VRIJE UNIVERSITEIT AMSTERDAM

UNIVERSITEIT VAN AMSTERDAM

Master's Thesis
(Course Code: XM_0011)

# Optimising Web Content Delivery with Partially Reliable Transport and QUIC

by

Aditya Narayaen Srivastava
(Student Number: 2775769)

*Submitted in partial fulfillment of the requirements*
*for the joint UvA-VU degree of*
*Master of Science*
*in*
*Computer Science*
*at the*
*Vrije Universiteit Amsterdam*

August 28, 2024

Certified by .....
Balakrishnan Chandrasekaran
Assistant Professor, VU, Amsterdam
*First Supervisor*

Certified by .....
Qi Guo
Ph.D. Candidate, MPI-INF/Universität des Saarlandes, Saarbrücken, DE
*Daily Supervisor*

Certified by .....
Marc X. Makkes
Assistant Professor, VU, Amsterdam
*Second Reader*

# Optimising Web Content Delivery with Partially Reliable Transport and QUIC

Aditya Narayaen Srivastava
Vrije Universiteit Amsterdam
Amsterdam, NL
a.n.srivastava@student.vu.nl

## Abstract

The rapid evolution of Internet technologies demands innovative methods to optimize web performance. Despite recent progress in multi-path and client-side strategies, the potential of emerging transport protocols, specifically Quick UDP Internet Connections (QUIC), for web content optimization remains underexplored. This thesis aims to study and examine the effectiveness of using partial reliability with QUIC to enhance web performance by prioritizing the delivery of web page assets based on it's indispensability. Analysis of the top 100K domains shows that only 27% currently support QUIC, indicating significant potential for broader adoption. We hypothesize that not all elements of a web page elements are equally vital to user experience and suggest using partial reliability for them, informed by application layer insights, to optimize content delivery under adverse network conditions. We look at categorizing these web page objects into 'Essential' and 'Non-Essential' elements, assessing the impact of these classifications on overall page load times and the page's functionality. A server-client environment using QUIC streams and datagrams is implemented to simulate real-world conditions and evaluate partial reliability's impact. Our findings, based on thorough network analysis and real user surveys, suggest that categorizing certain web objects as 'non-essential' might be feasible, and partial reliability with QUIC can modestly reduce delivery times but only under challenging network conditions. We also find that this approach adds different levels of setup complexity and may introduce inconsistencies in content delivery. Overall, while the Partially Reliable Transport strategy shows promise, practical challenges—such as diverse file structures and size, varying quality levels, and fluctuating network conditions—pose significant implementation obstacles. This research contributes to the development of next-generation web protocols by proposing an in-network strategy that leverages QUIC's partial reliability features to address protocol ossification. In conclusion, this thesis aims to offer insights into the potential and limitations of partial reliability in web content delivery, laying the groundwork for future research in this evolving field.

## 1 Introduction

The rapid evolution of Internet technologies has brought about increasingly complex and dynamic web applications, creating a pressing need for more adaptable and efficient transport protocols [59]. As network infrastructures advance—offering higher data rates and enhanced communication capabilities across both wired and wireless environments—there has been a concerted effort within the research community to develop transport layer protocols that can fully exploit these advancements and optimize overall network performance [74].

Historically, data transfer on the Internet has relied on protocols like Transmission Control Protocol (TCP) and User Datagram Protocol (UDP), which were originally designed for simpler, static web pages [63]. However, the modern web, characterized by sophisticated and interactive platforms, has exposed the limitations of these traditional protocols [31]. In response, Quick UDP Internet Connections (QUIC) has emerged as a powerful alternative, addressing many of these limitations by offering reduced latency, enhanced security, and more efficient performance [28][64]. The integration of QUIC as the transport layer for HTTP/3 has further improved web performance, providing significant advantages over previous versions of the HTTP protocol [72][26][25][7].

While numerous optimization strategies have been explored across different layers of the network stack [74], the potential of employing partial reliability in conjunction with QUIC for optimizing web content delivery remains underexplored. A notable innovation in this context is the "Unreliable Datagram Extension" for QUIC [48], which introduces *DATAGRAM* frames. These frames facilitate the transmission of data that does not require guaranteed delivery or in-order processing, thereby prioritizing low-latency delivery at the potential cost of data loss. This approach challenges conventional paradigms of web content delivery by allowing for selective reliability based on the importance of the data being transmitted.

This thesis investigates the concept of *Partially Reliable Transport with QUIC*, leveraging QUIC's datagram extension to achieve nuanced control over the reliability of data transmission at the transport layer. The central hypothesis posits that certain web page elements—such as CSS styles, advertisements, and font files—can be classified as 'Non-Essential' for reliable delivery under constrained network conditions. By selectively transmitting these non-essential elements via unreliable QUIC datagrams while reserving reliable streams for critical components, the aim is to enhance data delivery efficiency and improve the overall user experience under constrained network environment.

**Specifically, this thesis aims to:**

(1) Review existing literature on improving the web browsing experience and demonstrate the potential of QUIC and partial reliability systems for optimizing web content delivery.
(2) Evaluate the advantages of QUIC over TCP, particularly under challenging network conditions.
(3) Analyze various websites and their assets to determine which elements can be classified as 'Essential'

or 'Non-Essential' based on their impact on network performance.
(4) Conduct emulations to deliver web assets classified by their essentialness using fully and partially reliable transport, observing the effects on network performance and latency.
(5) Gather User insights on their preferences regarding web page loading and its elements during periods of slow connectivity.
(6) Present the results, go over the ramifications of classifying some web page elements as optional, explain the advantages of using partially reliable transport to deliver web assets, and evaluate its viability in real-world situations.

In order to test this hypothesis, we use a strong method that combines theoretical analysis with real-world testing. At first, we carefully chose a sample of test websites, avoiding content-specific biases to examine. Web elements are then labeled as "Essential" or "Non-Essential" based on a series of tests that can assess how these test web pages and their objects can affect network and page functionality and it's usability.

The included rigorous testing using a QUIC-supporting Chromium browser and the *Mahimahi*, *WebPageTest*, *TamperMonkey*, *Mitmproxy* tools to record and replay different versions of the test websites under controlled conditions. Additional performance assessments and network simulations are conducted with tools such as *Lighthouse* and Chrome *DevTools*. The performance of test websites is also compared across *TCP + HTTP/2* and *HTTP/3 + QUIC*, under varying network scenarios.

To explore the latency benefits offered by QUIC datagrams for non-essential web elements, an emulation server-client setup is developed using the Quiche library by Cloudflare [12], which supports the QUIC Datagrams extension [48]. This emulation replicates a web server environment similar to a Content Delivery Network (CDN), delivering static assets to a client over HTTP/3 and QUIC streams, as well as datagram chunks. The total fetch times of these assets for the web pages are then measured under challenging network conditions, comparing the timing differences between delivery via streams, datagrams, and a combination of both. Additionally, the results of our user survey provide insights into user preferences and tolerances for partially loaded web pages with delayed web objects, informing potential strategies for categorizing web elements and implementing partially reliable transport in real-world scenarios.

Furthermore, this thesis research also uncovers significant challenges in implementing partially reliable transport for web content delivery. We find out that while the use of unreliable datagrams for 'Non-Essential' web assets can reduce latency, especially under poor lossy network conditions, it also introduces risks related to data integrity and rendering, ultimately affecting the Quality of Experience (QoE) for users. The current structure of web technologies, which often requires full integrity for files such as fonts, CSS, and ad content, is not inherently suited for partial delivery. Unlike video or image files, which can degrade gracefully, the partial delivery of these elements can result in incomplete files, leading to corruption or missing data, and potentially causing issues in rendering and overall page layout.

From a transport protocol perspective, ensuring the reliable operation of HTTP/3 over QUIC in the presence of unreliable datagrams presents a significant challenge. Unlike reliable streams, unreliable datagrams lack built-in flow control, necessitating the development of custom mechanisms at the application level. Additionally, QUIC datagrams face size limitations to prevent IP fragmentation, requiring sophisticated logic for fragmentation and reassembly at the application layer for larger files. Despite the theoretical advantages of this hybrid approach, the practical complexities and performance trade-offs demand careful consideration as well. The use of datagrams requires advanced application logic to manage packet loss and reordering effectively. These findings highlight the need for application-level intelligence to manage partial delivery and ensure a consistent user experience.

Ultimately, this thesis examines the dual capabilities of QUIC and proposes a novel approach to web content delivery, contributing to the ongoing development of web protocols and advancements in the transport layer for web applications. The findings of this thesis have implications for stakeholders across the web ecosystem, including content delivery networks, web developers, and protocol designers.

## 2 Background

Over the past three decades, the protocols underpinning web content delivery have evolved significantly, driven by the increasing complexity of web applications and the need for improved performance under varying network conditions. This section provides an overview of key transport layer protocols and the methods developed to enhance web performance.

*2.0.1 TCP and UDP* The *Transmission Control Protocol* (TCP), originally specified in the 1970s, has long been the backbone of reliable data transmission on the Internet [53]. TCP's strengths lie in its ability to provide ordered, error-checked delivery of a byte stream, making it suitable for a wide range of applications. Its segment structure includes headers for source and destination ports, sequence and acknowledgment numbers, flags, and a sliding window for flow control. TCP also employs a three-way handshake (`SYN`, `SYN-ACK`, `ACK`) to establish connections, ensuring reliability at the cost of speed. Moreover, it incorporates congestion control algorithms like *Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery* to maintain network stability [74].

However, TCP's design, suitable for the networks of the 1980s, now imposes limitations in modern high-speed networks. The protocol suffers from *head-of-line blocking*, where all subsequent segments are delayed if a single segment is lost, a significant issue in high-latency or lossy networks common today. Additionally, TCP's connection establishment overhead introduces latency, especially problematic for the short-lived connections typical of web traffic [59]. The widespread deployment of middleboxes—devices

that inspect, filter, and manipulate traffic—complicates modifications to TCP, contributing to what is known as "***protocol ossification***" [47].

In addition to TCP, the *User Datagram Protocol* (UDP) [52] has been a crucial component of the Internet protocol suite. UDP provides a connectionless and unreliable datagram service, with minimal overhead and no guarantees of delivery, ordering, or congestion control. Its header structure is simple, comprising only source and destination ports, length, and checksum fields. This simplicity allows for efficient transmission with minimal delay, making UDP ideal for applications where low latency is prioritized over reliability. However, the absence of built-in congestion control in UDP can lead to network instability if used without caution [63].

*2.0.2 SCTP and DCCP* To address the limitations of both TCP and UDP, researchers have proposed alternative transport protocols. One such protocol is the *Stream Control Transmission Protocol* (SCTP), standardized in 2000 [66]. SCTP's packet structure, which allows for the transmission of one or more chunks within a common header, introduced flexibility in data transmission. Its support for *multi-streaming* reduces head-of-line blocking by enabling multiple independent streams within a single connection. Additionally, SCTP supports multi-homing, allowing the use of multiple network interfaces to enhance reliability and facilitate smoother handovers. Unlike TCP's byte-stream model, SCTP preserves message boundaries, simplifying application-layer framing [67]. Despite these innovations, SCTP has seen limited adoption on the public Internet, primarily due to a lack of support from middleboxes and operating systems [51].

Another effort to overcome TCP's limitations is the *Datagram Congestion Control Protocol* (DCCP), standardized in 2006. DCCP was designed to offer an unreliable transport protocol with congestion control. Key features of DCCP include a flexible congestion control framework that allows applications to choose from various algorithms, connection management mechanisms, and a feature negotiation system [33]. While DCCP addressed critical needs, particularly for real-time applications that require congestion control but can tolerate some packet loss, its adoption has been limited. This is largely due to interference from *middleboxes*, which often block or mishandle DCCP traffic, and a lack of native support in networking stacks and operating systems [59]. The absence of this infrastructure support discouraged application adoption, and vice versa, ultimately preventing DCCP's widespread deployment.

*2.0.3 HTTP 1/2/3* The evolution of the Hypertext Transfer Protocol (HTTP) from version 0.9 to 3.0 illustrates significant advancements in web communication protocols, with each version addressing the shortcomings of its predecessor and adapting to the evolving needs of web applications and internet connectivity.

HTTP/1.1, standardized in 1999, laid the groundwork for web communication for many years [16]. It introduced several key improvements over HTTP/1.0, including persistent connections and *pipelining*. Persistent connections allowed multiple requests to be sent over a single TCP connection, thereby reducing the overhead associated with repeatedly establishing new connections. *Pipelining* enabled clients to send multiple requests without waiting for responses, theoretically improving performance. However, HTTP/1.1 had several notable limitations, such as the lack of header compression, which led to redundant header information being transmitted with each request, increasing overhead. Additionally, its inefficient use of TCP connections meant that, despite the benefits of persistent connections, browsers often resorted to opening multiple TCP connections to mitigate performance issues [25]. These limitations became increasingly problematic as web pages grew in complexity, often requiring dozens or even hundreds of resources to render both sequentially and simultaneously [8].

To address these challenges, HTTP/2, standardized in 2015 [4], introduced several critical enhancements to improve web performance. These included multiplexing, which allowed multiple streams of data to be sent concurrently over a single TCP connection, thereby eliminating head-of-line blocking at the application layer. It also featured header compression using the HPACK algorithm, which significantly reduced header overhead [50]. Additionally, HTTP/2 introduced server push capabilities, enabling servers to proactively send resources to clients to potentially reduce latency, and stream prioritization, allowing clients to specify the relative importance of requested resources for more efficient management. Despite these improvements over HTTP/1.1, HTTP/2 continued to rely on TCP as its transport protocol, thereby inheriting TCP's limitations, especially in mobile and lossy network environments [70].

The introduction of HTTP/3, which uses QUIC as its transport mechanism, marked a major change in the evolution of the protocol [7]. It fixed many of HTTP/2's remaining issues and added some new ones. For example, it got rid of *head-of-line blocking* (HOL) at the transport layer by using QUIC's stream-based architecture. This was a problem in HTTP/2 because TCP works with byte streams. Moreover, HTTP/2 needed a TCP+TLS handshake to establish a connection; HTTP/3's 0-RTT and 1-RTT handshakes cut down on the time needed to do so, leading to much lower latency. In addition, QUIC's design makes it easier to recover from loss and move connections, which helps HTTP/3 work better in tough network conditions like mobile and lossy ones. The built-in encryption from QUIC makes security automatic and simplifies the protocol stack by combining the transport and security layers [63]. HTTP/3 also adds *QPACK*, a header compression scheme that works better with QUIC's out-of-order delivery model and makes header transmission faster [34].

**HTTP Versions release timeline**

- **HTTP 0.9 (1991)** — Tim Berners-Lee introduced the first version of HTTP in 1991. It was a simple protocol supporting only the GET method, with no headers or status codes. The response was limited to HTML content [26].
- **HTTP 1.0 (1996)** — Officially released as RFC 1945 in 1996, HTTP 1.0 introduced request and response

headers, allowing for metadata exchange and support for various content types beyond HTML [5].

- **HTTP 1.1 (1997-2014)** — First published as RFC 2068 in January 1997, HTTP 1.1 brought significant improvements including persistent connections, pipelining, and enhanced caching mechanisms. It was later updated in RFC 2616 (1999) and RFC 7230-7235 (2014) [16][17].
- **HTTP 2.0 (2015)** — Standardized as RFC 7540 in 2015, HTTP/2 introduced a binary framing layer, multiplexing, and header compression, focusing on performance improvements [4].
- **HTTP 3 (2022)** — Officially published in 2022 as RFC 9114, HTTP/3 is based on the QUIC protocol, which runs over UDP instead of TCP, aiming to reduce latency and improve performance in challenging network conditions [7].

Once HTTP/3 and QUIC are more stable and widely used, they should be able to solve many of the problems that modern web apps have, especially when they are used in complex and varied network settings[59].

## 2.1 QUIC: The Internet's new transport protocol

QUIC (Quick UDP Internet Connections) represents a major advancement in transport layer protocols, specifically designed to overcome the limitations of traditional TCP/TLS/HTTP stacks and to improve performance in modern Internet environments.

*2.1.1 Goals and Design Principles* To address the drawbacks of current transport protocols, particularly in terms of latency, HOL blocking, and security constraints, Google first developed QUIC in 2012 as an experimental protocol for web applications. After several years of testing and refinement, Google submitted QUIC to the *Internet Engineering Task Force* (IETF) in 2015, leading to the formation of the IETF QUIC working group in 2016. Following extensive discussions and modifications, QUIC was standardized in May 2021 with the publication of RFC 8999 through RFC 9000 [28].

QUIC was conceived with several primary objectives: reducing connection establishment latency, eliminating HOL blocking, supporting connection migration, and providing built-in security [64]. These goals were realized through innovative design features and careful protocol engineering, addressing the inadequacies of TCP and TLS in handling modern Internet traffic [37].

TCP, despite its historical significance, suffers from inherent limitations, particularly in its inability to scale effectively with modern traffic demands. HOL blocking remains a critical bottleneck, even with HTTP/2's multiplexing efforts at the application layer. Additionally, TLS introduces overhead, particularly due to the round-trip time (RTT) required before data transmission can begin, as well as redundant packet sizes [57]. These limitations motivated the development of QUIC as a more flexible and efficient alternative.

By operating over UDP, QUIC avoids many of the ossification issues associated with TCP. This choice in design lets

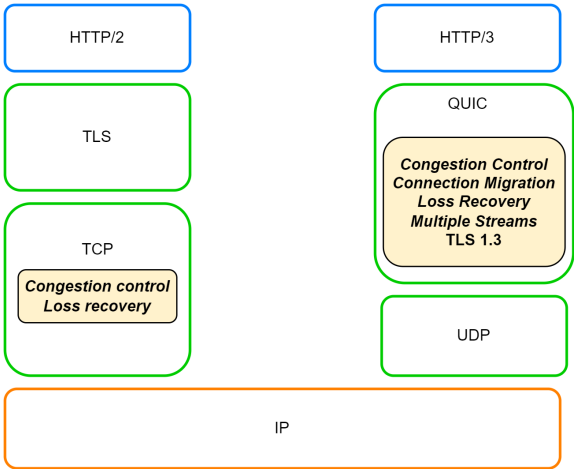



**Figure 1: Protocol Stack of TCP and QUIC, as described in 2.1.1.**

QUIC set up its own connection management, flow control, and congestion control in user space, as shown in 1. This lets it adapt, change, and be faster without having to make changes to the kernel [37]. This user-space implementation makes QUIC easier for developers to integrate and adopt, facilitating its widespread use in modern web applications. QUIC's application-layer implementation uses UDP to multiplex streams and sessions, eliminating the need for *kernel-level* intervention [36]. Additionally, QUIC uses the *BBR* congestion control algorithm [60] to handle congestion. This algorithm changes its congestion window, pipe modeling, and pacing rate based on predictions of bottleneck bandwidth and round-trip propagation time. This adaptability allows QUIC to maintain optimal throughput and minimize queuing delays, even under fluctuating network conditions [10]. However, the increased complexity of QUIC's packet processing and crypto-graphic operations can lead to higher CPU utilization, which must be carefully evaluated when considering QUIC-based solutions over TCP.

*2.1.2 Multiplexing and Stream Management* One of QUIC's most significant innovations is its approach to multiplexing. Unlike TCP-based protocols, where HOL blocking can affect all streams in the event of packet loss, QUIC allows multiple streams of data to be sent concurrently over a single connection, with each stream maintaining independent flow control [37]. This design ensures that packet loss impacts only the specific streams involved, rather than blocking all streams, thereby improving overall connection performance.

QUIC abstracts raw data transfer into streams, which can be either unidirectional or bidirectional flows of data between the client and server [28]. These streams are composed of STREAM frames, which are bundled into packets for transmission. This architecture not only supports efficient multiplexing but also minimizes the impact of packet loss or delay, enhancing the reliability and speed of data transmission across the connection.


**Figure 2: Frame Structures of QUIC STREAM frame (top) and HTTP/3 STREAM frame (bottom).**

*2.1.3 Connection Establishment and Security* QUIC makes connection establishment faster, especially for the quick, short-lived connections common in web browsing, thanks to its 0-RTT handshake capability. This feature allows clients to start sending data right away during the first round trip of a new connection by combining the crypto-graphic and transport handshakes [29].
QUIC is also built with security in mind. It uses encryption and authentication based on TLS 1.3, right out of the box. This built-in security not only boosts privacy but also helps QUIC avoid interference from *middleboxes* and prevents the kind of protocol ossification that hampers older protocols. By reducing the number of necessary round trips to just **one**, QUIC enables quicker connections and faster data transfer [64].

*2.1.4 Congestion Control and Performance Optimization* QUIC's approach to congestion control is designed to be adaptable. The default congestion control algorithm is based on TCP NewReno, as outlined in RFC 9002 [27]. Unlike TCP, which manages congestion on a byte-stream basis, QUIC handles it per packet, allowing for more precise control and faster recovery from packet loss. It also includes features like explicit congestion notification (ECN) and pacing to smooth out data transmission and avoid bursts [27].
Early tests have shown that QUIC can significantly improve performance, especially in situations with low bandwidth, high latency, or high packet loss. For instance, Google reported a 3% reduction in average page load times and a 30% decrease in YouTube re-buffering events [64]. However, in more stable and well-optimized networks, the benefits of QUIC might be less dramatic compared to HTTP/TCP setups [29].

## 3 Multi-path Strategies

Multi-path networking has emerged as a promising solution to overcome the limitations of single-path connections, especially in heterogeneous network environments. At the transport layer, multi-path protocol solutions like MPTCP [11], MPQUIC [69], CMT-SCTP [68], and *DChannel* [62] use multiple network interfaces at the same time to make web applications run faster and more reliably.

*3.0.1 Multi-path TCP (MPTCP)* MPTCP extended the traditional TCP protocol to utilize multiple paths concurrently. It employs a *token-based* connection identification system to manage these paths. To ensure fairness and control over regular TCP flows, MPTCP uses coupled congestion control algorithms like *Linked Increases Algorithm* (LIA) and *Opportunistic Linked Increases Algorithm* (OLIA) [32]. MPTCP is designed to work with different scheduling algorithms, like round-robin, lowest-RTT-first, and redundant transmission strategies, so that data can be sent quickly and efficiently across all available sub flows [11]. Additionally, it enhances reliability by redirecting traffic to available paths in case of path failures, thereby maintaining an uninterrupted connection.

**Table 1: Comparison of Multipath Transport Protocols**

| Feature | MPTCP | MPQUIC | CMT-SCTP |
|---|---|---|---|
| **Protocol** | TCP | UDP | SCTP |
| **Connection** | Subflows | Conn. IDs | Assoc. |
| **Security** | TLS/subflow | TLS 1.3 | DTLS (opt.) |
| **HoL Blocking** | Subflow | None | Assoc. |
| **Middlebox** | Challenging | Better | Limited |
| **Congestion Ctrl** | Coupled | Indep. | Indep. |
| **Reliability** | Full | Config. | Config. |
| **Use Cases** | General | Web, RT | Data ctrs, telephony |

*3.0.2 Concurrent Multipath Transfer SCTP (CMT-SCTP)* *Concurrent Multi-path Transfer SCTP* (CMT-SCTP) extends the *Stream Control Transmission Protocol* (SCTP) to enable the concurrent use of multiple paths [68]. Unlike MPTCP, which operates on a subflow concept, CMT-SCTP maintains a single SCTP association across multiple paths through a feature called *Association Management*. It addresses buffer blocking issues by implementing advanced techniques such as *split fast* re-transmit and delayed acknowledgment. Furthermore, CMT-SCTP's "chunk-oriented" transmission allows for more flexible scheduling and re-transmission strategies compared to byte-stream protocols, offering improved performance in multi path scenarios.

### 3.1 Multi-path Support in QUIC

Initially, QUIC lacked support for multiple network paths, a significant limitation given that mobile devices frequently connect to both cellular networks and WiFi simultaneously. To overcome this, *Multi-Path QUIC* (MP-QUIC) was developed, allowing data to be transmitted concurrently over multiple network paths [76]. It identifies these paths during

connection setup and ensures that data is delivered in the correct order across them.
It's important to differentiate MP-QUIC from MPTCP, which is a separate protocol extension for TCP. Unlike MPTCP, MP-QUIC is specifically designed for QUIC and takes advantage of QUIC's unique features and architecture. MP-QUIC aims to integrate seamlessly with existing QUIC implementations [35]. At the network level, MP-QUIC traffic appears similar to regular QUIC traffic, allowing it to traverse *middleboxes* and firewalls that support QUIC. This compatibility enhances performance for applications that transfer large amounts of data, increases throughput across different network scenarios, and provides smoother transitions between network connections[35].

*3.1.1 Design* MP-QUIC shows a design advancement in multi-path networking, addressing critical challenges through innovative design choices. For instance,

- An ADD_ADDRESS frame for advertising paths in dual-stack environments.
- A smart path ID system that uses odd and even numbers for client- and server-initiated paths, respectively.
- A sophisticated packet scheduling algorithm that balances low latency with congestion avoidance.

This helps the protocol effectively handle path identification, reliable data transmission, dynamic path management, efficient packet scheduling, and fair congestion control across multiple paths[76].

# 4 Partial Reliability

*4.0.1 Partial Reliability in TCP* Although several advanced approaches have introduced partial reliability for TCP, these methods still face significant challenges and limitations that hinder their widespread adoption in web applications. One such approach is *Partial-Reliable TCP* (PR-TCP), which implements selective re-transmission to prioritize certain packets, potentially improving efficiency for applications with varying data importance. According to [39], PR-TCP outperforms TCP *Reno* and UDP, showing at least a 12% improvement. However, this approach introduces complexity in packet management and risks disrupting TCP's established congestion control mechanisms. Additionally, the dual implementation options (**Basic** for both sender and client, and **Single-Side** for sender only) create interoperability issues, complicating its broader adoption [39].
Another variant, *PRTP-ECN*, focuses on Quality of Service (QoS) for soft real-time constraints, trading off between reliability and *jitter* characteristics. This trade-off, however, limits its applicability across different types of applications [22]. 'E2TCP,' another variant, offers partial reliability with an emphasis on energy efficiency, particularly relevant for wireless environments. Despite its potential, E2TCP seems to struggle balance energy conservation while maintaining acceptable data integrity across diverse network conditions [13].

*Challenges* These approaches all encounter the challenge of maintaining TCP-friendliness and fairness when coexisting with standard TCP flows in the network. While each method addresses specific aspects of partial reliability, they collectively show the difficulty of modifying TCP's core behavior without compromising its strengths or introducing new complexities at hardware and software level [40]. Furthermore, the increasing complexity of dynamic, content-heavy web pages presents additional optimization challenges. Additionally, the growing focus on end-to-end encryption and user privacy adds further constraints to content delivery optimization techniques [43].

## 4.1 Partial Reliability with QUIC

As discussed in section 4.0.1, attempts to implement partial reliability within TCP have not fully met the demands of modern network environments. Consequently, the networking community has shifted its focus from modifying TCP to evolving QUIC. The QUIC protocol, especially with the addition of the Datagram extension described in RFC 9221 [48], is a big step forward for making modern network protocols partially reliable. This extension supports the transmission of unreliable datagrams over a QUIC connection, addressing the critical need for low-latency, best-effort delivery in today's applications.

*4.1.1 The QUIC Datagram* The primary objective of the QUIC Datagram extension is to allow the possibility of concurrent transmission of both unreliable and reliable data over a single QUIC connection. This extension leverages QUIC's inherent security and congestion control mechanisms while providing a low-latency, best-effort delivery service.
The QUIC Datagram extension adds the `DATAGRAM` frame type. This frame has a Frame Type field, an optional Length field (indicated by the 'L' bit), and the Datagram payload. Support for receiving `DATAGRAM` frames is advertised through the `max_datagram_frame_size` transport parameter during the QUIC handshake. This parameter indicates the maximum size of a `DATAGRAM` frame (including frame type, length, and payload) that the endpoint is willing to receive.
Unlike other solutions discussed further in section 8, which are modifications of reliable systems, the QUIC Datagram was designed to be **inherently unreliable**. It follows a no re-transmission policy for lost frames and is exempt from flow control mechanisms. This design allows for out-of-order delivery without requiring acknowledgment from the receiver, making every delivery a "best-effort" delivery.

However, due to their unreliable and unordered nature, it also suggests that QUIC datagrams are not ideal for sending entire files that require guaranteed delivery. To send a file using datagrams, it must be split into smaller *chunks* that fit within the maximum datagram size, with each chunk sent as a separate datagram. The application must implement its own sequencing and reassembly logic, possibly by adding sequence numbers to each chunk, and handle error detection and retransmission requests for missing or corrupted chunks [48]. Flow control and congestion control still apply to the chunks, so the sender must also be mindful of network conditions. For large files, the hybrid approach could be beneficial—sending the bulk of the file reliably via QUIC streams, while using datagrams for time-sensitive updates

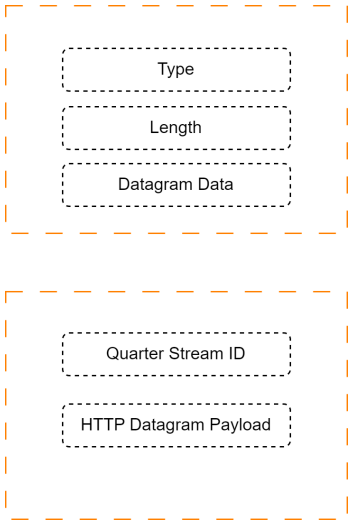



**Figure 3: Frame Structures of QUIC DATAGRAM frame (top) and HTTP/3 DATAGRAM frame (bottom).**

or patches. Furthermore, Datagrams may be particularly more suitable for real-time streaming scenarios where some packet loss is acceptable. Additional metadata about the file, such as size and name, should be sent reliably via streams rather than datagrams.
Additionally, QUIC Datagram frames do not include a built-in mechanism for *demultiplexing* application contexts. To address this limitation, HTTP/3 [7] introduces a payload identifier at the start of it's own datagram frame at application layer called `quarter stream ID` (shown in 3), facilitating all datagrams associated with a request can be allotted to the specific stream and managed by the transport layer, when using HTTP/3 over QUIC. Hence, the Datagram extension's ability to balance reliability with low-latency delivery can position QUIC as a versatile and forward-looking protocol, well-suited to the diverse requirements of web applications. However, due to the application-layer logic required, sending entire files via datagrams is currently a less explored approach.

## 5 Methodology

In this section, we detail the design and techniques used to evaluate the potential of partially reliable transport with QUIC for optimizing web page performance. Our methodology integrates network analysis, website evaluation, and user experience assessment under various network conditions, allowing us to examine both the theoretical potential and practical feasibility of our proposed optimization techniques.

### 5.1 Data Collection and Preparation

*Tranco List Analysis* Although the internet is vast, the adoption of newer protocols like QUIC and IPv6 remains uncertain, making it challenging to gauge their real impact and potential. QUIC aims to enhance web speed, while IPv6 is crucial for sustaining internet growth. However, both protocols face significant adoption barriers. By identifying websites that support these protocols, we can better understand their current influence and uncover opportunities for broader implementation.

To establish a baseline understanding of protocol adoption, we analyzed data from the top 100,000 public domains, sourced from the *Tranco list* [38]. The Tranco List was chosen for its ranking of popular websites, which provides a stable and reliable source for research-based use, avoiding issues like inconsistencies, daily fluctuations, and manipulation [38].

To tailor the list to our specific requirements, We customized the original top 1 million domains list by applying the following pre-configured options before downloading:

- **List Configuration:**
  - **Sub-Lists Used:** CrUX, Majestic, Radar, Umbrella
  - **Time Frame:** 30 days
  - **Combination Method:** Harmonic progression (first domain gets 1 point, second 1/2, …, last 1/N points, not present = 0 points)
  - **Domain Aggregation:** Limited to prefixes of length 100K
  - **Filters Applied:**
    - Include domains present for at least 1 day
    - Include domains in at least 1 list
    - Exclude domains flagged by Google Safe Browsing
    - Include only pay-level domains (without subdomains)
    - No filtering by effective top-level domain/public suffix
  - **Final Output List:** 100K domains

The final output was a .csv file containing a ranking of the top 100,000 most popular domains, generated by aggregating and filtering data from multiple source lists to ensure relevance and up-to-date accuracy for our analysis needs.

To assess QUIC and IPv6 support across this extensive dataset, a custom crawler was developed as a *Python* script. This script systematically evaluated HTTP/3 over QUIC and IPv6 support for each domain in our .csv list file. Leveraging the *asyncio* and *aiohttp* libraries, the script performed asynchronous requests, enabling efficient parallel processing.
IPv6 support was determined using the `getaddrinfo()` function within *asyncio*, which resolves domain names and returns address information entries. The script found IPv6 support by looking for entries with the IPv6 address family (`socket.AF_INET6`) and specifically by checking the DNS configuration for **AAAA** records. For QUIC support, the script made HTTPS requests via *aiohttp* and inspected the 'Alt-Svc' (Alternative Services) header in the responses. It checked for QUIC-related identifiers, such as `['quic', 'h3', 'h3-29', 'h3-28', 'h3-27']`. The presence of 'h3' indicated HTTP/3 support, while 'quic' or any 'h3-xx' versions indicated QUIC support.
The script executed with a runtime exceeding two hours and generated a summary report, including the total number of domains analyzed and the percentage supporting each protocol, in a new *.csv* file. To enhance robustness, a custom *CookieJar* was implemented for cookie management, and

a retry mechanism was added to reattempt failed requests at least once to ensure comprehensive coverage of active domains.

*5.1.1 Website Selection* For our performance experimentation and testing of web pages, we selected websites from diverse categories, ensuring varied elements to minimize bias and maintain neutrality. To achieve a representative sample of average website structure and behavior, certain categories were deliberately excluded:

- Video streaming platforms (e.g., YouTube, Twitch, Netflix)
- Adult content websites
- Sports streaming platforms
- Image-intensive sites (e.g., Tumblr, Pinterest, Instagram)
- Online gaming sites (including multiplayer games and gambling platforms)

These exclusions were based on the association of these categories with specific content types or technical requirements that could skew our results. Video streaming sites focus on video delivery, while image-intensive sites are optimized for handling large volumes of visual media. Similarly, online gaming sites have unique performance demands that do not align with general web browsing behavior. By omitting these categories, we reduced the influence of content-specific optimizations, ensuring that our findings are applicable to a broader range of web development scenarios.

To generate our test dataset, we loaded and recorded 15 websites using a *Chromium* browser with `Cache disabled` and `No throttling` settings via *Chrome DevTools*. The data collection was conducted over two distinct days of the week (Friday and Sunday) and at two separate times (11 am and 9 pm) to mitigate potential time-of-day and day-of-week biases in website content. The two versions of the collected `HAR` (HTTP Archive) files were then stored for further analysis.

Our objective was to extract key information, including detailed network requests, precise timing data, content size, and response information, from the respective *.har* files. We executed another *Python* script to parse each *.har* file, which provided valuable insights into the performance characteristics of each website. Based on this analysis, we selected a representative mix of 10 websites for further testing, as shown in Table 2.

This final selection of 10 websites from the Tranco list was made to ensure a diverse range of web elements and content types to conduct a more comprehensive analysis of web page performance across different sectors. These sites, spanning categories such as *News, E-commerce, Blogs, Travel*, and *Business*, generated an average of **180 requests** and connected to approximately 15 unique servers (ref. 3) during a page load for their web content.

## 5.2 QUIC Performance Testing

Recent research states several advantages of QUIC over the traditional TCP, TLS, and HTTP/2 stack, particularly in design and performance across various network conditions [64, 74]. To quantify these benefits and assess QUIC's performance under challenging network conditions for our case, we established a custom bench-marking environment to evaluate its performance and suitability for Web Applications compared to TCP.

For our QUIC implementation, we utilized the Cloudflare's *Quiche* library, a fully-featured implementation of the QUIC protocol, supporting HTTP/3. In contrast, TCP tests were conducted using the standard Linux TCP stack with HTTP/2, reflecting real-world deployment scenarios.

*5.2.1 Quiche Selection* Among the various QUIC implementations available, including *picoquic*, *lsquic*, *msquic*, and *aioquic*, each exhibits different performance characteristics based on its design and interpretation of the QUIC specification [55]. To select the most suitable implementation for our study, we referenced the *QUIC Interop Test Runner* results [55], a framework that benchmarks interoperability and performance across QUIC implementations, revealing key differences between client and server behaviors.

Based on these results and a analysis of available libraries, we selected Cloudflare's *Quiche* library [12] for our experiments. *Quiche* was chosen for its performance-oriented design, strict adherence to the QUIC protocol, and superior compatibility with other implementations.

A notable advantage of *Quiche* is its integrated support for both HTTP/3 and QUIC datagrams, which was particularly beneficial for our investigation into partially reliable transport. *Quiche* library, still under continuous development,

**Table 2: Characteristics of selected Test Websites**

| Website | Type | Total Requests | Total Size (KB) |
|---|---|---|---|
| airbnb.com | Travel & hospitality | 282 | 4465 |
| booking.com | Travel & hospitality | 222 | 4267 |
| weather.com | Utility | 173 | 3009 |
| shopify.com | E-commerce | 186 | 2346 |
| ebay.com | E-commerce | 124 | 2248 |
| harvard.edu | Blogs | 99 | 14379 |
| nasdaq.com | Finance | 170 | 5512 |
| nytimes.com | News & media | 156 | 7263 |
| time.com | News & media | 186 | 8050 |
| washingtonpost.com | News & media | 219 | 5425 |

**Table 3: Number of unique servers used by each test website for web content delivery**

| File Name | Unique Servers |
|---|---|
| time.com | 28 |
| nytimes.com | 22 |
| weather.com | 19 |
| washingtonpost.com | 18 |
| harvard.edu | 15 |
| airbnb.com | 14 |
| nasdaq.com | 11 |
| booking.com | 8 |
| shopify.com | 8 |
| ebay.com | 6 |

allows concurrent transmission of reliable (streams) and unreliable (datagrams) data over a single QUIC connection, enabling applications to selectively choose transmission methods on a *per-message* basis. This capability, combined with QUIC's inherent security and congestion control, makes *Quiche* an ideal platform for exploring and implementing partially reliable transport systems.
Additionally, *Quiche*'s implementation in *Rust* provided crucial benefits, such as memory safety and efficient concurrency, which aligned with our research's high-performance and precision requirements.

*5.2.2 Experiment Setup* We constructed a controlled experimental environment using Quiche to evaluate QUIC's performance against TCP with HTTP/2. The setup involved two virtual machines (VMs) on a local Linux host, configured as client and server, both running 64-bit *Ubuntu* 22.04.4 LTS.
For data transfer tests, we prepared 50 MB files on both the TCP and QUIC server VMs. These files were transmitted to a custom QUIC client and a standard TCP client (using *wget*). To maintain experimental control and minimize confounding variables, we restricted our network to IPv4 addressing. This decision was informed by the prevalence of IPv4 in real-world scenarios, as evidenced by our analysis of the Tranco list [38]. The client VM initiated requests to the server VM, which hosted both QUIC and TCP servers. For QUIC tests, we used a custom client built with *Quiche*, while TCP tests employed *wget*. This configuration enabled direct performance comparisons under identical network conditions.

*5.2.3 Network Configuration* The network environment between the two VMs was carefully adjusted to mimic real-world scenarios. We set the *Maximum Transmission Unit* (MTU) size to `1500 bytes` at the interface level to ensure that QUIC packets could be transmitted without fragmentation. Network conditions were manipulated using traffic control (*tc*) tools to simulate different scenarios, including varying latencies and packet loss rates.

$$\begin{aligned}\text{QUIC Packet Size} &= \text{QUIC Payload} + \text{UDP Header} + \text{IPv4 Header} \\ &= 1350 \text{ bytes} + 8 \text{ bytes} + 20 \text{ bytes} \\ &= 1378 \text{ bytes} \\ &< \text{MTU (1500 bytes)}\end{aligned} \tag{1}$$

We used Linux traffic control tools such as *tc* and *netem* to create different network conditions with the following parameters:

- **Bandwidth limitations**: Set at 40 Mbps and 100 Mbps to cover typical use cases from web browsing to large file transfers.
- **One-way delay**: Latencies of 10ms, 50ms, and 120ms were introduced to simulate everything from local wifi delay to intercontinental connection delays.
- **Packet loss**: Controlled packet loss rates of 0% and 5% were applied to evaluate protocol performance under varying levels of data loss.

These configurations allowed us to test the performance of QUIC and TCP under a wide range of challenging conditions, from ideal local networks to congested, high-latency international connections. By varying these parameters, we could assess how each protocol performs in different scenarios, offering a comprehensive comparison.

*Testing* We conducted experiments to evaluate key performance metrics that demonstrate the advantages of QUIC over TCP. These tests focused on `Throughput`, `Total Transfer Time`, and `Protocol Overhead` under various network conditions for both protocols. We calculated protocol overhead using the formula:

$$\text{Overhead} = \frac{\text{Total bytes transferred} - \text{File size}}{\text{File size}} \times 100\%$$

Additionally, the `Packet Loss`, particularly relevant in wireless networks [59], was determined from sequence number gaps and explict loss signals.

Our test suite included:

(1) `Peak Throughput`: Measured at 40 Mbps and 100 Mbps to assess maximum data transfer rates.
(2) `Throughput vs Packet Loss`: Evaluated performance with 0% to 5% packet loss.
(3) `Protocol Overhead`: Compared data overhead at different packet loss rates.
(4) `Throughput vs Bandwidth and Delay`: Tested delay (10ms, 50ms) and bandwidth (40 Mbps, 100 Mbps) combinations.
(5) `Transfer Time vs Delay`: Measured the impact of a 120ms delay on transfer times.

Tools and Data Collection To ensure statistical reliability, each test was repeated *five*, and the results were averaged to account for any performance variations. Data collection involved standard network diagnostic tools and custom scripts to capture detailed performance metrics such as `tcpdump` for capturing and analyzing packet-level traffic. `iperf3` to measure throughput and jitter and `wget` for TCP/HTTP2 transfers, recording request and response times. After data collection, we filtered the raw data to focus on **timings, throughput and bytes transferred**, and used Jupyter notebook for visualization and analysis.

## 5.3 Understanding a Website

Related work in 8 has explored the relationship and behavior of web page elements, aiming to optimize load times using application layer knowledge . These studies have proposed various methodologies to categorize web page components as critical or "essential." However, there is no consensus on a standardized approach for this categorization. Each study employs its own unique criteria and methods to determine which elements are crucial for the initial rendering and functionality of a web page but there is still some general approach that can be followed.
According to [61], the number of objects requested is the most important factor when analyzing which metrics are most critical for predicting page rendering and load times. Furthermore, they suggest that the number of servers is the most reliable factor in determining the variation in load times [61].

In the following section, we explain our experiment and test bed used for better classification of 'Essential' and 'Non-Essential' elements for a web page. Our objective was to strike an optimal balance for improving page load times while preserving the elements that users find most valuable, hence 'Essential'.

*5.3.1 Web Page load* When a user navigates to a website, the client browser finds the IP address of the requested domain, either from stored cache or through DNS query, and sends an HTTP(s) request to the website host server. The host responds and sends back the requested HTML as a response [21].The browser processes the HTML markup and builds the *Domain Object Model* (DOM) tree. Then a *Cascading Style Sheets Object Model* (CSSOM) is built from the styles associated with the DOM. The DOM and the CSSOM are both individual data structures, where the DOM represents the content of the processed HTML and the CSSOM represents only the styling. The DOM and CSSOM are combined into a render tree. Hereafter, a layout is generated and finally, the web page is painted/rendered on the device of the client [21].

Our DOM tree analysis of our test websites consequently provided us the most obvious essential objects for a web page to be delivered reliably and load such as:

- *Critical Path* files: Inline or early-loaded styles that directly impact the initial render, minimizing Critical Path Length and reducing render-blocking resources (like critical CSS or JS).
- Initial HTML files: The core document structure, crucial for *First Contentful Paint* (FCP) and starting the *Critical Rendering Path* or if an element is fetched immediately after the initial HTML or repeatedly accessed.
- Render-Critical JavaScript files: Scripts that alter the original `CSSOM` or `DOM`, which has an immediate impact on how render trees of a webpage are constructed and laid out.

Since these components are integral to the critical rendering path and heavily influence rendering dependencies, their absence or delayed loading can significantly affect key performance metrics such as *First Contentful Paint* (FCP), *Largest Contentful Paint* (LCP), and *Time to Interactive* (TTI). Such disruptions can compromise the page's visual hierarchy and overall functionality [61]. Although optimization strategies like `code splitting`, `lazy loading`, and `resource prioritization` are crucial for enhancing performance, they have been extensively studied and fall outside the scope of this thesis 8.
In addition to the essential components, our study further explored the influence of less obvious elements on web page performance. The process of loading a modern web page is inherently complex, involving a multitude of factors, optimization strategies, and stakeholders, which precludes a universal solution [71].

*5.3.2 Categorization of Elements* Mislabeling a critically important web page object as "Non-Essential" can disrupt the DOM structure, leading to unpredictable formatting, and cause rendering issues in the browser [61] during page load. Therefore, it was crucial for us to carefully assess the 'Essentialness' of each web page object based on specific attributes that determine its importance.

To streamline this assessment, we began by reducing overlapping categories in the 'types' of request made by the web page for various page elements. This was achieved by filtering and sorting requests based on hints from their respective request *URLs* (also called *Initiators*)and the data we collected. We categorized the various objects/elements into **eight** broader but distinct 'Content Categories' for analysis, as *MediaType File* (MIME type): *HTML, CSS, JavaScript, Ads/Analytics, Fonts, JSON, Images*, and *Others*. This categorization enabled us to identify objects that could be considered Non-Essential, making them potential candidates for best-effort, unreliable transport.

In our study of partial reliability, we focused on two primary heuristics to determine the 'Essentialness' of web page objects:

**Network Impact:** This heuristic examines the back-end server perspective of a website, measuring how different object types affect overall network performance. We considered:

- The impact of each content type on network load and the number of network requests.
- The likelihood of certain elements being dropped or de-prioritized during poor network conditions.
- The load time for each object type over both reliable and partially reliable networks.

**User Impact:** This heuristic considers the front-end rendering from the user's perspective. Research by Dennis Guse [23] indicates that metrics like *Page Load Time* (PLT) alone are insufficient predictors of quality when partial load failures occur. Instead, Quality of Experience (QoE) metrics should be factored into the assessment of loading failures.

To evaluate QoE, we conducted a survey among internet users, presenting them with various iterations of the same website. Participants were asked to provide feedback on:

- Visual attractiveness of the web page.
- Preference for different web page variants.
- Overall score of the websites.

By integrating these two perspectives—Network and from Web users—we aimed to balance technical performance with user experience. This dual approach allowed us to make informed decisions about which elements are truly essential for a web page, considering optimisations for both - network efficiency and the end-user's experience.

## 5.4 Network impact

As mentioned earlier, the *number of requests* made and the *variety of servers* involved during a page load are critical factors that can significantly impact web page loading times [61]. As Michael Butkiewicz also found in his study that the number of requests has a greater impact on the loading time than the number of bytes transferred. Specifically, the browser sends an HTTP request each time the parser encounters a request for a new element. Each request requires a round trip to a server, causing increased loading time [61]. Consequently, we decided to examine network requests,

servers and content type during a web page load, by analysing our 10 test websites mentioned in 2 further.

*5.4.1 Network Request Analysis* To gain insights into the network behavior of our test websites (2), we performed a comprehensive analysis of the network requests. We began by parsing the collected data using a custom *Python* script, similar to the one described in Section 5.1.1, extracting baseline metrics such as the number of *web object types*, total *size per object type*, *number of requests*, and *average load times* across all 10 test websites.

For a robust examination of network requests and their assigned priorities, we utilized a open-source network interception tool called *MitmProxy* [1]. We set up *MitmProxy* by cloning and building the package from its official GitHub repository, configuring our testing environment (`LocalHost`) to route all traffic through *localhost:8080*, the default Mitmproxy port. This allowed *MitmProxy* to act as a man-in-the-middle, decrypting SSL/TLS connections and enabling real-time inspection and modification of network flows. For HTTPS interception, *MitmProxy* generated custom on-the-fly certificates using its own *Certificate Authority* (CA). We installed the root certificate on our local machine to enable transparent SSL/TLS inspection. Network traffic was logged using *MitmProxy*'s sub-tool, *mitmdump*. By executing `$ mitmdump -w example.log`, we captured complete HTTP(S) session logs for all test websites, ensuring consistent and reproducible request replays.

Furthermore, operating within the isolated Python environment of *MitmProxy*, we executed a custom script that processed the dump file using *MitmProxy*'s *io* module, generating a detailed analysis of network requests in *.JSON* format. This approach allowed us to extract and categorize data into attributes such as `"objectSize"`, `"contentType"`, `"priority"`, `"server_rtt"`, `"download_ms"`, `"request_type"`, and `"renderBlocking"`.

The `"priority"` attribute was analyzed specifically to identify resources deemed critical by the browser, marked as *Lowest, Low, High, Highest*. These priority labels, typically pre-defined by website developers, are included in request headers and indicate which resources are loaded first. Identifying objects with "Highest" and "Lowest" priorities allowed us to determine more easily which resources could be deferred without compromising user experience.

We examined the `"renderBlocking"` resources tag, as it is crucial for the initial rendering of meaningful content, thereby impacting metrics such as *First Contentful Paint* (FCP) and overall page performance. Additionally, we analyzed `"server_rtt"` to measure server responsiveness and `"download_ms"` to assess the download duration for each resource, particularly larger files. This comprehensive analysis of our test websites provided a clear understanding of the content being requested, its loading latency, and its role in the page rendering process.

*5.4.2 Device Variations* To analyze how web pages adapt to different devices, we used Chrome *DevTools* to simulate various devices for our test website. The tool allowed us to select device profiles from a drop-down menu within a responsive viewport. Given that 96.3% of global internet


**Figure 4: Original The New York Times Homepage with vs without additional Fonts and CSS style elements loaded. The Web page drops to default fonts and we observe *FOUT* (Flash of Unstyled Text) and layout issues.**

users access the web via mobile phones, 65.6% via laptops or desktops, and 27.3% via tablets [14], we focused our testing on these three device types: mobile phones, tablets, and desktop computers. Currently, web pages adjust to different screen sizes and capabilities through *responsive web design* as shown in 5. This begins with the `viewport` meta tag, which sets the initial scale and layout width. Developers use CSS media queries to apply styles based on screen size, defining breakpoints for mobile devices (up to 480px wide), tablets (481px to 1024px), and desktops (1025px and wider).

As the page loads, the browser applies device-specific styles, adjusting layouts (e.g., multi-column to single column), resizing images, and scaling other elements. To enhance performance on mobile devices, many designs employ techniques like lazy loading to delay non-essential resources.

Additionally, we observed the use of *progressive enhancement*, where a basic version of the site loads first, followed by additional features as the device's capabilities allow. This helped us identify which elements are deferred until later.

*5.4.3 Testing for Frontend SPOF* In this phase, our goal was to identify and analyze **Frontend Single Points of Failure (SPOF)** on websites, which can significantly degrade web performance by causing delays or even complete page load failures if certain resources fail to load. Identifying these SPOFs is essential for ensuring a resilient and user-friendly


**Figure 5: nytimes.com homepage variations across different device types 1) Mobile 2) ipad/Tablet 3) Desktop showcasing web page layout shift according to the user's device.**

website [20]. To conduct this analysis, we employed *WebPageTest* [2], a comprehensive web performance toolkit that provides detailed diagnostics on a web page's performance under various loading conditions. We applied this tool to our selected test domains to validate our findings on specific objects and assess potential performance improvements. Each test was conducted from different global locations using real browsers and customizable network conditions. We specifically utilized the SPOF feature in *WebPageTest*, which simulates the failure of specified domains by rerouting requests to *blackhole.webpagetest.org*, a server that silently drops all traffic.

*Test Website: nytimes.com* For our testing, we selected nytimes.com from our test database due to its generation of over 200 requests per page load, making it a representative example of a moderately content-rich website. We categorized these requests by MIME type [18], which allowed us to identify common hosting domains associated with standard MIME object types, such as HTML, CSS, JavaScript, fonts, images, and videos. For instance, we identified gt.nyt.com as the server hosting additional designer font files for nytimes.com. By blocking or delaying this domain, we could analyze the impact of delayed or absent font loading on the page's layout and rendering performance.

*Baseline and Modified Testing Comparison* We established a baseline by loading the original website under stable network conditions, generating a requests_log.csv and a waterfall chart. Next, we blacklisted requests from the gt.nyt.com domain, responsible for serving fonts, to observe how the website adapts when these fonts are not delivered and assess their criticality to the webpage. For the testing setup, we configured *WebPageTest* with a custom device profile, setting the browser dimensions to the default *desktop* resolution of 1920x1080, the most common screen resolution for laptops and desktops [59], and used *Google Chrome* as our test browser.

We selected the **`Amsterdam, Netherlands`** server location in *WebPageTest* due to its proximity to our testing site, minimizing additional latency. To simulate challenging network conditions and extend the data collection period for better analysis, we set the *connection speed* to 3G, standardized at **1.6 Mbps** *upload* and **768 Kbps** *download*.

*Focused Analysis on Critical Requests* Our analysis specifically targeted the first 50 requests made within the initial 20 seconds of the page load, as shown in the fig. 6. By narrowing our focus to this critical time frame, we were able to closely examine the elements and their associated elements (through DOM tree structure) that have the most significant impact on the *First Contentful Paint* (FCP) and initial rendering of the page. Additionally, this simulation testing allowed us to observe how the website behaves when certain elements are completely blocked from loading at the network level, based on their domains, sorted by their type. By comparing the original and modified versions of the websites, we were able to analyze the website's behavior under different variations and have avoid *Frotend SPOF* 5.4.3 causing elements to be categorised as Non-Essential.

*5.4.4 Testing with Mahimahi* To further evaluate which packets could be considered 'Non-Essential' under poor network conditions, we utilized network simulation tool called *Mahimahi* [45]. *Mahimahi* is a lightweight and accurate record-and-replay framework specifically designed for HTTP-based web applications. A key advantage of *Mahimahi* is its ability to emulate the multi-server nature of modern web applications, which is helpful for obtaining more realistic performance measurements. After building *Mahimahi* on

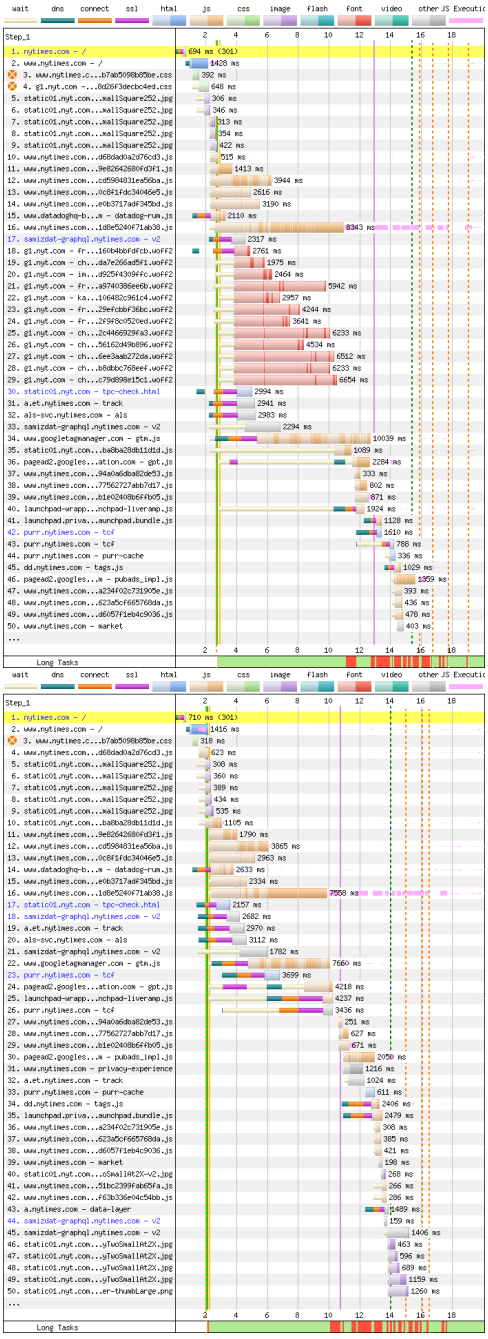



**Figure 6: *Nytimes.com* Waterfall Chart Eg. - Original page load vs Without additional font's domain page load**

our `localmachine`, we used Linux *network namespaces* to isolate web traffic, allowing us to run multiple experiments concurrently without interference. Additionally, *Mahimahi* supports unmodified commercial browsers, such as *Chromium* (which we used), ensuring that our tests were conducted in environments closely resembling real-world conditions.

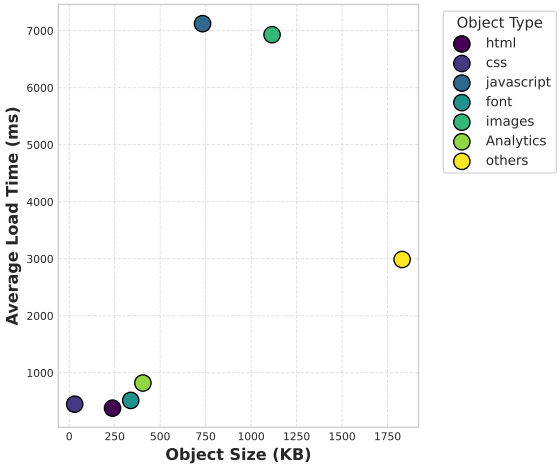



**Figure 7: Avg. Loading time vs Avg. Web Object Size based on content type of the web object**
Note: Measured over a fast and stable 4G speed connection

*Components* *Mahimahi* leverages *Apache* web servers to simulate the multi-server architecture of modern web applications. During replay, it instantiated a separate *Apache* instance for each recorded server, with each bound to a unique IP address. This setup preserved the multi-origin structure, ensuring accurate request routing and faithful emulation for us. The primary components of *Mahimahi* included the *RecordShell*, which captured HTTP(S) traffic, and the *ReplayShell*, which accurately replayed the recorded sessions. Network emulation shells such as *DelayShell*, *LinkShell*, and *LossShell* enabled us to simulate unfavourable network conditions. The composability of these tools allowed for complex simulations by nesting multiple effects, which was essential for our diverse testing scenarios.

*Additional Insights* While the original *Mahimahi* tool by [45] is effective, it only supports HTTP/2 and TCP, lacking native support for QUIC or HTTP/3. To conduct our tests on a QUIC and HTTP/3 server, we used an enhanced variant of *Mahimahi*, known as *bvc-Mahimahi* [6]. Developed by the Bilibili Video Cloud team, this version replaces the default *Apache* server with *Nginx*, enabling support for HTTP/3 (QUIC). We used this updated tool to perform performance testing of loading web pages, gathering metrics such as *Time to First Byte* (TTFB), *First Contentful Paint* (FCP), *Speed Index* (SI), and *Total Page Load Time* (PLT) by replaying web page between TCP and QUIC servers.

## 5.5 Network Tracing and Emulation

Next, we utilized *Mahimahi*'s `LinkShell` tool to emulate realistic cellular network conditions using real-world network traces from providers such as `AT&T` and `T-Mobile`. These traces, which are time-series data files, capture the instantaneous throughput of a cellular link over an extended period, typically spanning several minutes. Each line in a trace file represents a packet delivery opportunity, indicating when an MTU-sized packet (*1500 bytes*) can be transmitted over the emulated link.

`LinkShell` leverages these traces to create a high-fidelity emulation of cellular network behavior. It maintains separate `uplink` and `downlink` queues, releasing packets based on the corresponding input trace. This method accurately models the time-varying nature of cellular networks, capturing Bandwidth fluctuations, Latency Variations and burty behavior often seen in cellular networks. By incorporating these traces into our `LinkShell` during the web replay process, we conducted comparative analyses of the recorded web pages under different network conditions. This approach provided insights into how web applications perform under varying conditions, specifically identifying which elements were **deprioritized** or **dropped** under challenging network scenarios.

*5.5.1 Emulation Workflow* The first step in this process was to record our target test page using the *Recordshell*.`$ mm-webrecord /tmp/nytimes chromium-browser –ignore-certificate-errors`
This command, for example, captured all HTTP(S) traffic during a browsing session on the New York Times website. This recording saved the entire session for subsequent replay, allowing us to analyze the performance of the website when we replay it under just *Replayshell* or *Replay + Linkshell* combined. `–ignore-certificate-errors` flag instructs Chromium to ignore SSL/TLS certificate errors and proceed with loading websites even if their certificates are invalid, expired, or not trusted (for testing purpose).


**Figure 8: *Linkshell* embedded into Mahimahi when executing *Replayshell***

Next, we simulated various network trace using Mahimahi's `LinkShell`. To emulate the cellular network environments provided by AT&T and T-mobile, we executed `$ mm-link ATT-LTE-driving-2016.up ATT-LTE-driving-2016.down –meter-all`.
This was crucial as it adjusted the link conditions according to the real-world trace data, simulating the fluctuating bandwidth and latency typically experienced in cellular networks. During this phase, we also told the shell to log `uplink` and `downlink` performance metrics, which captured how the network conditions impacted the packets of the website. Finally, we replayed the recorded browsing session under these simulated network conditions using `$ mm-webreplay`

We repeated this process **four** times, testing variations of nytimes.com under both AT&T and T-Mobile network traces on different days of the week at random times. We recorded the webpage on *Monday*, mid-week on *Wednesday*, at the week-end on *Friday*, and finally on *Sunday*. Given that nytimes.com is a news website with frequently changing content, conducting tests on different days ensured that our data was not biased toward a single version of the website. This approach allowed us to identify the objects most frequently dropped across different page versions.


**Figure 9: Replaying *nytimes.com* over a T-Mobile network trace**

After simulating these four versions of the website, we analyzed the collected data (logs, .har files, and requests.csv files) to identify patterns in packet drops based on URLs and content types, correlating them with specific network conditions. This analysis revealed which types of packets were **most susceptible to drop under poor lossy network conditions** and at what points in the network trace these drops occurred.

Our detailed scrutiny identified which requests were essential for quickly delivering visible content during page load and which could be dropped or delayed. We observed that elements originating from the host domain *nytimes.com* as initiator were crucial for loading the basic structure and rendering of the page. In contrast, requests from domains such as *g1.nyt.com* (for fonts), *static01.nyt.com* and *vp.nyt.com* (for static images and videos), and various third-party domains like *accounts.google.com* and *pagead2.googlesyndication.com* related to ads, tracking, and JavaScript elements were dropped and could be considered non-essential and potentially delayed or de-prioritized from initial loading without significantly affecting the user experience under unfavourable network conditions.

Based on our analyses in Sections 5.4.3, 5.4, and 5.4.4, we developed a prioritization chart to guide our further analysis. This chart categorizes content types outlined in 4, organizing them by type, average requests per page load, average size per category, and whether they can be considered essential or non-essential based on our findings so far, mentioned in 4.

**Table 4: Content Category Analysis on Network Impact**

| Content Category | Avg. Requests | Avg. Size (KB) | Essential |
|---|---|---|---|
| Ads/Analytics | 9 | 405 | No |
| Images | 40 | 815 | Maybe |
| HTML | 5 | 238 | Yes |
| JavaScript | 29 | 733.5 | Yes |
| JSON | 16 | 120 | Maybe |
| CSS | 4 | 30 | Yes |
| Fonts | 12 | 338.1 | No |
| Others | 14 | 1130 - 3778 | Maybe |

## 5.6 Partially Reliable Transport

To compare the latency of reliable QUIC streams and partially reliable QUIC datagrams in web object transfer, we

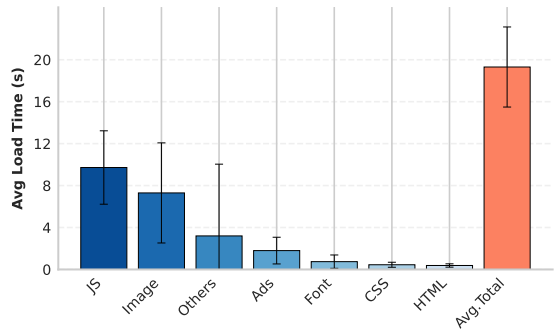



**Figure 10: Content type vs Avg. Load Time**
*Note: Data represents average load times across 4 test pages. Error bars indicate 95% confidence intervals. Total average load time: 19,309 ms ± 1200 ms (95% CI)*

developed a custom client-server emulator using the Quiche library. This emulator simulates a web environment where a server responds to sequential client requests by transmitting web object files. We designed the system to evaluate how effectively datagrams' low latency performs when delivering Essential and Non-Essential web objects compared to streams. The emulator uses both QUIC and HTTP/3 protocols to manage these transfers, closely mimicking real-world web applications. By measuring end-to-end delivery times in this controlled environment, we could assess the performance differences (if any) between the two transport methods.

## 5.7 Database

To facilitate comprehensive emulation, we started by conducting extensive data collection and analysis for our test website, *nytimes.com*. The website data was gathered over a weekend period (*Friday, Saturday, Sunday*), with collections occurring twice daily at `11 AM` and `9 PM`. This approach allowed us to capture six variations in the website's content and structure across different times and days, ensuring that our dataset reflected the dynamic nature of a news website.

To process and analyze this data, we developed a custom parsing tool in *Rust*, named *HAR_Replica*. This tool was a critical component of our *Web Objects_simulator* system, designed specifically to handle the collected HTTP Archive (HAR) files. The *HAR_Replica* tool enabled us to generate replicated dummy web objects, detailed request logs, and transfer sequences for each of the six versions of the website that we collected.

The *HAR_Replica* tool performed several key functions. Firstly, it parsed the HAR files, which contained detailed records of web browser interactions. It then categorized HTTP requests based on MIME type and URL, allowing us to classify the different types of content on the website. Following this, the tool generated binary files, termed *dummy_files*, that matched the size and type of the original web objects. These files were stored in an output directory, ready for use in subsequent simulations. In addition to file generation, the tool meticulously logged transfer details, including each object's ID, category, size, and original transfer time, into a log output file. For each version of the website, *HAR_Replica* also generated a *Transfer_sequence.json* file. This file contained an array of objects representing the resources transferred during the recorded session, with key attributes such as object ID, resource category, file extension, size, transfer time, start time, and a generated file name based on its ID and extension.

## 5.8 Emulator Design Overview

The emulator was designed to simulate a realistic web browsing scenario, with the server hosting a variety of dummy files representing typical web content such as HTML, CSS, JavaScript, fonts, and images. The client was programmed to request these files in a predefined sequence, mimicking the behavior of a modern web browser using HTTP/3 over QUIC. A key feature of the emulator is its ability to utilize both reliable (stream-based) and unreliable (datagram-based) transfer methods. The choice between these methods is determined by the nature of the content being transferred, with 'Non-Essential' objects being candidates for datagram-based transfer to prioritize speed over guaranteed delivery.

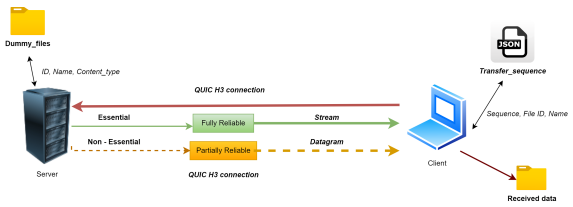



**Figure 11: Design Overview of our server-client emulator for network and partially reliable transport testing, as described in 5.8.**

*5.8.1 Server Implementation* The server component, named quiche_har_server, was engineered to emulate a web server capable of delivering content over QUIC. It manages HTTP/3 connections, handles a directory of dummy files, and processes file transfer requests using a file ID counter for unique identification. The server employs a dedicated event loop to manage HTTP/3 events, continuously polling for incoming requests. Upon receiving a file request, the server determines the appropriate transfer method based on the file type. This decision is made using a should_use_datagram() function, which assesses whether the file is suitable for datagram-based transfer. For datagram-based transfers, the server implements a custom protocol to handle the lack of built-in stream identifiers in QUIC datagrams. This protocol introduces a 'Quarter Stream ID' concept, allowing the server to manage multiple concurrent datagram transfers effectively.

*5.8.2 Client Implementation* The client component `quiche_har_client`, simulates a web browser by sequentially requesting files over HTTP/3. It manages QUIC and HTTP/3 connections, tracks file transfers, and collects comprehensive performance metrics. The client processes requests based on a predefined sequence specified in a transfer_sequence.json file. For each file, it sends an HTTP/3 GET request and handles the response accordingly. The client

is capable of processing both stream-based and datagram-based responses, adapting its behavior based on the server's chosen transfer method. A crucial aspect of the client implementation is its datagram processing mechanism. This system differentiates between metadata datagrams (containing information about the file transfer) and file chunk datagrams. The client uses the 'Quarter Stream ID' to track and reassemble datagram-based file transfers, handling potential out-of-order delivery and *packet loss*.

*Performance Metrics Collection* The emulator incorporates a comprehensive metrics collection system. For each file transfer, it records data such as total transfer time, end-to-end transfer time, number of bytes transferred, completion status, and the protocol used (stream or datagram). Additionally, it calculates aggregate metrics for the entire session, including total session duration, overall throughput, and the number of completed versus total objects. These metrics are systematically logged and compiled into detailed reports, providing insights into the performance characteristics of both stream-based and datagram-based transfers. This data forms the basis for analyzing the efficiency and effectiveness of using QUIC datagrams for web content delivery.

*5.8.3 Experimental Setup* The experimental setup was carefully designed to create a controlled environment for evaluating the performance of QUIC datagram-based transfers in comparison to traditional stream-based transfers. To minimize external variables, the experiments were conducted on a local network. The server was configured to listen on a specific IP address and port, `0.0.0.0:4433`, while the client connected to this endpoint, ensuring a consistent and reliable communication path. Both the client and server were configured identically to maintain uniformity across all experiments.

*Dummy File Repository* A set of six dummy_files directory were created using the six web pages *.har* files we captured, to represent a variety of web content types and sizes of each. Each directory included a diverse range of files, such as:
- Small HTML files (5–30 KB each)
- CSS stylesheets (12–50 KB each)
- JavaScript files (20–45 KB each)
- Images in various formats (JPEG, PNG, WebP) (300 KB – 2 MB each)
- Font files (WOFF, WOFF2, 33–86 KB each)
- Larger video media files (11–19 MB each)

Additionally, for each of the six page versions, corresponding `transfer_sequence.json` files were generated to guide the sequence of content loading during the experiments.

*Network Conditions* Initial tests were conducted under optimal network conditions to establish a baseline. To simulate more challenging network environments, traffic control (`tc`) was applied on the client virtual machine to limit bandwidth to realistic speeds typical of mobile connections. Given that 4G networks represent approximately 60% of global mobile connections [73], the tests were conducted under two specific conditions:

- Standard 4G Connection: Bandwidth limited to 12 Mbps.
- Constrained 4G Connection: Bandwidth reduced to 9 Mbps with an additional 5% *packet loss* to emulate adverse network conditions.

These conditions were implemented on the client side using `sudo` access traffic control commands, such as `$ tc qdisc add dev enp0s3 root tbf rate 12mbit`.

*Data Collection and Repetition* The client was instrumented to gather detailed performance metrics, including transfer start and end times, total bytes transferred, the number of packets sent and received, and completion rates for datagram-based transfers. To ensure the robustness of the results, each test for a web page version was repeated five times, accounting for system-level variations and ensuring statistical significance.

## 5.9 Emulation: Datagram vs. Stream Comparison

The testing environment was configured with both the client and server running on the same local machine, utilizing local IPv4 connections to communicate with each other directly. This setup ensured that both processes were on the same network and physical hardware, allowing for precise control over the experimental conditions. The network path was simplified, with `0.0.0.0:4433` as a server address meaning that the server is open to connections from any IP address on port 4433.

*Baseline Configuration* To establish a baseline, we first transferred all dummy web objects from the `dummy_files` directory using only reliable QUIC streams at average network speed of 12 Mbps[73]. This step determined the minimum fetch time required for transferring web objects under optimal conditions. We then simulated adverse network conditions by limiting the bandwidth to 9 Mbps and introducing 5% packet loss using `netem` and `tc` commands, reflecting a constrained 4G connection.
Building on the baseline scenario with constrained bandwidth and *packet loss*, we conducted **Five** more scenarios focused on distinguishing between "Essential" and "Non-Essential" elements for each page load on the same constrained network conditions. The scenarios included:

- Fonts as Non-Essential: Additional styling font files were sent via datagrams, while all other objects used reliable streams.
- Ad files and .css as Non-Essential: Ads, analytics and corresponding files were transmitted via datagrams, with all other objects using streams. Dummy files matching the size of average ad files were used for testing, since there is no 'single' file that makes up an ad.
- 3rd Party Objects as Non-Essential: Only essential objects were sent via streams, with additional *.js* scripts, trackers, and extra *.css* scripts sent via datagrams.
- Fonts and Ads as Non-Essential: Both fonts and ad files were transferred using datagrams, with other objects sent through streams.

- Fonts and 3rd Party Objects as Non-Essential: Fonts and 3rd party files were sent via datagrams, while other content used reliable streams.

These scenarios were tested across six different web page versions, repeated five times. From each scenario, we calculated the *mean* and 90th *percentile* of the results to provide a performance evaluation, minimizing the impact of outliers.

## 5.10 Real User Survey

Even with latency metrics, determining which web page content is truly "Non-Essential" is challenging without considering the user's perspective and *Quality of Experience* (QoE). Metrics from our emulation alone do not capture how each element affects the user's experience on the front-end. To understand this impact, it was crucial to balance essential and non-essential elements while preserving the page's structure, appearance, and usability during loading. To capture user preferences, we conducted an online survey (available at [65]), targeting real web users. The survey aimed to gather feedback on how a web page and it's certain elements load, particularly under slow internet conditions.

*5.10.1 Survey Design* Participants were presented with seven variations of the *New York Times* homepage, each simulated under slow and challenging network conditions. These variations included different **delays** applied to elements such as additional *fonts*, *CSS styles*, *third-party content*, and *JavaScript*. Participants were asked to select their preferred version, ranging from the original page load to versions where all 'non-essential' elements were delayed during the initial load. The survey collected their responses through a Google form, featuring recordings of these page load scenarios available on `YouTube` [65]. Participants rated their levels of agreement, preference, or intensity of feeling regarding the different page loads using a *Likert* scale.

To create the custom delayed loading scenarios, we developed a JavaScript for the Chrome browser extension *TamperMonkey*, which allowed us to delay the loading of 'Non-Essential' elements on the test web pages. Additionally, we used Chrome `DevTools` to simulate slow and unfavorable network conditions commonly experienced by internet users, such as slow 3G and 4G speeds and 0-5% packet loss on the browser. The custom *TamperMonkey* script was carefully designed to intercept and delay specific resource requests during the very early stages of page loading, ensuring an accurate simulation of slow network conditions. Executing at `document-start`, the script could apply delays immediately the browser initiated requests for the web page. The script focused on delaying resources like fonts, CSS, and JavaScript by overriding the `XMLHttpRequest` and `fetch APIs`. A `shouldDelay` function was used to check each request against predefined patterns (such as .css, .js, .woff) to determine if it should be delayed. If a match was found, the delay set by us (*120-1000 milliseconds*) was applied. This approach, which intercepted requests directly rather than modifying the `DOM`, ensured that the simulation closely reflected real-world scenarios of slow loading times.

*5.10.2 Web Page Variations* We recorded seven different variations of our test website under the same network conditions based on what we learned from our earlier network request analysis, talked about in 5.6 (5.4.1) [23] to better understand the user's perspective.

These variants of the web page were

(1) **Version 1: Standard Web Page Load**
This version represents the default loading experience of the *New York Times* homepage under slow internet conditions, where all elements (fonts, CSS, JavaScript) loaded without any additional delays.
(2) **Version 2A: Delayed Fonts Loading**
In this variation, additional font files are delayed during the initial page load by *140 ms* from their original load time, simulating a scenario where fonts are de-prioritized to potentially improve other essential elements loading times.
(3) **Version 2B: Extended Font Delay Under Challenging Conditions**
This version extends the delay time for fonts under even more challenging network conditions by increasing the initial delay to *1 second* to showcase the users an even more prominent change in the page loading structure.
(4) **Version 3A: Delayed CSS Elements**
Here, the loading of certain CSS elements and styles is delayed during the initial page load by *182 ms* for every associated request. This version explores how the delay in loading visual styles affects the perceived speed and usability of the page.
(5) **Version 3B: Extended CSS Delay Under Challenging Conditions**
Similar to Version 3A, but with an even greater delay for CSS elements, again at *1 second* delay, this variation tests the limits of CSS prioritization under poor network conditions.
(6) **Version 4A: Delayed Fonts, Non-Essential CSS, and Third-Party JavaScripts**
This version introduces delays to the additional fonts, non-essential CSS styles, and third-party JavaScript elements by *376 ms* during the initial loading phase, aiming to see how a more comprehensive delay strategy impacts user satisfaction.
(7) **Version 4B: Extended Delay of Fonts, CSS, and JavaScript Under More Challenging Conditions**
Building on Version 4A, this variation further extends the delays for fonts, CSS, and JavaScript, particularly under more difficult network conditions, by setting the delay to *1 second*, to understand the cumulative effect of delaying multiple resources.

We simulated poor network conditions for the web page recordings by throttling the connection to slow 4G (*9 Mbps*) and slow 3G (*3 Mbps*) with an additional 5% packet loss during page loading, using the throttling and network manipulation tools available in*Chrome* browser's *DevTools*. This approach provided survey participants with a realistic experience of how the website performs under less-than-ideal conditions that they might encounter in their own usage.

## 6 Result and Evaluation

In this section, we present and discuss the findings of our comprehensive investigation aimed at classifying web content and evaluating the partial reliable transport.

*QUIC Support Analysis* Our comprehensive analysis of the top 100,000 public domains on the internet revealed significant insights into the adoption rates of modern web protocols and addressing schemes. Specifically, we found that only 27% of these domains currently support HTTP/3 and QUIC (Quick UDP Internet Connections). This relatively low adoption rate suggests that the majority of high-traffic websites have yet to leverage the potential benefits of QUIC.

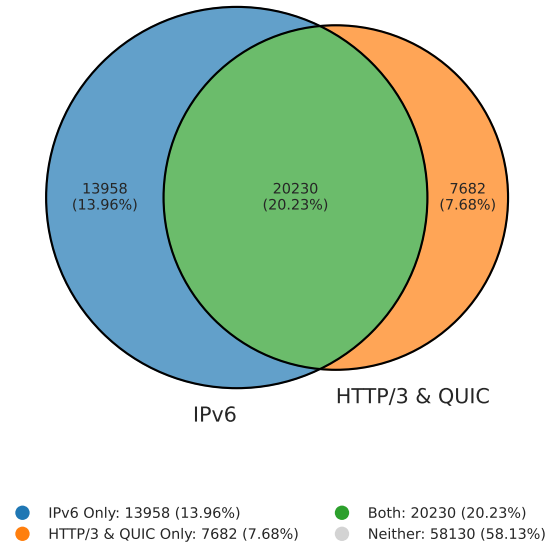



**Figure 12: IPv6 and HTTP/3 & QUIC protocol support among the top 100,000 Web Domains**

Concurrently, our investigation into IPv6 support yielded similar adoption results. Only 34.19% of the analyzed domains currently demonstrate support for IPv6 addressing. This finding is particularly noteworthy given the exhaustion of IPv4 address space and the growing necessity for expanded addressing capabilities to accommodate the proliferation of internet-connected devices.

*QUIC performance evaluation* Our tests of key performance metrics were used to look at the possible benefits of using QUIC instead of TCP. The results of our tests showed that TCP had slightly higher peak throughput in situations with bandwidth capacities of 40 Mbps and 100 Mbps. However, under packet loss conditions, QUIC consistently outperformed TCP. Notably, at a 5% packet loss rate, QUIC achieved a throughput of 22 Mbps, representing an 83.3% improvement over TCP's 12 Mbps.
Additionally, while QUIC's overhead was consistently higher than TCP's across all scenarios, it demonstrated greater stability under increasing packet loss. For example, under 0% packet loss, QUIC exhibited an overhead of approximately 4.0 MB, compared to TCP's 2.1 MB. As packet loss increased to 5%, QUIC's overhead rose slightly to 4.2 MB, whereas TCP's overhead increased more significantly to 3.0 MB. This indicated that although QUIC incurs higher absolute overhead, its rate of increase is more stable than TCP's as packet loss intensifies.
In `bandwidth-delay` scenarios, QUIC consistently outperformed TCP, particularly under conditions of packet loss. With a 40 Mbps bandwidth and 5% packet loss, QUIC achieved a throughput of 22 Mbps, in contrast to TCP's 12 Mbps. At 100 Mbps bandwidth with 0% packet loss, QUIC slightly exceeded TCP, recording a throughput of 95 Mbps compared to TCP's 93 Mbps. Moreover, under 5% packet loss at 100 Mbps, QUIC maintained a significant advantage, achieving 75 Mbps compared to TCP's 55 Mbps.

These results highlight QUIC's superior scalability and resilience across diverse network conditions. However, under extreme latency conditions, such as with a 120 ms delay, QUIC exhibited a slightly longer transfer time, completing the transfer in 30 seconds compared to TCP's 28 seconds, a 7.1% slower performance. This suggests that while QUIC generally excels in throughput and overhead efficiency, TCP may hold a slight edge in high-latency environments.

*Web page replays* We used *bvc-mahimahi* to record and play back web page replay tests over servers that supported TCP (HTTP/2) and QUIC (HTTP/3). These tests showed how well these protocols worked on several key metrics for web page loading, such as *Time to First Byte* (TTFB), *First Contentful Paint* (FCP), *Speed Index* (SI), and *Total Page Load Time* (PLT), as shown in Figure 14. The results highlight QUIC's strengths, particularly under less favorable network conditions.
In a stable network environment with a 12 Mbps bandwidth, QUIC HTTP/3 outperformed TCP HTTP/2 significantly. For instance, TTFB was reduced by 36.6%, dropping from 0.41 seconds with TCP HTTP/2 to just 0.26 seconds with the QUIC server. This improvement is largely thanks to QUIC's 0-RTT connection setup, which skips extra round trips during the TLS handshake, speeding up the initial data transfer. Similarly, QUIC HTTP/3 improved FCP by 16.2%, reducing it from 1.48 seconds with TCP HTTP/2 to 1.24 seconds. This faster delivery of the first visible content can be attributed to QUIC's advanced multiplexing and congestion control, which helped deliver content more efficiently.
The *Speed Index*, measuring how quickly visual content is rendered and ready to use, also improved under QUIC HTTP/3, decreasing by 9.3% from 6.32 seconds to 5.73 seconds related to QUIC's stream prioritization, which helped speed up the rendering of content that a user could see. Additionally, the *Total Page Load Time* (PLT) saw a similar 9.3% reduction under QUIC HTTP/3, from 6.32 seconds to 5.73 seconds. This improvement reflects the cumulative effect of QUIC's faster connection management and data transfer, leading to a quicker overall page load.
However, when network conditions deteriorated, QUIC's advantages were less pronounced. For example, with 5% packet loss at 12 Mbps bandwidth, TTFB increased by 8.8% to 0.38 seconds, although it still outperformed TCP HTTP/2. FCP also increased to 1.72 seconds, and both the Speed Index

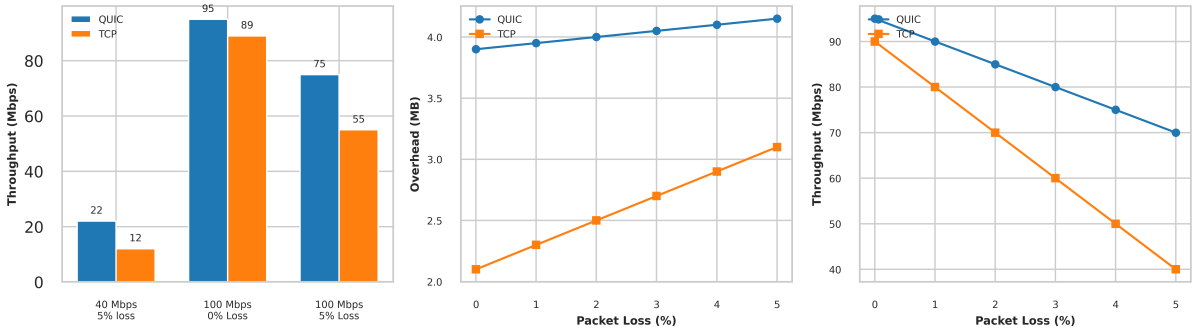



**Figure 13: QUIC vs TCP performance comparison (a) Throughput Comparison under varying conditions, (b) Overhead vs Packet Loss, and (c) Throughput vs Packet Loss.**

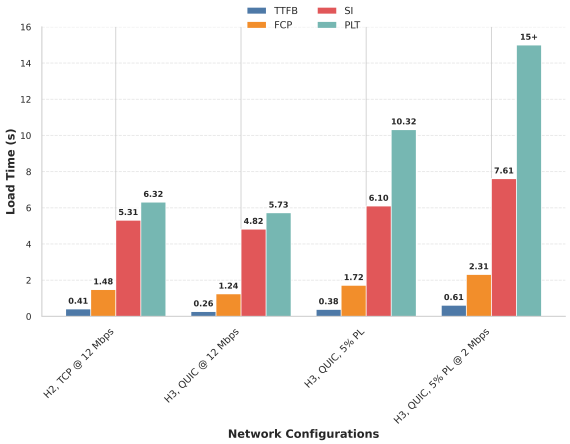



**Figure 14: Web page performance across different network configurations.** *TTFB: Time to First Byte, FCP: First Contentful Paint, SI: Speed Index, PLT: Page Load Time.*
**Note:** PLT for the configuration with QUIC 5% packet loss and 2 Mbps bandwidth exceeds 15 seconds.

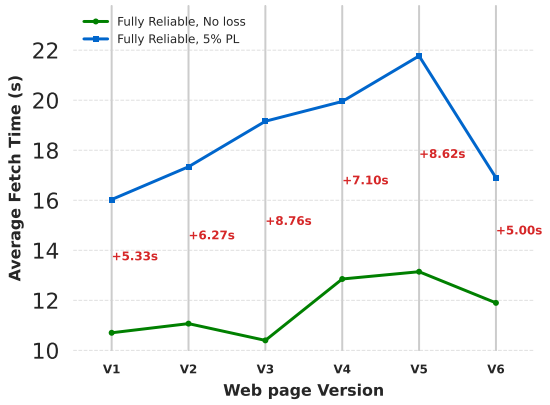



**Figure 15: Emulation - Fully Reliable Transport (Baseline)** *Note- Mean (Avg. 4G): 11.68s, Mean (Slow 4G): 18.53s*

and PLT showed significant slowdowns, rising to 6.10 seconds and 10.32 seconds, respectively. These results suggest that while QUIC is more resilient to packet loss than TCP, performance still degrades noticeably in such conditions.

In even more challenging scenarios, like 5% packet loss combined with reduced bandwidth (2 Mbps), QUIC's performance dropped further. TTFB increased to 0.61 seconds, FCP to 2.31 seconds, SI to 7.61 seconds, and PLT exceeded 15 seconds. These findings indicate that while QUIC maintains some efficiency under difficult conditions, there's still room for improvement in its congestion control and loss recovery mechanisms to better handle severely degraded networks.

## 6.1 PRT results

Our emulation experiments of `nytimes.com` website offered a detailed timing analysis of employing Partially Reliable Transport (PRT) for non-essential web page elements under varying network conditions.

*Baseline Performance in Relation to Network Conditions* We observed that in a stable network environments (e.g., constant bandwidth with 0% packet loss), the performance between Fully Reliable and Partially Reliable transport was nearly identical, with both streams and datagrams exhibiting similar throughput and latency. Under these ideal conditions, QUIC streams' TCP-like reliability mechanisms did not introduce significant overhead, resulting in closely aligned latency and fetch times for both transport modes. However, as network conditions worsened, with reduced bandwidth and increased packet loss (e.g., 9 Mbps with 5% packet loss), the advantages of a partially reliable transport (PRT) became more apparent. In these challenging scenarios, the average fetch time for each page load delivered via Partially Reliable transport was reduced by 99.9 ms, with the 90th percentile showing a reduction of 115.3 ms.

The evaluation of PRT under consistent but constrained network conditions, however, revealed varying levels of effectiveness depending on the type and size of the object files of that web page. For font files, with PRT, the reduction in total fetch times for font files ranged from 98.4 ms to 115.3 ms between Fully Reliable and Partially Reliable transport. These reductions correspond to approximately 0.52%


**(a) Font files** **(b) Ads/CSS files** **(c) Third-party files**

**Figure 16: Comparison of fully reliable vs partially reliable transport for different simulations based on web objects type on a 9 Mbps, 5% packet loss connection.** Note- (a) Mean: 99.2ms, 90th Percentile: 115.3ms; (b) Mean: 113.4ms, 90th Percentile: 132.4ms; (c) Mean: 136.0ms, 90th Percentile: 151.6ms

to 0.60% of the total page load time. This moderate impact, as demonstrated in the 16a graph, indicates that while fonts are often non-essential for the initial rendering of the page, their delivery via datagrams can still contribute very little to reducing overall latency, limited by the relatively small size of these files.

Ad files, which consisted of a mixture of static css, multimedia content, and third-party JavaScript, showed more substantial benefits from PRT. The average reduction in fetch times ranged from 114.1 ms to 132.4 ms, as shown in the 16b graph. These reductions, representing 0.63% to 0.74% of total load time, can be attributed to the size and complexity of ad content, which typically involves multiple HTTP requests and more data size and files.
For third-party JavaScript files and analytics, the application of PRT exhibited the highest variability in performance improvements for single type of content-type tested, with reductions compared to fully reliable transport ranging from 136.0 ms to 151.6 ms reduction in average fetch times between the two. This variability is depicted in the 16c graph.

*Cumulative Effects and Strategic PRT Application* The more significant latency reductions were observed when multiple non-essential elements were grouped and transmitted via datagrams for a web page, especially under conditions simulating a bursty, lossy network. For instance, in scenarios where fonts, third-party JavaScript, and CSS were transmitted together on the PRT transport, the cumulative fetch time reduction averaged 243.7 ms, with a peak reduction of 291.5 ms at the 90th *percentile* compared to fully reliable transport. These results, as shown in the 18b graph, indicate that the strategic application of PRT to a combination of non-essential elements yields more substantial timing differences as the data size increases.

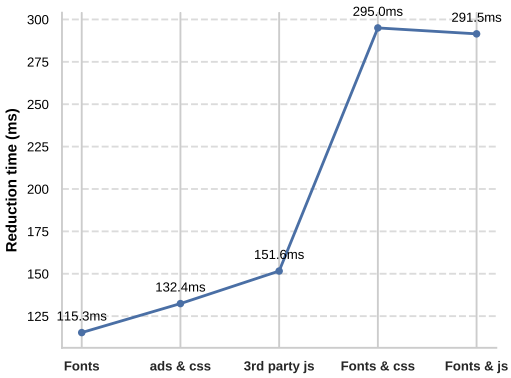
However, it is critical to consider the trade-offs involved. The 17b graph reveals that as the complexity and size of the grouped content increase, so does the risk of data loss. For example, when fonts and javascripts were grouped, the data loss was close to 9%. This suggests that while PRT can reduce latency, it also loses data to gain latency.

*6.1.1 User Survey Results* The survey involved real internet users as participants and offered insights into how users prioritize web page elements and their tolerance for delayed loading of non-essential content. The survey sample primarily consisted of frequent internet users, with 70% of respondents aged 25-34 and 30% aged 18-24. All respondents to the survey indicated that they use the internet **several times a day**, highlighting their strong familiarity with digital experiences. This makes their feedback especially valuable for understanding user behavior under different network conditions.

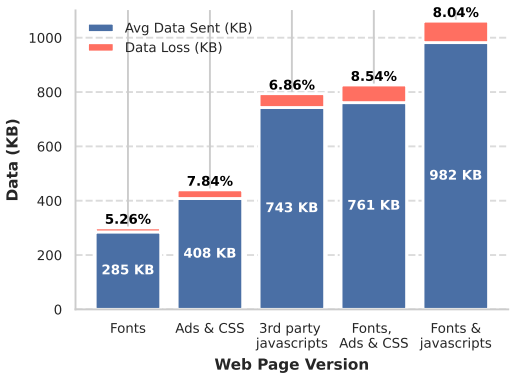
*Perception of Non-Essential Elements* When evaluating Version 1 of the web page, which included all content elements, 60% of respondents agreed or strongly agreed that some elements could be considered non-essential for the initial page load. This suggests a general openness among users to loading strategies that prioritize essential content over non-essential elements, especially in scenarios where network conditions are less than ideal. The feedback indicates that users are willing to accept a streamlined initial load if it means faster access to the most critical content.

*Font and Style Preferences* The importance of designer-chosen fonts and additional style elements varied among participants. Half of the respondents indicated that designer fonts were of low importance, expressing a preference for faster loading times, with the option to load fonts later as connectivity improved. Another 30% viewed fonts as somewhat important but were willing to compromise on them in favor of speed when necessary. The remaining 20% considered designer fonts to be unimportant to their overall reading experience.
Similarly, opinions on additional style elements, such as custom layouts, animations, and color schemes, leaned toward prioritizing speed. Sixty percent of respondents prioritized quick loading over these visual enhancements, while 20% considered them important but not essential. Another 20% found these elements to be of little importance.

*Third-Party JavaScript Functionalities* Responses regarding third-party JavaScript functionalities were more varied, reflecting diverse user priorities. Half of the participants viewed these scripts as somewhat important, favoring a balance between quick load times and the potential for added functionalities as connectivity improved. In contrast, 30% of

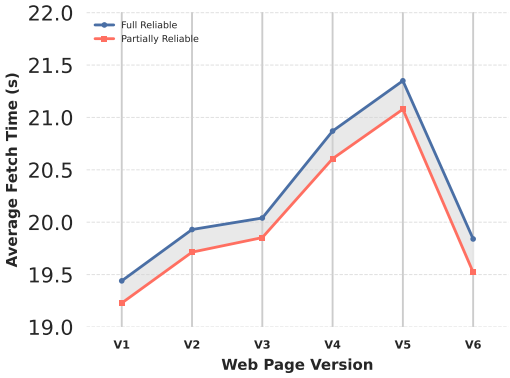



**(a) Testing scenarios based on content-type and their 90th percentile latency benefit in average fetch times of each web page variation.**

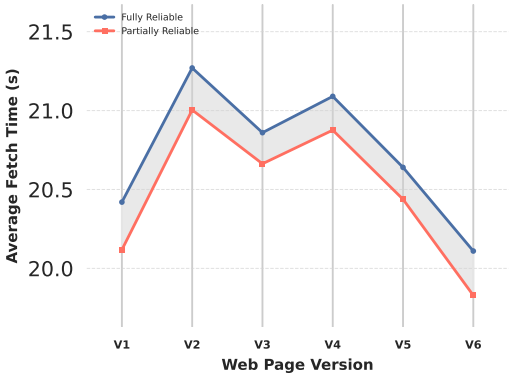



**(b) Avg. Data sent vs Data lost via QUIC datagrams transmission under slow network condition and 5% packet loss.**

**Figure 17: Comparison of latency benefits and data loss in different scenarios**




**(a) Fonts, ads, and additional CSS**




**(b) Fonts, JavaScript, and additional CSS**

**Figure 18: Comparison of Reliable vs Partially Reliable Transport for combined file types on a 9 Mbps, 5% packet loss connection.** Note- (a) Mean: 244.8ms, 90th Percentile: 295.0 ms; (b) Mean: 243.7ms, 90th Percentile: 291.5ms

respondents deemed third-party scripts unimportant, emphasizing the need for speed over additional interactive features. However, 10% of participants considered these scripts crucial, indicating a willingness to accept slower load times for the sake of full functionality.

*User Experience with Alternative Versions* The survey also asked for user preferences across different versions of the web page, each offering varying levels of optimization. When comparing Version 2, which incorporated font delay, to Version 1 (original page load), 40% of participants expressed a preference for Version 2, while 30% favored Version 1, and another 30% reported no significant preference. In the comparison between Version 3 (Delayed css) and Version 1, 50% of respondents preferred Version 1, 30% preferred Version 3, and 20% were indifferent.

Notably, Version 4, which involved delayed loading of non-essential elements such as fonts, CSS, and third-party JavaScripts, was well-received. Sixty percent of respondents

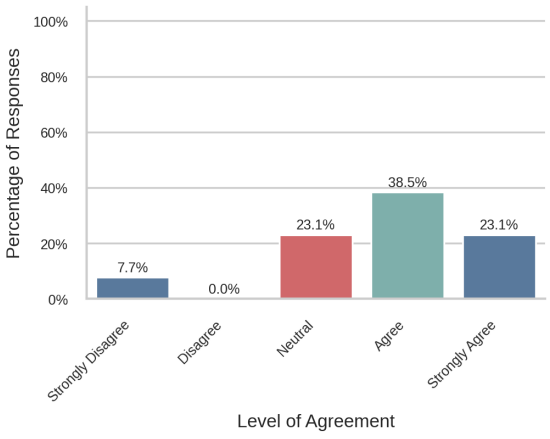



**Figure 19: Survey Responses for "Version 4 (delayed fonts, CSS, and JavaScripts) provides a satisfactory user experience".**

agreed or strongly agreed that this version provided a satisfactory user experience, while 23% were neutral, as shown in 19. These findings suggest that users generally appreciate faster loading times but their preferences may vary depending on the specific trade-offs between speed and functionality.

Overall, the survey results indicate a clear preference for faster loading times, with users willing to compromise on non-essential elements like custom fonts and additional styles when network conditions are poor. However, the mixed responses regarding third-party functionalities suggest that these features remain important to a significant portion of users, even at the cost of slower load times. The positive reception of Version 4, where non-essential content was delayed, suggests that such strategies can effectively enhance user experience in environments with constrained network conditions.

## 7 Discussion

Overall, our study provided us good insights about the benefits of QUIC over TCP, how to group web objects by how "essential" they are, and how Partially Reliable Transport (PRT) and QUIC could be used to improve the delivery of web content, especially when network conditions are bad. However, several limitations and considerations must be recognized, which moderate the enthusiasm for these approaches and underscore the need for further research.
While our use of website emulation tools like MahiMahi provided a controlled environment for testing, it is important to acknowledge the limitations of these tools in capturing the complexities of real-world internet usage. For instance, our tests did not account for the variability in user behavior, network congestion, or optimisations by web browsers and content delivery networks (CDNs) that can significantly affect web page performance per load. In real life, web elements may be served from various CDN caching layers, resulting in different latency times. Additionally, browsers may use different dependency graphs for optimization, which can change the performance outcomes [47]. Therefore, while our results show promise, they need to be validated in much more real-world settings to ensure their *generalizability*.
Our application of server-client for partially reliable transport although showed potential in reducing load times in *milliseconds* by bypassing guaranteed delivery for non-essential content, but this approach is not without its challenges. The major trade-offs associated with our PRT approach is the risk of incomplete file delivery leading to broken or poorly rendered web pages, highlight the limitations of current web systems in handling partially delivered content. While the observed latency reductions—particularly at the 90th *percentile*—are encouraging, this often comes at the cost of increased data loss. In this experiment setup, we deliberately avoided simulating the rendering aspects of web objects due to their fundamental nature. Unlike videos or images, which can tolerate some level of quality degradation under challenging conditions—such as dropping frames or pixels—web object files (e.g., *HTML, CSS, JS, WOFF/WOFF2*) **lack** this flexibility. These files must be *fully intact* to function correctly, and any packet loss during transfer can render them *corrupted* or *incomplete*. In the absence of re-transmission mechanisms, as is the case with QUIC datagrams, incomplete files could lead to unpredictable behavior or even cause the web page to break. This could significantly affect the website's CLS score, further degrading the user experience. To prevent such issues, browsers would require *additional intelligence* at the application layer to identify incomplete objects and avoid rendering them.

Furthermore, our user survey revealed that while participants generally favored faster loading times, there was a tolerance for delayed loading of non-essential content, such as fonts and additional style elements. However, the survey's relatively small sample size and focus on a single website through a video limits the ability to apply these results more broadly. Moreover, the simulated network conditions used in the survey may not fully capture the range of user experiences in real-world scenarios.
Additionally, while our study demonstrates the potential benefits of QUIC and PRT for improving web content delivery, the findings are context-dependent and may not fully translate to the diverse and unpredictable conditions of everyday internet usage. The small sample of websites analyzed may not represent the full spectrum of web pages, and the controlled testing environments do not account for all the variable factors that can be encountered in real-world networks. Future research should focus on expanding the scope of testing to include a broader array of websites and more diverse network conditions. Additionally, developing adaptive algorithms that dynamically switch between reliable and partially reliable transport based on real-time network conditions and content criticality could further enhance web performance.

## 8 Related Work

This section examines the previous research conducted with a similar objective to ours, which is to enhance the performance of web pages through various optimization techniques.

### 8.1 Dependency Graph Optimization

Dependency graph optimization techniques have become common and valuable tools for improving web performance at the application layer. These methods utilize the complex connections between web page resources to reduce loading times and enhance user experience.

*8.1.1 Fine-grained Dependency Tracking* Modern times' web pages are composed of multiple interconnected resources, such as *HTML, CSS, JavaScript, images, fonts* and other media. Conventional browsers like *Chrome, Firefox* tend to make cautious assumptions about these interdependencies, which frequently results in less than optimal loading sequences. Advanced dependency tracking systems such as *WProf* [71] and Polaris [44] utilize a detailed analysis to create more precise dependency graphs. *WProf* decomposes the process of loading a page into individual and fundamental units known as *activities*. These include parsing individual

HTML tags, loading single objects, evaluating JavaScript or CSS, and rendering *DOM* updates.
For Dependency Types, the paper *Polaris* offers the idea of using multiple categories of dependencies are identified such as *Flow* dependencies (e.g., loading an object depends on parsing its referencing tag), *Output* dependencies (e.g., JavaScript execution may block HTML parsing to avoid DOM conflicts), *Lazy/Eager binding* (e.g., preloading strategies), and *Resource constraints* (e.g., limitations on concurrent TCP connections)[44].
For Browser-specific Policies, they claim different browsers implement varying dependency policies. For example, Internet Explorer enforces stricter CSS evaluation dependencies compared to *WebKit*-based browsers[71].

*8.1.2 Optimizing Loading Strategies* Using the detailed dependency information, these systems can implement advanced loading strategies. By applying graph algorithms to the dependency structure, the critical path of the page load can be identified. This allows prioritization of resources that directly impact the overall load time. The authors of [71] state that with a comprehensive understanding of true dependencies, more resources can be loaded in parallel without violating correctness. Moreover, for resource prioritization, Network and computational resources can be allocated more efficiently based on the dependency structure, ensuring critical resources are processed first [71].

*8.1.3 Results and Challenges* The studies [71][44] have shown significant performance improvements using these techniques. *Polaris* demonstrated a 34% reduction in page load times at the median, and 59% at the 95th *percentile* across various network conditions [44]. While *WProf*'s research showed that computation (HTML parsing and JavaScript execution) makes up 35% of the critical path in most page loads, this shows how important it is to optimize both network and computation aspects [71].
While these solutions look promising, implementing these optimization techniques presents several challenges.Having deep integration with browser internals is often required to accurately track dependencies and implement optimized loading strategies. The analysis and optimization processes must be lightweight to avoid introducing significant overhead. Furthermore, Modern web applications with highly dynamic content complicate static dependency analysis requires runtime adaptation [44].

## 8.2 Partial Reliability Solutions

*8.2.1 WebTransport* Building upon QUIC, the *WebTransport* API has emerged as a powerful tool for web applications, leveraging QUIC and HTTP/3 to provide both datagram and stream-based communication by *Google*. This flexibility allows developers to choose the most appropriate communication model for their specific use case, potentially improving performance over traditional *WebSockets* in many scenarios [42]. It also allows *bi-directional* streams for reliable, ordered data transmission similar to TCP but without head-of-line blocking at the connection level. It has the support for unidirectional Streams as well which enables efficient one-way communication for scenarios like server push or telemetry data needed for simpler web applications. It also enables Datagrams support which provides a mechanism for sending unreliable, unordered data, suitable for real-time applications like gaming or live video streaming. While promising, it faces usage and adoptation support at browser level[42] Currently, only three out of countless browsers seem to offer support for *WebTransport*.

*8.2.2 VOXEL* In the area of multimedia streaming, the VOXEL system [46] represents a notable advancement, integrating DASH-like adaptive streaming with QUIC's partial reliability features. By combining *adaptive bitrate streaming* techniques with QUIC's features, VOXEL demonstrates significant improvements in quality of experience metrics, especially in adverse network environments. VOXEL employs a clever offline algorithm to rank video frames within each segment based on their impact on quality, allowing for intelligent *partial reliability* decisions[46]. Along with that, it brings an enhanced *ABR algorithm*, that considers frame drop tolerance and partial reliability to optimize end-user QoE directly. Through the modified version of QUIC that supports partial reliability, it allows for selective dropping of less important frames during network congestion[46].

## 8.3 Proxy-Based Solutions

Additionally, various Proxy-based solutions have been extensively explored to enhance time-sensitive data transfer and optimize a web page's content delivery across diverse network conditions. These solutions can be categorized into *application-layer* and *transport-layer* intermediaries, each offering unique advantages and optimizations.

*8.3.1 Application-Layer Proxies* In Application-layer proxies, also known as **Performance-enhancing Proxies** (PEPs), operate at the highest layer of the OSI model, intercepting and manipulating application-level protocols such as HTTP. These proxies can implement various optimization techniques such as
**Protocol Acceleration**: PEPs can implement protocol-specific optimizations. For instance, SPDY over satellite leverages SPDY's *multiplexing* and *header compression* features to mitigate the impact of high latency in satellite networks [56].
**Content Optimization**: Application-layer proxies can perform content-aware optimizations such as *image compression*, *text minification*, and *adaptive bitrate streaming* for video content.
**Caching and Prefetching**: These proxies can implement intelligent caching mechanisms and predictive prefetching to reduce latency and bandwidth consumption.
**Request Coalescing**: Multiple small requests can be combined into larger, more efficient transfers, reducing the overhead of multiple round trips.
However, application-layer proxies face challenges with end-to-end encryption, as they require access to unencrypted application data to perform optimizations.

*8.3.2 Transport-Layer Proxies* In Transport-layer proxies which operate at a lower level, intercepting and manipulating transport protocol traffic solutions seem to offer several advantages.

**Protocol-Specific Optimizations**: Solutions like *Sidecar* can implement protocol-specific enhancements without modifying end-host stacks. For TCP, this might include adjusting congestion control parameters or implementing selective acknowledgments (SACK) [77].
**Split Connections**: Transport-layer proxies can terminate connections on either side, allowing for optimized protocol parameters tailored to each network segment's characteristics.
**Cross-Layer Optimizations**: By operating at the transport layer, these proxies can leverage information from both network and transport layers to make informed decisions about traffic optimization.

*CDN and SDN Integration* Recent work in Content Delivery Networks (CDNs) has explored more sophisticated proxy-based solutions that leverage *Software-Defined Networking* (SDN) principles.
**CDN- and Protocol-Aware Steering:** *CP-Steering* is a new content steering solution that improves adaptive video streaming by taking into account both CDN features and transport protocol features, especially QUIC [15]. This approach dynamically selects the most appropriate CDN and protocol configuration based on real-time network conditions and content requirements [15]. Mukerjee et al. in [78] propose an SDN-based CDN architecture that allows for fine-grained control over content routing and delivery. This approach enables *Dynamic load balancing* across CDN nodes, *Protocol-aware traffic* engineering, and *Adaptive content placement* strategies.

## 8.4 Client-side Optimizations

Client-side optimization techniques have also seen significant advancements, leveraging sophisticated analysis methods, protocol-specific enhancements, and machine learning approaches to improve web performance.

*8.4.1 Causal Profiling and What-If Analysis* Wang et al. in [71] introduced a causal profiling technique for web page load time analysis. This approach constructs a fine-grained dependency graph of the page load process, capturing both network and computation activities. It employs counterfactual analysis to estimate the impact of potential optimizations without actually implementing them. This provides developers with actionable insights by identifying critical paths and bottlenecks in the page load process. The technique uses a modified *Chromium* browser to collect detailed timing information and constructs a causal model that accounts for both deterministic and probabilistic dependencies between page load activities[71].
**Fine-grained HTTP/3 Prioritization**: Wong et al. [75] proposed a reinforcement learning approach for HTTP/3 stream prioritization. Their system uses a *deep Q-network* (DQN) to learn optimal prioritization strategies. It considers factors such as resource type, size, and dependency relationships. Additionally, it adapts to different network conditions and page structures in real-time. The RL agent is trained on a diverse set of web pages and network conditions, learning to make prioritization decisions that minimize page load time and improve key performance metrics like *First Contentful Paint* (FCP) and *Time to Interactive* (TTI)[75].

*8.4.2 Heuristic-based Resource Fetching* CASPR [58] represents a class of heuristic-based algorithms that operate within the browser's networking constraints. CASPR predicts which resources will be needed based on historical data and current page structure then the resources are assigned priorities based on their predicted impact on page load time. It's algorithm applies *Constraint-aware Scheduling* by respecting browser limitations on concurrent connections and per-host connection limits. Additionally, CASPR adjusts its predictions and priorities as the page load progresses, adapting to actual resource requirements. As a result, the algorithm demonstrates significant improvements in page load times, especially for complex web applications with numerous interdependent resources [58].

*8.4.3 Machine Learning Integration* The integration of machine learning techniques with web performance optimization is an emerging trend. **Adaptive Bitrate Streaming over QUIC** is one example, as proposed by Mao et al. in [3], where a reinforcement learning approach is utilized for adaptive bitrate (ABR) streaming over QUIC. Their system employs a *deep deterministic policy gradient* (DDPG) algorithm to learn optimal bitrate selection policies. This approach takes into account QUIC-specific metrics such as *stream states* and congestion control parameters, allowing the system to adapt to network variations and user preferences in real-time [3].

## 8.5 Server-side Optimizations

Server-side optimizations play a crucial role in enhancing web page performance by reducing server response time, minimizing resource usage, and efficiently handling client requests. Recent research and industry practices have focused on several key areas. Among various server-side optimizations, two stand out for their significant impact and common implementation for web page performance.

*8.5.1 Caching Mechanisms* Caching is a critical optimization that can dramatically reduce server load and improve response times [9].
**Multi-level Caching**: Implementing a hierarchical caching system that combines:
- Browser-side caching for repeat visitors
- Server-side caching to serve content closer to users
- Content Delivery Networks (CDNs) for static assets
**Intelligent Cache Control**: Developing sophisticated strategies to manage cached content. They use HTTP *Cache-Control headers* to specify caching behavior and implement *time-to-live* (TTL) values based on content volatility. Also employs *cache invalidation* techniques to ensure content freshness[9].

**Edge Caching**: Utilizing CDNs to cache content closer to end-users and reducing latency by serving content from geographically distributed edge locations. This improves content availability and provides redundancy[24]. Effective caching can significantly reduce server response times, minimize database queries, and improve overall page load times.

As stated by Website Magazine, "Caching can help reduce load time because it retrieves content closer to the user's location. By retrieving the content more efficiently, caching reduces the overall latency in the round-trip delivery time" [9].

*8.5.2 Asynchronous Processing and Event-Driven Architectures* Leveraging asynchronous processing seems to greatly improve server responsiveness and scalability [19] by implementing event-driven, non-blocking I/O models and using technologies like *Node.js*'s event loop handling a large number of concurrent connections efficiently without spawning additional threads. By utilizing asynchronous loading for scripts and resources and using the `async` and `defer` attributes for script loading, improvement can be seen in page responsiveness and reducing render-blocking resources [19].

## 8.6 Protocol-level Optimizations

Advancements in transport protocols, particularly QUIC, have led to significant protocol-level optimizations aimed at improving performance, reliability, and flexibility. These optimizations range from modifications to the core protocol to extensions and enhancements that cater to specific use cases.

*8.6.1 QUIC Performance Enhancements* QUIC implementations have focused on optimizing various aspects of the protocol:

**Packet Size Optimization**: Researchers have shown that increasing QUIC packet sizes can lead to substantial performance gains. For instance, the *Litespeed* implementation demonstrated a performance increase from 400Mbps to over 600Mbps by increasing packet size from 1280 to 4096 bytes [54]. This optimization leverages DPLPMTUD (*Datagram Packetization Layer Path MTU Discovery*) to safely increase packet sizes.

However, it is also mentioned that companies with their own QUIC implementation, like *Cloudflare* and *Microsoft*, do not make their performance dependent on altering the packet size. One reason is that the maximum packet size that the receiver—a C*hromium browser*, specifically—can handle, which is reportedly 1476 bytes, limits the size of the packet even with the use of DPLPMTUD. Instead, they depend on transmitting *packet trains* and minimizing the expense of utilizing the UDP socket API. It becomes much less important how big individual packets are once the implementation starts using *packet trains* [54].

**UDP Socket API Improvements**: An important constraint in the performance of QUIC was found in the socket layer. The study [54] found that the socket layer, specifically the functions `sendto, recvfrom, sendmsg, and recvmsg` accounted for 70 to 80% of the CPU load. They advise that by using sendmmsg and GSO (*Generic Segmentation Offloading*) to send multiple packet trains in a single call and implementing `sendmsg` with GSO, we can achieve better efficient packet transmission. This process is part of fine-tuning the interaction between congestion control, flow control, pacing, and packet train formation [54].

*8.6.2 More Protocol Modifications and Extensions* **uQUIC (Unreliable QUIC)**: This change puts decreasing time delay ahead of reliability for some types of traffic. *uQUIC* adds an unreliable data transmission mode that can be set up within QUIC. This lets applications choose when to sacrifice reliability for lower latency [30].

**FlEC (Flexible Error Correction)**: FlEC on the other hand, enhances QUIC with application-tailored reliability mechanisms. It introduces a flexible error correction framework that allows applications to specify their reliability requirements dynamically [41]. FlEC has shown improved performance for various use cases by reducing re-transmission overhead, using delay-constrained messaging, which minimizes tail latency.

**QUIC Delayed ACK Extension**: This extension fine-tunes transport layer behavior by potentially reducing ACK traffic by up to 50% through delayed acknowledgments. The `maximum ACK delay transport` parameter defines the time the receiver can delay sending acknowledgments, balancing between reducing ACK traffic and providing timely feedback [49].

# 9 Conclusion

In conclusion, this thesis has highlighted the significant potential of QUIC and partial reliability transport (PRT) to improve web content delivery, particularly under challenging network conditions. While TCP showed slight advantages in stable environments, QUIC excelled in scenarios with packet loss and high latency, making it a more resilient and scalable option for modern web applications. Even with these advantages, our results indicate a low current adoption rate of QUIC and HTTP/3 suggesting there is significant potential for wider implementation. Our research into partial reliability showed modest promise in reducing latency for web content delivery, although it comes with the trade-off of potential data. The effectiveness of partially reliable transport varied by content and network conditions, emphasizes the need for careful implementation to balance performance improvements with potential impact on the user experience.

Furthermore, the user survey carried out for this study offered crucial understanding into user preferences, showing a widespread willingness to prioritize critical content for quicker initial loading, especially under poor network conditions. The feedback and preference on delayed loading of 'Non-Essential' elements web page versions highlights the possibility of these methods improving a user's experience in scenarios with constrained network performance.

Overall, QUIC and its partial reliability are technologies that can be crucial in determining the future of resilient and efficient content delivery for web applications. As the internet continues to evolve and users increasingly demand fast and reliable web experiences, the careful deployment of these technologies, influenced by user needs and technical strengths, could significantly boost web performance across diverse network conditions.

# Bibliography

[1] [n.d.]. mitmproxy - an interactive HTTPS proxy. https://mitmproxy.org/.

[2] 2024. WebPageTest - Website Performance and Optimization Test. https://www.webpagetest.org/. Accessed: 2024-08-05.

[3] Sultan Almuhammadi et al. 2022. QUIC Network Traffic Classification Using Ensemble Machine Learning Techniques. *Applied Sciences* 13, 8 (2022), 4725. https://doi.org/10.3390/app13084725

[4] M. Belshe, R. Peon, and M. Thomson. 2015. *Hypertext Transfer Protocol Version 2 (HTTP/2).* RFC 7540. RFC Editor. http://www.rfc-editor.org/rfc/rfc7540.txt

[5] T. Berners-Lee, R. Fielding, and H. Frystyk. 1996. *Hypertext Transfer Protocol – HTTP/1.0.* RFC 1945. RFC Editor. https://doi.org/10.17487/RFC1945

[6] bilibili. n.d.. mahimahi. https://github.com/bilibili/mahimahi

[7] M. Bishop. 2022. *HTTP/3.* RFC 9114. RFC Editor. http://www.rfc-editor.org/rfc/rfc9114.txt

[8] Michael Butkiewicz, Harsha V Madhyastha, and Vyas Sekar. 2011. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. 313–328.

[9] CacheFly Team. 2023. Understanding and Utilizing Caching for Improved Web Performance. *Website Magazine* (October 2023). https://www.cachefly.com/news/understanding-and-utilizing-caching-for-improved-web-performance/

[10] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. 2016. BBR: Congestion-based congestion control. In *Communications of the ACM*, Vol. 60. ACM, 58–66.

[11] Yung-Chih Chen, Yeon sup Lim, and Richard J. Gibbens. 2021. A Measurement-based Study of MultiPath TCP Performance over Wireless Networks. In *Proceedings of the 6th International Conference, ICICCT* (New Delhi, India).

[12] Cloudflare. [n.d.]. quiche: Savoury implementation of the QUIC transport protocol and HTTP/3. https://github.com/cloudflare/quiche

[13] L. Donckers, P. J. M. Havinga, and G. J. M. Smit. 2002. Enhancing Energy Efficient TCP by Partial Reliability. In *13th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*. IEEE.

[14] Exploding Topics. 2024. Internet Traffic from Mobile Devices (Aug 2024). https://explodingtopics.com/blog/mobile-internet-traffic Accessed on August 14, 2024.

[15] Reza Farahani, Abdelhak Bentaleb, Mohammad Shojafar, and Hermann Hellwagner. 2023. CP-Steering: CDN- and Protocol-Aware Content Steering Solution for HTTP Adaptive Video Streaming. https://doi.org/10.1145/3588444.3591044

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. 1999. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 2616. RFC Editor. http://www.rfc-editor.org/rfc/rfc2616.txt

[17] R. Fielding and J. Reschke. 2014. *Hypertext Transfer Protocol (HTTP/1.1).* RFC 7230-7235. IETF. https://www.rfc-editor.org/info/rfc7230 RFC 7230-7235 (HTTP/1.1) obsolete RFC 2616.

[18] Ned Freed and Dr. Nathaniel S. Borenstein. 1996. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046. https://doi.org/10.17487/RFC2046

[19] GeeksforGeeks. 2023. Asynchronous Processing and Event-Driven Architectures. https://www.geeksforgeeks.org/

[20] Google. 2024. PageSpeed Insights. developers.google.com/speed/pagespeed/insights/

[21] Ilya Grigorik. 2020. Constructing the Object Model. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model.

[22] Karl-Johan Grinnemo and Anna Brunstrom. 2001. Evaluation of the QoS offered by PRTP-ECN - A TCP Compliant Partially Reliable Transport Protocol. In *9th International Workshop on QoS (IWQoS)*. Karlsruhe, Germany, 217–231.

[23] Dennis Guse, Sebastian Schuck, Oliver Hohlfeld, Alexander Raake, and Sebastian Möller. 2015. Subjective quality of webpage loading: The impact of delayed and missing elements on quality ratings and task completion time. *2015 Seventh International Workshop on Quality of Multimedia Experience (QoMEX)* (2015), 1–6. https://api.semanticscholar.org/CorpusID:39044384

[24] HarperDB. 2023. Edge Caching Explained Why You Should Be Using It. https://www.harperdb.io/post/edge-caching-explained-why-you-should-be-using-it

[25] HTTP/1.1 specification. 1999. HTTP/1.1. https://http.dev/1.1.

[26] http.dev. 2023. HTTP/0.9 explained. https://http.dev/0.9

[27] Jana Iyengar and Ian Swett. 2021. *QUIC Loss Detection and Congestion Control.* Technical Report 9002. https://doi.org/10.17487/RFC9002

[28] J. Iyengar and M. Thomson. 2021. *QUIC: A UDP-Based Multiplexed and Secure Transport.* RFC 9000. RFC Editor. http://www.rfc-editor.org/rfc/rfc9000.txt

[29] Arash Molavi Kakhki, Samuel Jero, David Choffnes, Cristina Nita-Rotaru, and Alan Mislove. 2017. Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols. In *Proceedings of the 2017 Internet Measurement Conference*. 290–303.

[30] Madhan Kanagarathinam, Syed Faraz Hasan, Sujith Rengan, Sukhdeep Singh, Gunjan Kumar Choudhary, Nawab Muhammad Faseeh Qureshi, and Hoejoo Lee. 2022. Enhanced QUIC Protocol for transferring Time-Sensitive Data. In *2022 IEEE International Conference on Communications Workshops (ICC Workshops)*. 1–6. https://doi.org/10.1109/ICCWorkshops53468.2022.9882167

[31] Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. 2013. MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution. *IEEE/ACM Transactions on Networking* (2013), 15. https://doi.org/10.1109/TNET.2013.2274462

[32] Ramin Khalili, Nicolas Gast, Miroslav Popovic, Utkarsh Upadhyay, and Jean-Yves Le Boudec. 2013. OLIA: A Unified Congestion Control Algorithm for MPTCP. In *Proceedings of the ACM SIGMETRICS/international conference on Measurement and modeling of computer systems*. ACM, 411–412.

[33] E. Kohler, M. Handley, and S. Floyd. 2006. Datagram Congestion Control Protocol (DCCP). RFC 4340. https://doi.org/10.17487/RFC4340

[34] C. Krasic, M. Bishop, and A. Frindell. 2022. *QPACK: Field Compression for HTTP/3.* RFC 9204. IETF. https://doi.org/10.17487/RFC9204

[35] Zsolt Krämer, Felicián Németh, Attila Mihály, Sándor Molnár, István Pelle, Gergely Pongrácz, and Donát Scharnitzky. 2022. On the Potential of MP-QUIC as Transport Layer Aggregator for Multiple Cellular Networks. *Electronics* 11, 9 (2022). https://doi.org/10.3390/electronics11091492

[36] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, and Charles Krasic. 2017. The QUIC Transport Protocol: Design and Internet-Scale Deployment. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM)*. ACM.

[37] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The QUIC transport protocol: Design and Internet-scale deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. 183–196.

[38] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. https://doi.org/10.14722/ndss.2019.23386

[39] Yao-Nan Lien and Ming-Han Wu. 2002. Partial Reliable TCP. *Computer Science Department, National Chengchi University, Taipei, Taiwan* (2002). http://www.cs.nccu.edu.tw/~lien/Pub/c72minghan.pdf

[40] Olivier Mehani, Ralph Holz, Simone Ferlin, and Roksana Boreli. 2015. An Early Look at Multipath TCP Deployment in the Wild. In *Proceedings of the 6th International Workshop on Hot Topics in Planet-Scale Measurement (HotPlanet)*. ACM, 7–12. https://www.simula.no/publications/early-look-multipath-tcp-deployment-wild

[41] François Michel, Alejandro Cohen, Derya Malak, Quentin De Coninck, Muriel Médard, and Olivier Bonaventure. 2023. FlEC: Enhancing QUIC With Application-Tailored Reliability Mechanisms. *IEEE/ACM Transactions on Networking* 31, 2 (2023), 606–619. https://doi.org/10.1109/TNET.2022.3195611

[42] Mozilla Developer Network. 2024. WebTransport API. https://developer.mozilla.org/en-US/docs/Web/API/WebTransport_API

[43] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The Cost of the "S" in HTTPS. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 133–140.

[44] Ravi Netravali, Ameesh Goyal, James Mickens, and Hari Balakrishnan. 2016. Polaris: Faster Page Loads Using Fine-grained Dependency Tracking. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/netravali

[45] Ravi Netravali, Anirudh Sivaraman, Somak Das, Ameesh Goyal, Keith Winstein, James Mickens, and Hari Balakrishnan. 2015. Mahimahi: accurate record-and-replay for HTTP. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 417–429.

[46] Mirko Palmer, Malte Appel, Kevin Spiteri, Balakrishnan Chandrasekaran, Anja Feldmann, and Ramesh K. Sitaraman. 2021. VOXEL: cross-layer optimization for video streaming with imperfect transmission. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 359–374. https://doi.org/10.1145/3485983.3494864

[47] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante. 2017. De-ossifying the Internet transport layer: A survey and future perspectives. *IEEE Communications Surveys and Tutorials* 19, 1 (Feb 2017), 619–639.

[48] T. Pauly, E. Kinnear, and D. Schinazi. 2022. *An Unreliable Datagram Extension to QUIC.* RFC 9221. RFC Editor. http://www.rfc-editor.org/

rfc/rfc9221.txt
[49] Anna Pavlidis, Jana Iyengar, and Nicolas Kuhn. 2022. Reducing the acknowledgement frequency in IETF QUIC. *International Journal of Satellite Communications and Networking* (10 2022). https://doi.org/10.1002/sat.1487
[50] R. Peon and H. Ruellan. 2015. *HPACK: Header Compression for HTTP/2*. RFC 7541. RFC Editor. http://www.rfc-editor.org/rfc/rfc7541.txt
[51] Michele Polese, Federico Chiariotti, Elia Bonetto, Filippo Rigotto, Andrea Zanella, and Michele Zorzi. 2019. A Survey on Recent Advances in Transport Layer Protocols. *IEEE Communications Surveys and Tutorials* (2019). https://doi.org/10.1109/COMST.2019.2932905
[52] J. Postel. 1980. User Datagram Protocol. RFC 768. https://doi.org/10.17487/RFC0768
[53] J. Postel. 1981. Transmission Control Protocol. RFC 793. https://doi.org/10.17487/RFC0793
[54] Private Octopus Inc. 2023. QUIC Performance. https://www.privateoctopus.com/2023/12/12/quic-performance.html
[55] QUIC Working Group. [n.d.]. QUIC Interop Runner. https://github.com/quic-interop/quic-interop-runner.
[56] Cesare Roseti, Ahmed Abdel Salam, Michele Luglio, and Francesco Zampognaro. 2015. SPDY over satellite: Performance optimization through an end-to-end technology. In *2015 38th International Conference on Telecommunications and Signal Processing (TSP)*. 1–6. https://doi.org/10.1109/TSP.2015.7296430
[57] Jim Roskind. 2013. *QUIC: Quick UDP Internet Connections*. Technical Report. Google.
[58] Vaspol Ruamviboonsuk. 2020. Understanding and Improving the Performance of Web Page Loads. https://api.semanticscholar.org/CorpusID:248435765
[59] V. S, Ramya R, and S. Selvan. 2023. A Survey on Modern Innovative Secured Transport Layer Protocols on Recent Advances. In *International Conference on Computing Methodologies and Communication*. https://doi.org/10.1109/ICCMC56507.2023.10084044
[60] Dominik Scholz, Benedikt Jaeger, Lukas Schwaighofer, Daniel Raumer, Fabien Geyer, and Georg Carle. 2018. Towards a Deeper Understanding of TCP BBR Congestion Control. In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*. 1–9. https://doi.org/10.23919/IFIPNetworking.2018.8696830
[61] Vyas Sekar, Michael Butkiewicz, and V. Harsha Madhyastha. 2014. Characterizing web page complexity and its impact. *IEEE/ACM Transactions on Networking* 22, 3 (jun 2014), 948–961.
[62] William Sentosa, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Haitham Hassanieh, and Bruce Maggs. 2023. DChannel: Accelerating Mobile Applications With Parallel High-bandwidth and Low-latency Channels. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 419–436. https://www.usenix.org/conference/nsdi23/presentation/sentosa
[63] Michael Seufert, Raimund Schatz, Nikolas Wehner, Bruno Gardlo, and Pedro Casas. 2019. Is QUIC becoming the New TCP? On the Potential Impact of a New Protocol on Networked Multimedia QoE. In *2019 Eleventh International Conference on Quality of Multimedia Experience (QoMEX)*. 1–6. https://doi.org/10.1109/QoMEX.2019.8743223
[64] Tanya Shreedhar, Rohit Panda, Sergey Podanev, and Vaibhav Bajpai. 2022. Evaluating QUIC Performance Over Web, Cloud Storage, and Video Workloads. *IEEE Transactions on Network and Service Management* 19, 2 (2022), 1366–1381. https://doi.org/10.1109/TNSM.2021.3134562
[65] Aditya N Srivastava. 2024. Thesis Survey: Web Page Preference. https://forms.gle/fvtkzrrvfTuNnxhr5. Online survey.
[66] R. Stewart. 2007. Stream Control Transmission Protocol. RFC 4960. https://doi.org/10.17487/RFC4960
[67] Randall R. Stewart, Qiaobing Xie, Ken Morneault, Chip Sharp, Hannes Schwarzbauer, Tom Taylor, Ian Rytina, Milan Kalla, Lixia Zhang, and Vern Paxson. 2000. Stream control transmission protocol. *RFC 2960* (2000).
[68] Parul Tomar, Gyanendra Kumar, Lal Pratap Verma, Varun Kumar Sharma, Dimitris N. Kanellopoulos, SurSingh Rawat, and Youseef A. Alotaibi. 2022. CMT-SCTP and MPTCP Multipath Transport Protocols: A Comprehensive Review. *Electronics* (2022). https://api.semanticscholar.org/CorpusID:251264553
[69] Tobias Viernickel, Alexander Froemmgen, Amr Rizk, Boris Koldehofe, and Ralf Steinmetz. 2018. Multipath QUIC: A deployable multipath transport protocol. In *2018 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.
[70] Anduo Wang, Sébastien Barre, Jana Iyengar, Paulo Ferreira, and Roch Sivakumar. 2014. Speedy transactions in multipath TCP. In *Proceedings of the 2014 ACM conference on SIGCOMM*. 109–120.
[71] Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. 2013. Demystifying Page Load Performance with WProf. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 473–485. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao
[72] Wikipedia contributors. 2022. HTTP. https://en.wikipedia.org/wiki/HTTP.
[73] Wikipedia contributors. 2024. 4G. https://en.wikipedia.org/wiki/4G
[74] Konrad Wolsing, Jan Rüth, Klaus Wehrle, and Oliver Hohlfeld. 2020. A Performance Perspective on Web Optimized Protocol Stacks: TCP+TLS+HTTP/2 vs. QUIC. In *Proceedings of the 15th International Conference on Emerging Networking Experiments and Technologies (CoNEXT)*. ACM.
[75] KaKei Wong and Lin Cui. 2023. Fine-grained HTTP/3 prioritization via reinforcement learning. *Computer Networks* 233 (2023), 109880. https://doi.org/10.1016/j.comnet.2023.109880
[76] Yitao Xing, Kaiping Xue, Yuanzun Zhang, Jiangping Han, Jian Li, David S. L. Wei, Ruidong Li, Qibin Sun, and Jun Lu. 2023. A Stream-Aware MPQUIC Scheduler for HTTP Traffic in Mobile Networks. *IEEE Transactions on Wireless Communications* 22 (2023), 2775–2788. https://api.semanticscholar.org/CorpusID:253315377
[77] Gina Yuan, David Zhang, Matthew Sotoudeh, Michael Welzl, and Keith Winstein. 2022. Sidecar: in-network performance enhancements in the age of paranoid transport protocols. *Proceedings of the 21st ACM Workshop on Hot Topics in Networks* (2022). https://api.semanticscholar.org/CorpusID:253524372
[78] Mengying Zhou, Yang Chen, Shihan Lin, Xin Wang, Bingyang Liu, and Aaron Ding. 2024. Dissecting the Applicability of HTTP/3 in Content Delivery Networks.