvis.sh`). The script converts the prober's JSON-SEQ QLOG output into a legacy qlog-0.3 JSON structure expected by qvis and optionally applies the same event-minimization rules used during Rust-side QLOG generation. This makes it possible to inspect individual connections visually without modifying qvis itself.

One example output is shown in fig. 5.2, illustrating a complete end-to-end QUIC + HTTP/3 exchange. The diagram reveals key handshake events, packet-level bidirectional traffic, RTT evolution, and stream scheduling behavior, in line with the visualization capabilities documented in [38]. In particular, the sequence diagram highlights packet ordering and the timing relationship between client/server events; this aligns with the intended use of the tool and emphasizes qvis's ability to expose handshake deadlocks, loss episodes, reordering, and congestion-control anomalies.

While qvis proved highly useful for validating and debugging individual connections, the experiments also confirmed a major design limitation: qvis is not intended for large-scale or high-volume analysis. The tool operates entirely on single-connection JSON logs and does not support aggregated, multi-connection analyzes or batch visualization. As a consequence, qvis is an effective tool for manual inspection, teaching, and debugging—as intended by its authors—but it cannot be scaled to millions of domains or thousands of connection traces. This limitation is explicitly acknowledged in the qvis literature, where the focus is on interactive, human-driven analysis rather than automated or statistical processing across large datasets [38]. For this reason, qvis was used only as a qualitative inspection tool in this project, whereas all large-scale quantitative evaluations rely on the Python-based QLOG analysis pipeline described earlier.

5.3.4 Log-File Analysis

The `LogAnalyzer` concentrates on extracting and classifying error messages from the prober's textual logs. It processes all files matching `quic-lab.log*` and parses them sequentially, line by line. DNS resolution errors are detected by the presence of the substring `"ERROR: failed to lookup address information:"`, and for each occurrence the analyzer increments both a general DNS-error counter and a counter for the specific error message extracted from the line. Connection-related errors are identified through lines containing both `"connect"` and `"err:"`; in these cases, the substring following `"err:"` is extracted and tallied as the connection error message. Any remaining line containing the keyword `"ERROR"` is categorized as `other_error`. After processing all log files, the analyzer produces a JSON summary file `logs_summary.json` that reports overall error counts and histograms for DNS and connection errors. Additionally, it generates two CSV files—`logs_dns_error_counts.csv` and `logs_connect_error_counts.csv`—that contain per-message error counts sorted by frequency.

5.3.5 Cross-Correlation and Visualization

After the individual analyzers have completed, the CLI derives a cross-summary of all group identifiers observed across the different data sources. It counts how many group IDs appear in the recorder output, how many appear in the QLOG files, and determines the size of their intersection, including the identifiers that occur exclusively in one of the two. This information is written to `cross_summary.json` and provides a basis for evaluating how fully the QLOG data aligns with the recorder's coverage.

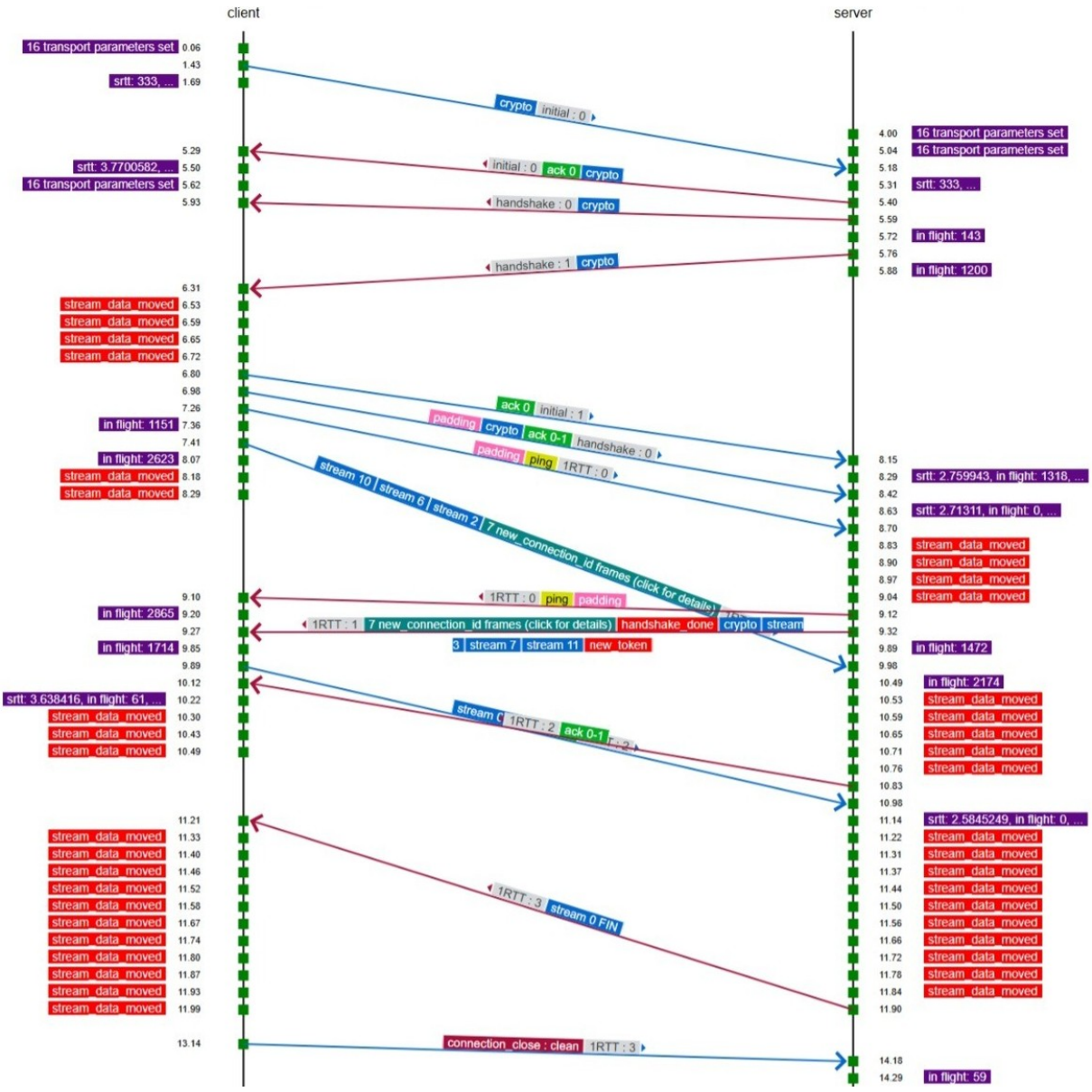

Figure 5.2: Example qvis sequence diagram generated from a single connection. Visualization based on [38].

The `Visualizer` renders selected aspects of these summaries as static PNG plots using Matplotlib:

- bar charts for handshake success, ALPN distributions, and dominant error codes from the recorder;
- bar charts for the most frequent QLOG event names and frame types, and per-direction packet-type distributions;
- histograms for selected numeric transport parameters (e.g., `max_idle_timeout`, `initial_max_data`) with weights derived from connection counts;
- bar charts for the most common DNS and connect error messages; and
- a bar chart showing the overlap and disjoint portions of group IDs between recorder and QLOG.

All plots are written to the analysis output directory. The toolbox thus closes the loop between raw measurement artefacts and human-readable insights, while remaining decoupled from the implementation details of the prober itself.

Chapter 6

Evaluation

This chapter evaluates the proposed Internet measurement framework based on two large-scale scans of the same domain set conducted from different vantage points. The first scan was conducted on privately operated servers in Switzerland, while the subsequent scan was executed on an Amazon Web Services (AWS) Elastic Compute Cloud (EC2) instance in the US-East region. Both scans used identical probe configuration and the same list of approximately 6 M domains. The goal of this evaluation is twofold: (i) to assess the robustness and scalability of the framework, and (ii) to study whether the vantage point introduces systematic bias, for example via middleboxes, path-specific impairments, or geolocation-based restrictions.

6.1 Experiment setup

This evaluation is based on two large-scale QUIC measurement runs carried out from two vantage points: privately operated servers in Switzerland and an AWS EC2 instance in the US-East region. Both scans used the same domain input, probing configuration, software stack, and measurement pipeline to ensure comparability across vantage points.

6.1.1 Domain Set Construction

The initial target list was derived from the Tranco ranking, using list identifier *8LKJV*[1]. The list was processed by the Domain Extractor module to filter entries as described in section 5.2. The extraction and filtering step produced the metrics depicted in table 6.1.

A total of 6.24 M domains were retained for probing and used in both vantage-point measurements.

[1] `https://tranco-list.eu/list/8LKJV/full`

Table 6.1: Overview of Extraction Metrics

Metric	Value
Extracted	6 436 542
Kept	6 242 562
Filtered	193 980
Time (s)	214.36
Extracted per second	30 026.787
Kept per second	29 121.86

6.1.2 Probing Configuration

All scans were executed using the same configuration file, which controlled concurrency, rate limiting, logging behavior, and the set of connection configurations. The scheduler was configured to allow up to *500 concurrent probes*, a sustained request rate of *150 requests/s*, and a burst size of *150*, with a minimum inter-attempt delay of *3 seconds* for the same domain. Logging of recorder files and QLOG traces was enabled for all connections.

The configuration defines three QUIC connection profiles corresponding to representative parameter sets for TQUIC, Firefox, and Chrome. These profiles differ in flow-control limits, initial per-stream budgets, maximum UDP payload sizes, and active connection ID limits. All configurations set `enable_multipath=true`, using the redundant scheduling algorithm. The full configuration file used in all scans is reproduced in listing A.1.

6.1.3 Deployment Environment

QUIC Lab was containerized to guarantee reproducible deployments across all vantage points. The scanner and the nginx-based opt-out page were bundled into a single Docker image. Deployment required only the minimal Compose specification shown in listing 6.1.

```yaml
services:
  quic-lab:
    container_name: quic-lab
    image: ghcr.io/quic-lab/quic-lab
    ports:
      - "80:80"
    dns:
      - "1.1.1.1"
      - "2606:4700:4700::1111"
      - "8.8.8.8"
      - "2001:4860:4860::8888"
    volumes:
      - ./in:/app/in
      - ./out:/app/out
```

Listing 6.1: Docker Compose Definition for QUIC Lab

This setup enabled zero-friction redeployment on both the private servers and the AWS instance and ensured strict environmental parity across vantage points.

DNS configuration

To minimise DNS-induced bias between vantage points, Cloudflare DNS was used as the primary resolver and Google DNS as the secondary resolver. This ensured consistent DNS resolution behavior and avoided artefacts caused by resolver-specific caching, filtering, or regional variability.

6.1.4 Opt-Out Infrastructure

To comply with ethical research guidelines, an opt-out mechanism was deployed. A lightweight nginx server was bundled into the same Docker container to serve a static information page describing the purpose of the measurements, the collected metadata, and the opt-out process as depicted in fig. 6.1. Furthermore, the `User-Agent` header was set to `"QUIC Lab (research; no-harm-intended; opt-out: opt-out@quiclab.anonaddy.com)"` to signal the research nature of the request. Opt-out requests submitted to the dedicated opt-out address were processed promptly, and affected domains were added to a blacklist.

6.1.5 Multipath QUIC Considerations

The framework uses the Tencent TQUIC implementation as its QUIC library. At the time of the experiment, TQUIC supported only draft-ietf-quic-multipath-05[2], an early and now outdated version of the multipath extension. No actively maintained QUIC implementation supporting more recent multipath drafts was available. Consequently, multipath detection in this study is limited to the presence or absence of the transport parameters defined in draft-05. The framework is modular and can incorporate updated logic once newer versions are implemented in TQUIC or other QUIC stacks.

6.2 Measurement Artefacts and Metrics

The framework produces three complementary artefact types per scan:

1. **Recorder data:** The recorder component stores one JSON record for each attempted connection. This record summarizes the result of the handshake, the negotiated Application-Layer Protocol Negotiation (ALPN) identifier, the presence of multipath-related transport parameters, local and peer-side error codes, and statistics such as number of bytes and packages sent, received, and lost.

[2] `https://github.com/Tencent/tquic/discussions/379#discussioncomment-10470855`

About These QUIC/HTTP/3 Scans

This host is running academic measurements to understand support and behavior of modern Internet protocols (e.g., QUIC and HTTP/3). We perform **lightweight active probes** to check feature availability, interoperability, and reliability across public infrastructure.

What we do

- Establish short, low-rate connections to public endpoints.
- Record minimal technical metadata needed for research.
- Follow conservative rate limits and avoid service disruption.

Data we may store

- Target domain and IP, connection timing, negotiated protocol versions/ALPN.
- Non-payload protocol metadata (no application content).
- Timestamp and basic scan outcome (success, error code).

Opt out

You can opt out any time. Send an email to opt-out@quiclab.anonaddy.com with the **domain(s)** you want excluded. Example:

```
Subject: Opt-out request
Body: Please exclude the following domains from your scans:
```

- `example.org`
- `sub.example.org`

We will add your domain(s) to our blocklist promptly.

Opt-out requests will be accepted until **11 November 2025**, the planned end of this study. After this date, no further scans will be conducted.

Questions

If you believe your service was misidentified or affected, contact opt-out@quiclab.anonaddy.com.

Research purpose only. No commercial use. We strive to be good Internet citizens.

Figure 6.1: Opt-Out Page

Table 6.2: Global Scan Statistics

	Private	**AWS**
Total Records	5 483 325	5 482 651
Unique Group IDs	5 483 325	5 482 651
QLOG Group IDs	5 491 470	5 489 530
Total Events	241 786 486	236 636 215
Events per Connection	44.0295	43.1068
Handshake OK	1 559 779 (28.44%)	1 728 807 (31.54%)
DNS Errors	756 521	757 633
Connect Errors	8199	6879

2. **QLOG traces:** For each connection, a structured QUIC trace is recorded in QLOG format, containing events such as packet transmission and reception, frame types, transport parameter settings, and loss events.

3. **Application logs:** The Rust implementation writes diagnostic log lines, which are used to parse high-level error categories at a later point: Domain Name System (DNS) lookup failures, connection establishment errors, and other implementation-level errors.

The QUIC Lab Analyzer merges these artefacts into machine-readable summaries: `recorder_summary.json`, `qlog_summary.json`, and `logs_summary.json`, complemented by CSV exports and plots. All numbers reported in this section are derived from these aggregated summaries. An overview of both scans is given in table 6.2.

6.3 Coverage and Handshake Outcomes

For the private Swiss vantage point, the recorder processed 5 483 325 records, corresponding to 5 483 325 unique group identifiers (one group identifier per probed target). From the AWS vantage, 5 482 651 records and group identifiers were observed. The small difference in total records is due to implementation-level retries and minor discrepancies in the execution runs and is negligible compared to the overall dataset size.

The first key metric is the proportion of successful QUIC handshakes. The recorder's `handshake_ok` flag was set to `true` for 1 559 779 connections on the private vantage and for 1 728 807 connections on the AWS vantage, as visualized in fig. 6.2. This corresponds to handshake success rates of 28.4% (private) and 31.5% (AWS). The difference of approximately 3 percentage points indicates that more targets are reachable from the AWS vantage, which suggests some level of path- or location-dependent reachability variation.

The ALPN distribution is tightly coupled to the handshake outcome. For the private vantage, 1 559 870 connections negotiated the HTTP/3 ALPN identifier `h3`, while 3 923 455 connections either did not complete the handshake or did not negotiate any ALPN (recorded as `<none>`). For the AWS vantage, 1 728 906 connections negotiated `h3` and 3 753 745 were recorded as `<none>`. Table 6.3 summarizes the ASPN distribution. The

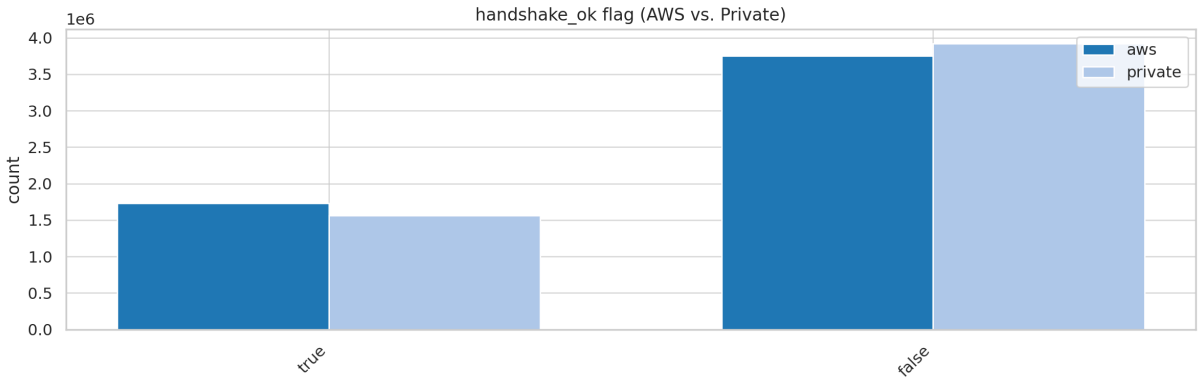

Figure 6.2: `handshake_ok` Flag set in Recorder (AWS and Private combined)

Table 6.3: ALPN distribution

	Private	**AWS**
h3	1 559 870	1 728 906
<none>	3 923 455	3 753 745

close numerical match between successful handshakes and `h3` ALPN counts in both scans shows that the framework predominantly observes QUIC endpoints that directly support Hypertext Transfer Protocol Version 3 (HTTP/3) and that nearly all successful QUIC handshakes lead to HTTP/3-capable connections.

A central objective of the thesis is to assess the deployment status of Multipath QUIC. For this purpose, the recorder tracks the `enable_multipath` transport parameter advertised by the remote peer. In both scans, `enable_multipath` is recorded as `false` for all 5.48 M records, and no single connection advertises multipath support. Consistently, the QLOG traces do not contain any `quic:path_*` events; the corresponding path-event counters remain zero for both vantage points. Within the limits of this dataset, no Internet-wide deployment of Multipath QUIC was observed in the remote transport parameters.

The internal consistency of the artefacts is confirmed by the cross-summary of recorder and QLOG group identifiers. On AWS, all 5 482 651 recorder group identifiers are also present in the QLOG set, and only 0.13% of QLOG group identifiers lack a corresponding recorder entry. On the private vantage, 99.999% of recorder identifiers are matched in QLOG, with 0.15% of QLOG identifiers having no corresponding recorder entry. This demonstrates that the instrumentation reliably produces coherent recorder and QLOG data at scale.

6.4 QLOG Event and Frame Distributions

The QLOG summaries quantify both the load on the logging subsystem and the structure of observed QUIC traffic. The AWS scan produced 236.6 M valid QLOG events across 5 489 530 unique connections, corresponding to an average of approximately 43.1 events per connection. The private scan produced 241.8 M events across 5 491 470 unique

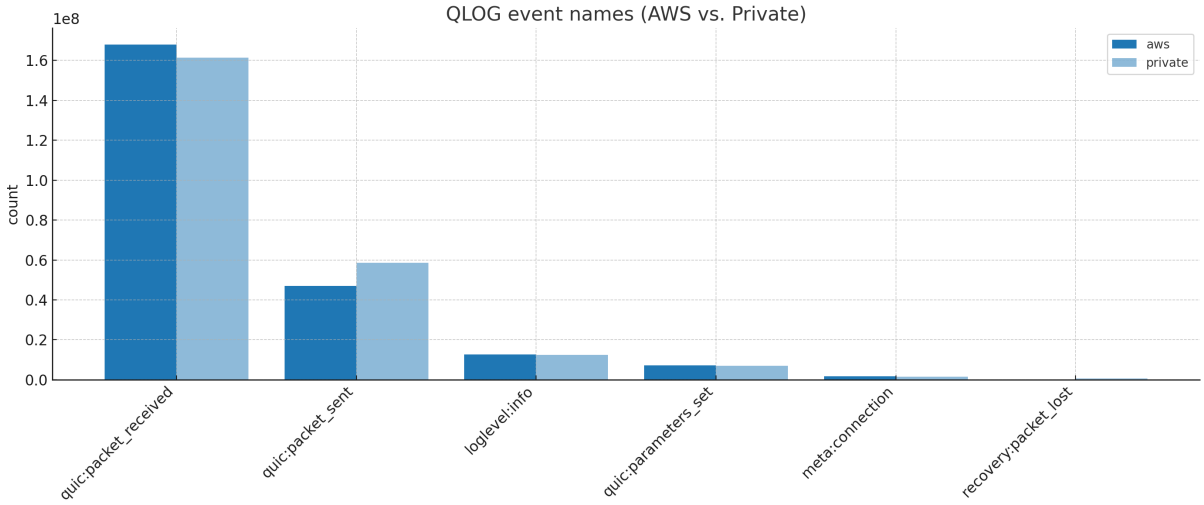

Figure 6.3: QLOG Event Names (AWS and Private combined)

connections, i.e., approximately 44.0 events per connection. The difference is modest and mainly results from a higher number of retransmission and loss-related events on the private vantage.

In both scans, the QLOG event space is dominated by `quic:packet_received` and `quic:packet_sent` events. On AWS, 167.9 M packet-receive events and 46.99 M packet-send events were recorded. On the private vantage, 161.5 M packet-receive events and 58.47 M packet-send events were observed. The absolute counts differ slightly, but the qualitative picture is similar: most QLOG entries correspond to individual QUIC packets, which underlines that the logging volume scales linearly with the number of packets rather than with the number of connections. Figure 6.3 visualizes the amount of events for every QLOG event and compares it for both scans.

At the frame level, both vantage points exhibit very similar distributions. The most frequent frame type is the STREAM frame, with 164.2 M occurrences (AWS) and 158.0 M occurrences (private), followed by CRYPTO frames (30.3 M vs. 30.5 M) and ACK frames (27.2 M vs. 38.0 M). PADDING, NEW_CONNECTION_ID, CONNECTION_CLOSE, HANDSHAKE_DONE, and PING frames also appear frequently in both scans, with counts in the low single-digit million range, as shown in fig. 6.4. These distributions are consistent with typical HTTP/3 workloads over QUIC: application data is primarily sent on a small number of bidirectional streams, with frequent acknowledgements and cryptographic handshake traffic.

A notable difference between vantage points appears in the number of `recovery:packet_lost` events. As shown in table 6.4, the AWS scan records 58 127 such events, whereas the private scan records 702 780 events. While these counters are implementation-specific, the order-of-magnitude difference indicates that the private vantage experiences significantly more packet loss or loss-like conditions (e.g., reordering above the loss threshold) than the AWS vantage. This is consistent with the expectation that a residential or non-datacenter link is more likely to exhibit congestion and jitter than a well-provisioned cloud network path.

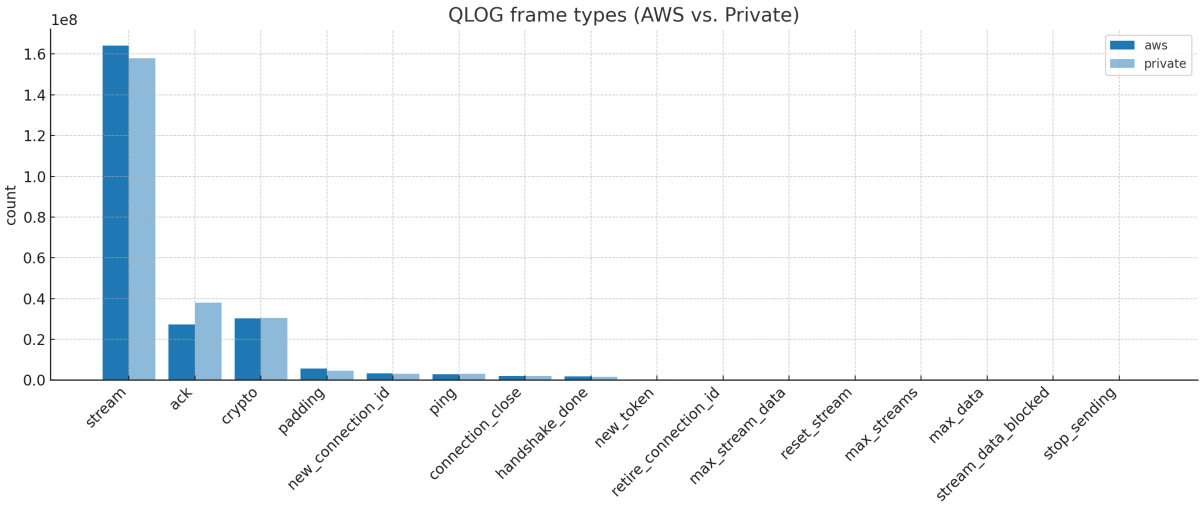

Figure 6.4: QLOG Frame Types (AWS and Private combined)

Table 6.4: Loss-related Event Counts

	Private	**AWS**
quic:packet_received	161 463 677	167 943 612
quic:packet_sent	58 474 765	46 986 403
loglevel:info	12 533 945	12 700 612
quic:parameters_set	7 051 330	7 218 435
meta:connection	1 559 766	1 728 807
recovery:packet_lost	702 780	58 127

6.5 Transport Parameter Distributions

The QLOG summaries further expose the distribution of remote QUIC transport parameters, which reflect server-side configuration. For each successful connection, the framework records the set of transport parameters observed in the server's `quic:parameters_set` events. Across both vantage points, 15 parameters are consistently present, including:

- `max_idle_timeout`
- `initial_max_data`
- `initial_max_stream_data_*`
- `initial_max_streams_*`
- `max_udp_payload_size`
- `ack_delay_exponent`
- `max_ack_delay`
- `active_connection_id_limit`
- `disable_active_migration`
- the negotiated Transport Layer Security (TLS) cipher suite.

The distributions of these parameters are remarkably stable across vantage points, which is expected, as they are properties of the remote endpoints rather than of the path. Only small variations appear due to the slightly different sets of endpoints that completed a handshake from each vantage point. In fact, the AWS scan completes approximately 10.8% more successful QUIC handshakes than the private vantage point (1 728 807 vs. 1 559 779), which directly translates into a proportional increase in the number of transport-parameter blocks observed. Consequently, the absolute counts in the AWS histograms are consistently higher by this factor, while the relative shapes of the distributions remain effectively identical. Figure 6.5 demonstrates this explicitly for the parameter `max_idle_timeout` and is representative of the behavior observed for all other transport-parameter distributions across vantage points.

The following trends are observed:

- **ACK delay exponent:** In both scans, the ACK delay exponent equals 3 for approximately 96.3–96.4% of connections, with small minorities using 8 or 10. This aligns with the default value suggested in the QUIC specification.
- **Active connection ID limit:** Approximately 83.3–83.4% of connections advertise an active connection identifier limit of 2. Values of 4, 8, or 3 are used by smaller fractions of endpoints. This suggests that the majority of deployments do not aggressively exploit connection migration or connection ID rotation.

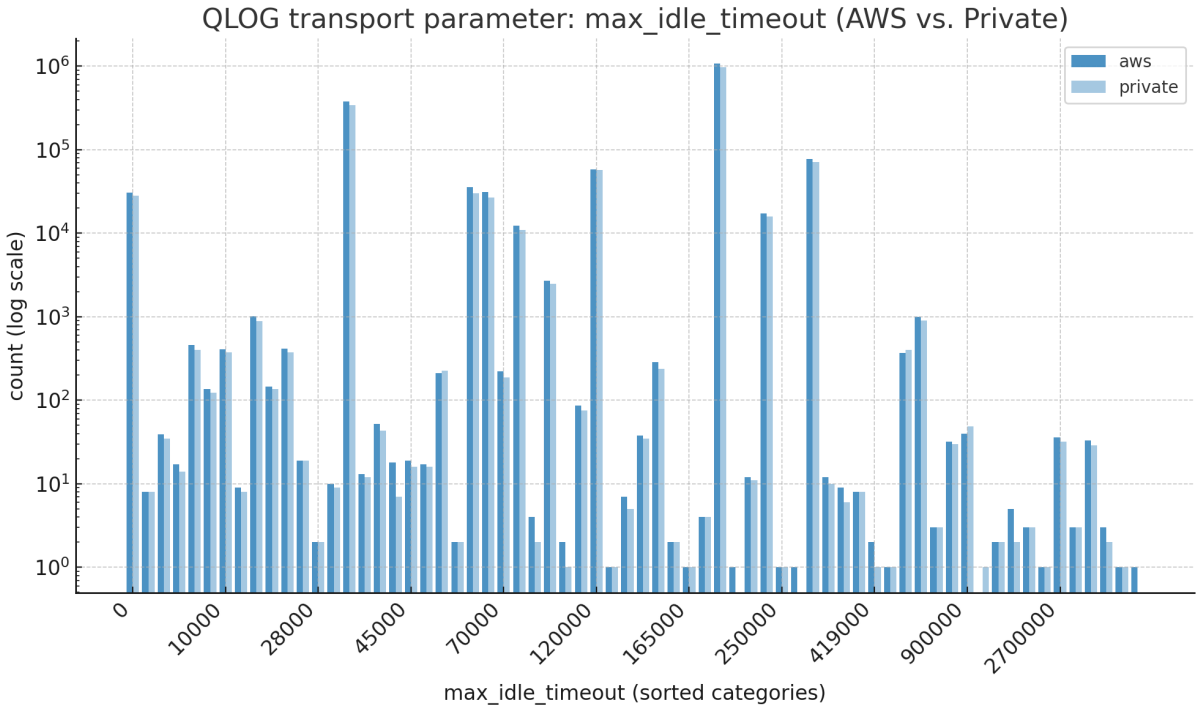

Figure 6.5: Distribution of `max_idle_timeout` transport parameter values observed (AWS and Private)

- **Disable active migration:** Around 74.7–74.8% of connections set `disable_active_migration=true`, indicating that active address migration is not permitted. Only about one quarter of endpoints allow active migration, which is consistent with the current deployment status where server operators tend to prefer stable paths over client-driven migration.

- **Flow control parameters:** For `initial_max_data`, roughly three quarters of endpoints cluster around large window sizes in the range of 10 MB to 16 MiB, with a prominent mode at 10 000 000 bytes (about 50% of AWS-side connections) and 10 485 760 bytes (about 47% on the private side). The `initial_max_stream_data_*` parameters show similar patterns, with many endpoints granting approximately 1 MB per stream and a second cluster at 1 MiB. This indicates that most deployments provision generous flow-control limits suitable for high-throughput HTTP/3 traffic.

- **Stream concurrency limits:** About 87% of connections advertise an `initial_max_streams_bidi` value of 100, and about 88% advertise an `initial_max_streams_uni` value of 3. A second cluster around 103 unidirectional streams is visible (6–7% of connections). This pattern is consistent across both vantage points and reflects common HTTP/3 stack defaults.

- **Idle timeout and maximum UDP payload size:** The most common `max_idle_timeout` value is 180 000 ms (180 s), used by about 62–63% of endpoints, followed by 30 000 ms (30 s) for roughly 22% of endpoints and 300,000 ms (300 s) for about 4–5%. For `max_udp_payload_size`, two configurations dominate: either

a conservative value around 1350 bytes (50% of endpoints on AWS, 27% on the private vantage) or the maximum of 65 527 bytes (36.6% on AWS, 59.6% on the private vantage). These values correspond to either path-MTU-conservative endpoints or endpoints that accept almost full-size User Datagram Protocol (UDP) datagrams and rely on path MTU discovery or fragmentation handling.

- **TLS ciphers:** Finally, the negotiated TLS cipher suites show a very clear picture. Approximately 93% of connections use AES-128-Galois/Counter Mode (AES-128-GCM), around 6.5% use AES-256-GCM, and less than 1% use ChaCha20-Poly1305. The distribution is nearly identical on both vantage points and reflects the dominant usage of AES-based AEAD ciphers in modern QUIC deployments.

Overall, the transport parameter distributions are highly consistent across vantage points. This suggests that the vantage point does not introduce observable bias in the characterization of server-side QUIC configurations.

6.6 Error Analysis

The application logs provide insight into failure modes during the scanning process. Errors are classified into DNS lookup failures, connection establishment errors, and a residual category of other errors.

DNS lookup failures are significant but nearly identical across vantage points. The AWS scan recorded 757 633 DNS lookup errors, whereas the private scan recorded 756 521. These counts correspond to roughly 0.138 DNS errors per probed target in both runs, i.e., approximately 14% of domains cannot be resolved successfully. The three dominant error messages are consistent across vantage points: “No address associated with hostname” (about 383k occurrences), “Name or service not known” (about 209k), and “Temporary failure in name resolution” (about 164k). This indicates that the DNS resolution outcome is essentially independent of the vantage point for this dataset.

Connection establishment errors are rare by comparison. On AWS, 6 879 connect errors were recorded, corresponding to about 0.00125 errors per target. On the private vantage, 8 199 connect errors (0.00150 per target) were observed. The primary connect error on AWS is “Network is unreachable,” which suggests transient routing or local socket issues within the cloud environment. On the private vantage, the most frequent connect errors are “Cannot assign requested address” and “Invalid argument,” which are symptomatic of local socket configuration or ephemeral port/address exhaustion. In both cases, connect errors are two orders of magnitude less frequent than DNS failures and do not materially affect the aggregate statistics.

The residual category `other_error` aggregates all log lines containing “ERROR” that do not match the DNS or connect patterns. It accounts for roughly 1.24 M events on AWS and 1.11 M events on the private vantage. These errors largely correspond to higher-level handshake failures or protocol-level issues and are better understood via the recorder’s peer-close error codes.

On the recorder side, local closes are overwhelmingly reported with error code 0 (NO_ERROR), which indicates that the measurement framework itself terminates connections cleanly after completing the probing logic. For example, on the private vantage, 1 547 661 of 1 547 687 local closes carry error code 0. Peer-side closes exhibit a rich distribution of error codes, with two dominant codes around 296 and 336 across both vantage points (roughly 250k and 75–80k occurrences, respectively), and smaller counts for core QUIC error codes such as 0 (NO_ERROR), 1 (INTERNAL_ERROR), 11 (PROTOCOL_VIOLATION), and 12 (INVALID_TOKEN). The near-identical distribution of peer-close error codes between vantage points indicates that endpoints react in the same way, regardless of whether the client is located in Switzerland or on AWS.

6.7 Impact of Vantage Point and Limitations

The comparison of both scans allows the following conclusions about the impact of the vantage point:

- The proportion of domains that can be resolved via DNS is practically identical for both vantage points. Vantage-point-dependent differences in authoritative DNS responses or resolver behavior, if present at all, are smaller than the noise in this dataset.
- The distribution of server-side QUIC transport parameters is extremely stable across vantage points. No systematic shift in idle timeouts, flow-control windows, stream limits, migration flags, or cipher suites is observed between the private server and the AWS instance.
- The main vantage-point-dependent differences arise in path quality metrics (as inferred from `recovery:packet_lost` events) and in handshake success rates. The private vantage experiences an order of magnitude more loss events and a slightly lower handshake success rate, which is consistent with more congested or heterogeneous paths compared to the datacenter environment.
- No evidence of vantage-point-specific blocking or protocol downgrades was observed at the aggregate level. In particular, there is no indication that HTTP/3 or QUIC are selectively blocked or degraded for one vantage point while being allowed for the other on a significant fraction of endpoints.

Two limitations of the present evaluation must be noted. First, the current QLOG traces in this dataset do not expose per-path events, which prevents a detailed analysis of multipath behavior even if it existed. Second, the analyzer was unable to extract per-packet size information reliably from the QLOG traces (all byte counters remain zero), which precludes throughput and volume-based comparisons. These limitations are due to the structure of the TQUIC QLOG output and can be addressed by future extensions of the logging and analysis, or directly by the authors of the TQUIC library.

In summary, the evaluation demonstrates that the proposed framework scales to millions of targets and produces internally consistent recorder, QLOG, and log artefacts. For the considered dataset, the vantage point has negligible impact on the characterization of QUIC and HTTP/3 deployment (ALPN negotiation, transport parameter distributions, cipher usage) but does affect loss-related metrics and, to a lesser extent, handshake success rates. Within these constraints, the measurements provide a robust view of current QUIC deployments and confirm the absence of observable Multipath QUIC support in the examined population.

6.8 Performance

This subsection evaluates the performance characteristics of the measurement framework during the large-scale scans. Both scans processed the same domain set and completed in approximately 58 hours, resulting in a sustained throughput of roughly 26 domains per second. The performance analysis focuses on CPU utilization, memory consumption, and network utilization on the private server in Switzerland and the AWS EC2 instance in the US-East region.

6.8.1 Computational Resource Usage

Private server

Resource monitoring is available for the initial 12 hours of the scan. During this period, CPU utilization fluctuates between approximately 10% and 25%, with an average around 17%. Memory usage remains low (typically below 1 GB) and does not exhibit any trend that would indicate growth, leak, or pressure. Figure 6.6 provides an overview of the CPU, memory, and network utilization of the private server during the first 12 hours of the scan. According to system-level telemetry and manual observation, this utilization pattern remained stable for the remainder of the 58-hour run, with no signs of saturation or performance degradation. This behavior indicates that the client-side processing pipeline (DNS resolution, QUIC handshakes, logging, and local preprocessing) is lightweight relative to the available hardware and that the scanning rate is not constrained by compute resources on the private vantage point.

AWS EC2 instance

Full 58-hour monitoring is available for the AWS instance. The CPU utilization profile closely mirrors the private server: values range between 18% and 24%, with a stable mean around 21%. Memory usage remains below 1 GB throughout the entire measurement period. This confirms that the framework's workload is well within the instance's capacity, even for long-running experiments. Figure 6.7 shows the corresponding resource utilization on the AWS EC2 instance, covering the full fifty-eight-hour duration of the scan.

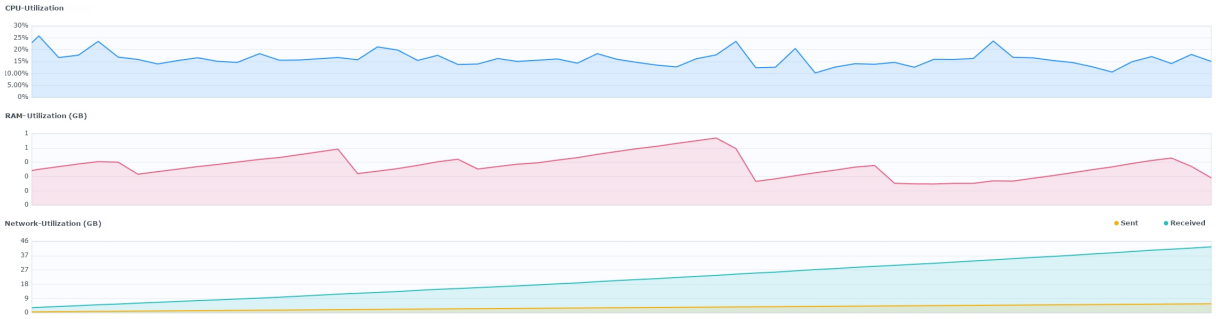

Figure 6.6: Resource Consumption of Private Server over the Period of 12 Hours

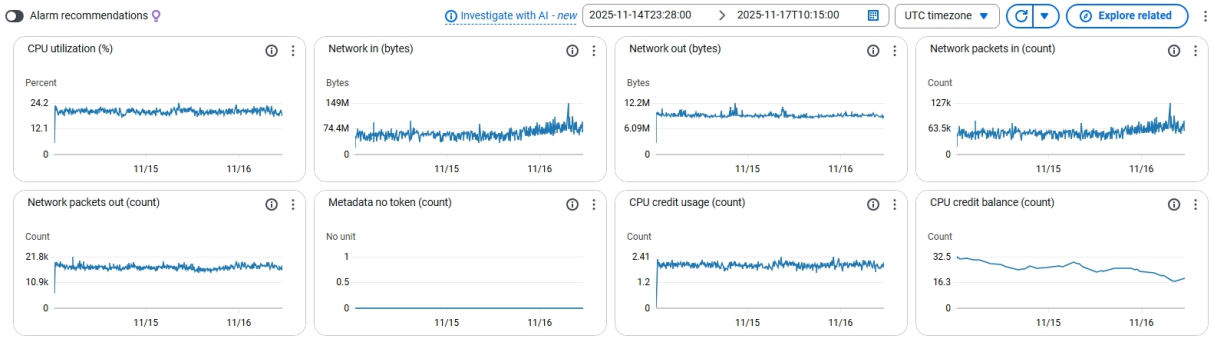

Figure 6.7: Resource Consumption of AWS EC2 over the entire Scan Period

Overall, both vantage points show almost identical computational behavior, with low and stable utilization, and no indication of bottlenecks caused by the measurement framework itself.

6.8.2 Network Utilization

Both vantage points exhibit consistent and gradually increasing inbound traffic as more QUIC endpoints respond over time. Outbound packet rates remain relatively stable, reflecting the constant probing rate enforced by the scheduler.

- On the **private server**, inbound network throughput grows steadily during the first 12 hours. This increase corresponds to the natural progression of the scan, where later stages of the domain list contain more active HTTP/3 deployments. Subsequent monitoring showed that this trend continued for the rest of the experiment.
- On **AWS**, where complete data is available, the same overall pattern appears: stable outbound traffic and a slow upward trend in inbound traffic, interrupted only by occasional short peaks due to bursty responses from certain clusters of endpoints.

At both vantage points, the network interface remains well below saturation, confirming that throughput is dominated by remote server behavior, path latency, and protocol-level timing—rather than by local link capacity or congestion.

6.8.3 Throughput and Runtime Characteristics

Both scans completed in ≈ 58 hours, with a sustained rate of:

$$\frac{5.48 \times 10^6 \,\text{domains}}{58 \,\text{hours}} \approx 26.2 \,\text{domains/s}$$

Neither vantage point exhibits slowdowns, backlog accumulation, or cyclic performance variation. This demonstrates that:

1. concurrency limits and rate-control mechanisms operate as intended,
2. long-running measurements do not accumulate overhead,
3. QLOG writing and recorder logging scale linearly without creating I/O pressure.

6.8.4 Summary

The performance evaluation shows that:

- CPU utilization remains low (<25%) and stable on both vantage points,
- Memory consumption is very small (<1 GB) and does not grow over time,
- Network utilization scales smoothly without interface saturation,
- Resource usage patterns are nearly identical on both machines,
- The difference in QUIC-level behavioral metrics (e.g., packet loss, handshake success rate) stems from **path properties**, not from local performance limitations.

Although only the first 12 hours of private-server utilization were recorded, the observed stability and absence of variance strongly indicate that the full run behaved consistently. The AWS data, covering the entire scan duration, supports this conclusion by showing nearly identical utilization patterns over the full 58-hour period.

Chapter 7

Summary and Conclusions

This chapter summaries the work conducted in this thesis, distills the main scientific conclusions, and outlines promising directions for future work.

7.1 Summary

The goal of this thesis was to design, implement, and evaluate a scalable framework for Internet-wide measurements of modern transport- and application-layer protocols, with a particular focus on QUIC, and emerging extensions such as Multipath QUIC. Rather than concentrating on a single protocol mechanism, the work set out to provide a reusable measurement toolkit that can be extended with new probes and analysis modules as the protocol ecosystem evolves.

To this end, the thesis first established the necessary background on Internet measurement, TLS 1.3, QUIC, and Multipath QUIC in chapter 2, and positioned the work in the context of existing active and passive measurement studies, as well as prior work on multipath transport and structured protocol logging in chapter 3. This survey motivated the need for a measurement framework that (i) operates at Internet scale, (ii) targets protocol features beyond simple reachability, and (iii) builds on structured logging formats such as qlog to preserve rich protocol semantics.

On this basis, chapter 4 introduced a modular architecture comprising three main subsystems: the *Domain Extractor*, the *QUIC Lab*, and the *QUIC Lab Analyzer*. The Domain Extractor ingests one or more input lists (such as Tranco lists), applies configurable filtering and normalization rules, and produces a canonicalized target set together with reproducible extraction artifacts. QUIC Lab is a general probing engine built around a small, stable core that handles configuration, scheduling, concurrency control, transport abstractions, logging, and recording, while delegating protocol-specific logic to pluggable probes. The QUIC Lab Analyzer ingests recorder outputs, qlog files, and application logs, and turns them into aggregated statistics and visualizations that directly address the research objectives. A central design principle throughout the architecture is a strict

separation of concerns: probes implement only the protocol logic, while orchestration, rate limiting, logging, and persistence are handled by the core framework.

Chapter 5 described a concrete realization of this architecture in Rust. The implementation integrates Tencent's TQUIC library as the QUIC transport, including its qlog support, and provides an HTTP/3 probe that issues minimal `GET` requests over QUIC to each target domain. The QUIC Lab Analyzer subproject provides parsers for recorder and qlog artifacts, performs multi-process aggregation of events, frames, and transport parameters, and exports results as JSON summaries and CSV files.

The evaluation in chapter 6 demonstrated the capabilities of the framework through a large-scale Internet measurement campaign. A domain set of 6.24 M targets was constructed from the Tranco list, with 6 436 542 input entries, 6 242 562 domains retained, and 193 980 filtered during extraction. Using the same configuration, two full scans were run from independent vantage points: privately operated servers in Switzerland and an AWS EC2 instance in the *US-East* region. Each scan attempted QUIC connections to approximately 5.48 M domains and enabled both recorder and qlog logging for all attempts.

7.2 Conclusions

The empirical analysis showed that QUIC and HTTP/3 are widely but not universally deployed among the considered domains. Only 28.44% of connection attempts from the private vantage point and 31.54% from AWS resulted in successful QUIC handshakes, with the remainder failing at the DNS or connection stage. Among the successful handshakes, the ALPN distribution was dominated by HTTP/3: the counts of `h3` ALPN values closely matched the number of successful handshakes, and all other connections reported `<none>`. The qlog-based analysis of remote QUIC transport parameters revealed a high degree of convergence on a small set of operational defaults. For example, approximately 96% of connections used an ACK delay exponent of three, about 83% advertised an `active_connection_id_limit` of two, and roughly three quarters disabled active migration. Flow-control limits and stream concurrency parameters clustered around values that are generous enough for high-throughput HTTP/3 traffic, while idle timeouts and maximum UDP payload sizes concentrated around a few typical configurations.

Furthermore, the framework was specifically instrumented to detect deployment of Multipath QUIC via the `enable_multipath` transport parameter. In the collected dataset, this parameter was consistently recorded as `false` across all 5.48 M connections, and no evidence of Internet-wide Multipath QUIC deployment was found. A comparison of the two vantage points indicated that DNS resolution success, transport parameter distributions, and TLS cipher suites were effectively identical, and that the main differences arose in low-level packet metrics, such as retransmission and loss events, which were slightly more pronounced on the privately hosted server. Resource monitoring showed that CPU utilization remained below 25%, memory usage stayed well under 1 GB, and network interfaces were far from saturation on both machines, confirming that the framework itself does not constitute a bottleneck even under long-running, large-scale workloads.

Overall, the thesis delivered (i) a modular, extensible measurement framework for QUIC, (ii) a concrete implementation based on TQUIC and qlog with support for pluggable probes, and (iii) an Internet-wide measurement study that characterizes current QUIC and HTTP/3 deployment practices and provides negative evidence regarding the deployment of Multipath QUIC in the examined population.

7.3 Limitations

Several limitations of the present work should be acknowledged when interpreting the results.

First, the domain set is restricted to a single Tranco snapshot and thus to a popularity-biased subset of the Internet. While this is appropriate for characterizing deployment practices among prominent domains, it leaves out the long tail of smaller sites, regional services, and specialized infrastructures. The measured distributions of transport parameters and protocol support therefore reflect the behavior of popular domains and may not generalize to the broader Internet.

Second, the analysis is constrained by the structure and semantics of the qlog output produced by the TQUIC library. In particular, reliable per-packet size information was not available in the collected traces, which precludes precise throughput and volume-based comparisons across endpoints and vantage points. Some higher-level metrics that could in principle be derived from qlog, such as detailed congestion-window dynamics, are also not exploited in the current analysis pipeline. These limitations stem from the specific qlog implementation and can be alleviated only by changes to the logging library or by complementing qlog with additional measurement sources.

Finally, all probing is conducted using a single client implementation (TQUIC) and a single HTTP/3 probe that issues a minimal `GET` request. While this choice simplifies the analysis and ensures consistent behavior across measurements, it may not trigger all code paths in server implementations, especially for features that are only enabled under specific application-layer conditions. The results therefore characterize server behavior under a particular, carefully controlled client workload rather than under arbitrary application traffic.

7.4 Future Work

A natural extension is to develop additional probes that exercise specific QUIC and HTTP/3 mechanisms beyond basic reachability and header retrieval. Examples include probes that explicitly trigger and measure connection migration under controlled changes of the client address, experiments that evaluate 0-RTT resumption behavior over repeated connections, or probes that assess QPACK behavior and header compression efficiency. The existing probe interface and scheduler are designed to support such extensions without modifications to the core.

Second, the analysis pipeline can be enhanced to exploit a larger portion of the information contained in qlog traces. This includes reconstructing per-connection time series of congestion-window size, loss recovery episodes, and round-trip times, as well as correlating transport-parameter choices with observed performance characteristics. Automated detection of anomalous behavior, such as excessive loss, reordering, or protocol violations, could be added on top of the existing aggregation modules, potentially leveraging machine-learning techniques for clustering and anomaly detection.

Finally, the current negative results regarding Multipath QUIC motivate continued monitoring of multipath-related transport parameters and, once deployments emerge, the design of dedicated multipath probes. These probes could, for example, deliberately establish multiple paths, inject controlled path failures, or manipulate path characteristics to study how endpoints schedule traffic across paths and how robust multipath implementations are under adverse network conditions.

Bibliography

[1] J. Iyengar and M. Thomson, *QUIC: A UDP-Based Multiplexed and Secure Transport*, RFC 9000, May 2021. DOI: 10.17487/RFC9000. [Online]. Available: https://www.rfc-editor.org/info/rfc9000.

[2] M. Bishop, *HTTP/3*, RFC 9114, Jun. 2022. DOI: 10.17487/RFC9114. [Online]. Available: https://www.rfc-editor.org/info/rfc9114.

[3] A. Langley, A. Riddoch, A. Wilk, *et al.*, "The quic transport protocol: Design and internet-scale deployment", in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM 17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 183–196, ISBN: 9781450346535. DOI: 10.1145/3098822.3098842. [Online]. Available: https://doi.org/10.1145/3098822.3098842.

[4] W. Eddy, *Transmission Control Protocol (TCP)*, RFC 9293, Aug. 2022. DOI: 10.17487/RFC9293. [Online]. Available: https://www.rfc-editor.org/info/rfc9293.

[5] J. Postel, *User Datagram Protocol*, RFC 768, Aug. 1980. DOI: 10.17487/RFC0768. [Online]. Available: https://www.rfc-editor.org/info/rfc768.

[6] P. Megyesi, Z. Kraemer, and S. Molnar, "How quick is quic?", in *2016 IEEE International Conference on Communications (ICC)*, 2016, pp. 1–6. DOI: https://doi.org/10.1109/ICC.2016.7510788.

[7] K. Wolsing, J. Rüth, K. Wehrle, and O. Hohlfeld, "A performance perspective on web optimized protocol stacks: Tcp+tls+http/2 vs. quic", in *Proceedings of the 2019 Applied Networking Research Workshop*, ser. ANRW '19, Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 1–7, ISBN: 9781450368483. DOI: 10.1145/3340301.3341123. [Online]. Available: https://doi.org/10.1145/3340301.3341123.

[8] E. Pauley and P. McDaniel, "Understanding the ethical frameworks of internet measurement studies", in *Proceedings of the 2nd IEEE International Workshop on Ethics in Computer Security (ETHICS)*, vol. 10, 2023. DOI: 10.14722/ethics.2023.239547.

[9] J. Rüth, I. Poese, C. Dietzel, and O. Hohlfeld, "A first look at quic in the wild", in *Passive and Active Measurement*, R. Beverly, G. Smaragdakis, and A. Feldmann, Eds., Cham: Springer International Publishing, 2018, pp. 255–268, ISBN: 978-3-319-76481-8.

[10] A. Langley, A. Riddoch, A. Wilk, *et al.*, "The quic transport protocol: Design and internet-scale deployment", in *Proceedings of the conference of the ACM special interest group on data communication*, 2017, pp. 183–196.

[11] J. Mücke, M. Nawrocki, R. Hiesgen, *et al.*, "Waiting for quic: Passive measurements to understand quic deployments", *Proceedings of the ACM on Networking*, vol. 3, no. CoNEXT4, pp. 1–26, 2025.

[12] P. De Vaere, T. Bühler, M. Kühlewind, and B. Trammell, "Three bits suffice: Explicit support for passive measurement of internet latency in quic and tcp", in *Proceedings of the Internet Measurement Conference 2018*, ser. IMC '18, Boston, MA, USA: Association for Computing Machinery, 2018, pp. 22–28, ISBN: 9781450356190. DOI: 10.1145/3278532.3278535. [Online]. Available: https://doi-org.ezproxy.uzh.ch/10.1145/3278532.3278535.

[13] L. Ciprian, "Ethics, products, top lists-and their use at internet measurement conferences", *Network*, vol. 23, 2018. DOI: 10.2313/NET-2018-03-1_04.

[14] J. Zirngibl, P. Buschmann, P. Sattler, B. Jaeger, J. Aulbach, and G. Carle, "It's over 9000: Analyzing early quic deployments with the standardization on the horizon", in *Proceedings of the 21st ACM Internet Measurement Conference*, ser. IMC '21, Virtual Event: Association for Computing Machinery, 2021, pp. 261–275, ISBN: 9781450391290. DOI: 10.1145/3487552.3487826.

[15] V. Le Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation", in *Proceedings of the 26th Annual Network and Distributed System Security Symposium*, ser. NDSS 2019, Feb. 2019. DOI: 10.14722/ndss.2019.23386.

[16] E. Rescorla, *The Transport Layer Security (TLS) Protocol Version 1.3*, RFC 8446, Aug. 2018. DOI: 10.17487/RFC8446. [Online]. Available: https://www.rfc-editor.org/info/rfc8446.

[17] M. Thomson and S. Turner, *Using TLS to Secure QUIC*, RFC 9001, May 2021. DOI: 10.17487/RFC9001. [Online]. Available: https://www.rfc-editor.org/info/rfc9001.

[18] J. Iyengar and I. Swett, *QUIC Loss Detection and Congestion Control*, RFC 9002, May 2021. DOI: 10.17487/RFC9002. [Online]. Available: https://www.rfc-editor.org/info/rfc9002.

[19] M. Thomson, *Version-Independent Properties of QUIC*, RFC 8999, May 2021. DOI: 10.17487/RFC8999. [Online]. Available: https://www.rfc-editor.org/info/rfc8999.

[20] D. Schinazi and E. Rescorla, *Compatible Version Negotiation for QUIC*, RFC 9368, May 2023. DOI: 10.17487/RFC9368. [Online]. Available: https://www.rfc-editor.org/info/rfc9368.

[21] Y. Liu, Y. Ma, Q. D. Coninck, O. Bonaventure, C. Huitema, and M. Kühlewind, "Managing multiple paths for a QUIC connection", Internet Engineering Task Force, Internet-Draft draft-ietf-quic-multipath-17, Oct. 2025, Work in Progress, 38 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-multipath/17/.

[22] B. Trammell and M. Kühlewind, *The Wire Image of a Network Protocol*, RFC 8546, Apr. 2019. DOI: 10.17487/RFC8546. [Online]. Available: https://www.rfc-editor.org/info/rfc8546.

[23] R. Marx, L. Niccolini, M. Seemann, and L. Pardue, "qlog: Structured Logging for Network Protocols", Internet Engineering Task Force, Internet-Draft draft-ietf-quic-qlog-main-schema-13, Oct. 2025, Work in Progress, 57 pp. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-quic-qlog-main-schema/13/.

[24] T. Bray, *The JavaScript Object Notation (JSON) Data Interchange Format*, RFC 8259, Dec. 2017. DOI: 10.17487/RFC8259. [Online]. Available: https://www.rfc-editor.org/info/rfc8259.

[25] N. Williams, *JavaScript Object Notation (JSON) Text Sequences*, RFC 7464, Feb. 2015. DOI: 10.17487/RFC7464. [Online]. Available: https://www.rfc-editor.org/info/rfc7464.

[26] T. Bray, *The I-JSON Message Format*, RFC 7493, Mar. 2015. DOI: 10.17487/RFC7493. [Online]. Available: https://www.rfc-editor.org/info/rfc7493.

[27] A. Cooper, H. Tschofenig, D. B. D. Aboba, *et al.*, *Privacy Considerations for Internet Protocols*, RFC 6973, Jul. 2013. DOI: 10.17487/RFC6973. [Online]. Available: https://www.rfc-editor.org/info/rfc6973.

[28] O. Bonaventure, M. Handley, C. Raiciu, *et al.*, "An overview of multipath tcp", *; login:*, vol. 37, no. 5, pp. 17–23, 2012.

[29] T. Shreedhar, D. Zeynali, O. Gasser, N. Mohan, and J. Ott, *A longitudinal view at the adoption of multipath tcp*, 2022. arXiv: 2205.12138 [cs.NI]. [Online]. Available: https://arxiv.org/abs/2205.12138.

[30] F. Aschenbrenner, T. Shreedhar, O. Gasser, N. Mohan, and J. Ott, "From single lane to highways: Analyzing the adoption of multipath tcp in the internet", in *2021 IFIP Networking Conference (IFIP Networking)*, Jun. 2021, pp. 1–9. DOI: 10.23919/IFIPNetworking52078.2021.9472785.

[31] Q. De Coninck and O. Bonaventure, "Multipath quic: Design and evaluation", in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17, Incheon, Republic of Korea: Association for Computing Machinery, 2017, pp. 160–166, ISBN: 9781450354226. DOI: 10.1145/3143361.3143370. [Online]. Available: https://doi.org/10.1145/3143361.3143370.

[32] I. Kunze, C. Sander, and K. Wehrle, "Does it spin? on the adoption and use of quic's spin bit", in *Proceedings of the 2023 ACM on Internet Measurement Conference*, ser. IMC '23, Montreal QC, Canada: Association for Computing Machinery, 2023, pp. 554–560, ISBN: 9798400703829. DOI: 10.1145/3618257.3624844. [Online]. Available: https://doi.org/10.1145/3618257.3624844.

[33] J. Zirngibl, F. Gebauer, P. Sattler, M. Sosnowski, and G. Carle, "Quic hunter: Finding quic deployments and identifying server libraries across the internet", in *Passive and Active Measurement*, P. Richter, V. Bajpai, and E. Carisimo, Eds., Cham: Springer Nature Switzerland, 2024, pp. 273–290, ISBN: 978-3-031-56252-5.

[34] B. Jaeger, J. Zirngibl, M. Kempf, K. Ploch, and G. Carle, "Quic on the highway: Evaluating performance on high-rate links", in *2023 IFIP Networking Conference (IFIP Networking)*, Jun. 2023, pp. 1–9. DOI: 10.23919/IFIPNetworking57963.2023.10186365.

[35] A. Buchet and C. Pelsser, "An analysis of quic connection migration in the wild", *SIGCOMM Comput. Commun. Rev.*, vol. 55, no. 1, pp. 3–9, Apr. 2025, ISSN: 0146-4833. DOI: 10.1145/3727063.3727066. [Online]. Available: https://doi.org/10.1145/3727063.3727066.

[36] X. Li, "Enhancing latency reduction and reliability for internet services with quic and webrtc", Ph.D. dissertation, Aalto University, School of Electrical Engineering, Department of Information and Communications Engineering, 2024, ISBN: 978-952-64-1991-6. [Online]. Available: https://urn.fi/URN:ISBN:978-952-64-1991-6.

[37] M. Zhou, Y. Chen, S. Lin, X. Wang, B. Liu, and A. Y. Ding, "Dissecting the applicability of http/3 in content delivery networks", in *2024 IEEE 44th International Conference on Distributed Computing Systems (ICDCS)*, Jul. 2024, pp. 936–946. DOI: 10.1109/ICDCS60910.2024.00091.

[38] R. Marx, W. Lamotte, and P. Quax, "Visualizing quic and http/3 with qlog and qvis", in *Proceedings of the SIGCOMM '20 Poster and Demo Sessions*, ser. SIGCOMM '20, Virtual event: Association for Computing Machinery, 2021, pp. 42–43, ISBN: 9781450380485. DOI: 10.1145/3405837.3412356. [Online]. Available: https://doi.org/10.1145/3405837.3412356.

Abbreviations

0-RTT	Zero Round-Trip Time
ACK	Acknowledgement
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
ALPN	Application-Layer Protocol Negotiation
API	Application Programming Interface
AWS	Amazon Web Services
CID	Connection ID
CI/CD	Continuous Integration and Continuous Deployment
CPU	Central Processing Unit
CSV	Comma-Separated Values
DNS	Domain Name System
EC2	Elastic Compute Cloud
GB	Gigabyte
HTTP	Hypertext Transfer Protocol
HTTP/3	Hypertext Transfer Protocol Version 3
I/O	Input/Output
IP	Internet Protocol
JSON	JavaScript Object Notation
JSONL	JSON Lines
JSON-SEQ	JSON Text Sequence
M	Million (10^6)
MB	Megabyte
MiB	Mebibyte
MTU	Maximum Transmission Unit
QLOG	Structured Logging Format for QUIC and HTTP/3 (qlog)
QPACK	HTTP/3 Header Compression Scheme
QUIC	Quick UDP Internet Connections
RTT	Round-Trip Time
SLD	Second-Level Domain
SQL	Structured Query Language
TCP	Transmission Control Protocol
TLD	Top-Level Domain
TLS	Transport Layer Security
TQUIC	Tencent QUIC Implementation
UDP	User Datagram Protocol

URL	Uniform Resource Locator
VM	Virtual Machine

List of Figures

List of Tables

List of Listings

Appendix A

Docker Compose File

```toml
[scheduler]
# 0 = number of CPUs
concurrency = 500
requests_per_second = 150
burst = 150
# Wait this long between attempts to the same domain (ms)
inter_attempt_delay_ms = 3000

[io]
in_dir = "in"
domains_file_name = "tranco_PLXVJ_filtered.txt"
out_dir = "out"

[general]
log_level = "INFO"  # OFF/ERROR/WARN/INFO/DEBUG/TRACE
save_log_files = true
save_recorder_files = true
save_qlog_files = true
save_keylog_files = false
# Caution: Creates one .session file for every connection
save_session_files = false


# connection_configs are tried in order until first success. You can add many.

# Default TQUIC
[[connection_config]]
port = 443
path = "/"
user_agent = "QUIC Lab (research; no-harm-intended; opt-out: opt-out@quiclab.anonaddy.com)"
verify_peer = false
alpn = ["h3"]
ip_version = "auto"  # "auto" | "v4" | "v6"
```

```
34
35 max_idle_timeout_ms = 30000
36 initial_max_data = 10485760
37 initial_max_stream_data_bidi_local = 5242880
38 initial_max_stream_data_bidi_remote = 2097152
39 initial_max_stream_data_uni = 1048576
40 initial_max_streams_bidi = 200
41 initial_max_streams_uni = 100
42 max_ack_delay = 25
43 active_connection_id_limit = 2
44 send_udp_payload_size = 1200
45 max_receive_buffer_size = 65536
46
47 enable_multipath = true
48 multipath_algorithm = "redundant"
49
50
51 # Firefox
52 [[connection_config]]
53 port = 443
54 path = "/"
55 user_agent = "QUIC Lab (research; no-harm-intended; opt-out:
       opt-out@quiclab.anonaddy.com)"
56 verify_peer = false
57 alpn = ["h3"]
58 ip_version = "auto"  # "auto" | "v4" | "v6"
59
60 max_idle_timeout_ms = 30000
61 initial_max_data = 25165824
62 initial_max_stream_data_bidi_local = 12582912
63 initial_max_stream_data_bidi_remote = 1048576
64 initial_max_stream_data_uni = 1048576
65 initial_max_streams_bidi = 16
66 initial_max_streams_uni = 16
67 max_ack_delay = 20
68 active_connection_id_limit = 8
69 send_udp_payload_size = 1200
70 max_receive_buffer_size = 65536
71
72 enable_multipath = true
73 multipath_algorithm = "redundant"
74
75
76 # Chrome
77 [[connection_config]]
78 port = 443
79 path = "/"
80 user_agent = "QUIC Lab (research; no-harm-intended; opt-out:
       opt-out@quiclab.anonaddy.com)"
81 verify_peer = false
```

```
alpn = ["h3"]
ip_version = "auto"  # "auto" | "v4" | "v6"

max_idle_timeout_ms = 30000
initial_max_data = 15728640
initial_max_stream_data_bidi_local = 6291456
initial_max_stream_data_bidi_remote = 6291456
initial_max_stream_data_uni = 6291456
initial_max_streams_bidi = 100
initial_max_streams_uni = 103
max_ack_delay = 20
active_connection_id_limit = 2
send_udp_payload_size = 1472
max_receive_buffer_size = 65536

enable_multipath = true
multipath_algorithm = "redundant"
```

Listing A.1: Configuration File used for all Scans