Funded by the
Erasmus+ Programme
of the European Union

Dissecting HTTP/2 and QUIC: Measurement, Evaluation and Optimization

JAWAD MANZOOR

Thesis submitted for the degree of Doctor of Philosophy at
Université catholique de Louvain
and
Universitat Politècnica de Catalunya

Advisors:
DR. RAMIN SADRE
DR. LLORENÇ CERDÀ-ALABERN

March 2019

To my family with whom i could not spend all those hours.

Dissecting HTTP/2 and QUIC: Measurement, Evaluation and Optimization. *April 2019.*

Jawad Manzoor
jawad.manzoor@uclouvain.be
jawad@ac.upc.edu

Computer Networks and Distributed Systems Group
Universitat Politècnica de Catalunya
Jordi Girona 1-3 C6, D6
08034 - Barcelona, Spain

Acknowledgments

I would like to show my gratitude to all the people involved in my PhD research. I am grateful to my advisors, Ramin Sadre and Llorenç Cerdà-Alabern for their continuous support and guidance throughout the years. They provided new ideas and research directions as well as valuable feedback and advice on my work.

I feel very fortunate to have Idilio Drago (Politecnico di Torino) as a collaborator. He not only provided me the opportunity to work on large and diverse traffic datasets for my research, but also provided great technical help expert opinion on different topics. This work would not have been possible without him.

I would also like to thank Leandro Navarro (UPC), the EMJDDC program coordinator for his kind support in administrative and bureaucratic matters.

I cannot thank my wife Dr. Dur E Zahra enough for supporting me all these years and taking great care of our daughters so that I can focus on my research.

Finally, I am grateful to my lovely daughters Raniya and Sofia, my parents and my family for their love, support and sacrifices.

This work was funded by the doctoral fellowship of European Commission (EACEA) through the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJDDC) program.

Abstract

The Internet is evolving from the perspective of both usage and connectivity. The meteoric rise of smartphones has not only facilitated connectivity for the masses, it has also increased their appetite for more responsive applications. The widespread availability of wireless networks has caused a paradigm shift in the way we access the Internet. This shift has resulted in a new trend where traditional applications are getting migrated to the cloud, e.g., Microsoft Office 365, Google Apps etc. As a result, modern web content has become extremely complex and requires efficient web delivery protocols to maintain users' experience regardless of the technology they use to connect to the Internet and despite variations in the quality of users' Internet connectivity.

To achieve this goal, efforts have been put into optimizing existing web and transport protocols, designing new low latency transport protocols and introducing enhancements in the WiFi MAC layer. In recent years, several improvements have been introduced in the HTTP protocol resulting in the HTTP/2 standard which allows more efficient use of network resources and a reduced perception of latency. QUIC transport protocol is another example of these ambitious efforts. Initially developed by Google as an experiment, the protocol has already made phenomenal strides, thanks to its support in Google's servers and Chrome browser.

However there is a lack of sufficient understanding and evaluation of these new protocols across a range of environments, which opens new opportunities for research in this direction. This thesis provides a comprehensive study on the behavior, usage and performance of HTTP/2 and QUIC, and advances them by implementing several optimizations. First, in order to understand the behavior of HTTP/1 and HTTP/2 traffic we analyze datasets of passive measurements collected in various operational networks and discover that they have very different characteristics. This calls for a reappraisal of traffic models, as well as HTTP traffic simulation and benchmarking approaches that were built on the understanding of HTTP/1 traffic only and may no longer be valid for modern web traffic. We develop a machine learning-based method compatible with existing flow monitoring systems for the classification of encrypted web traffic into appropriate HTTP versions. This will enable network administrators to identify H1 and H2 flows for network managements tasks such as traffic shaping or prioritization. We also investigate the behavior of HTTP/2 stream multiplexing in the wild. We devise a methodology for analysis of large datasets of network traffic comprising over 200 million flows to quantify the usage of H2 multiplexing in the wild and to understand its implications for network infrastructure.

Next, we show with the help of emulations that HTTP/2 exhibits poor performance in adverse scenarios such as under high packet losses or network congestion. We confirm that the use of a single connection sometimes impairs application performance of HTTP/2 and implement an optimization in Chromium browser to make it more robust in such scenarios.

Finally, we collect and analyze QUIC and TCP traffic in a production wireless mesh network. Our results show that while QUIC outperforms TCP in fixed networks, it exhibits significantly lower performance than TCP when there are wireless links in the end-to-end path. To see why this is the case, we carefully examine how delay variations which are common in wireless networks impact the congestion control and loss detection algorithms of QUIC. We also explore the interaction of QUIC transport with the advanced link layer features of WiFi such as frame aggregation. We fine-tune QUIC based on our findings and show notable increase in performance.

Resumen

Internet está evolucionando desde la perspectiva del uso y la conectividad. El ascenso meteórico de los teléfonos inteligentes no solo ha facilitado la conectividad para las masas, sino que también ha aumentado su apetito por aplicaciones más exigentes. La disponibilidad generalizada de las redes inalámbricas ha provocado un cambio de paradigma en la forma en que accedemos a Internet. Este cambio ha dado lugar a una nueva tendencia en la que las aplicaciones tradicionales se están migrando a la nube. Como resultado, el contenido web moderno se ha vuelto extremadamente complejo y requiere protocolos de entrega web eficientes para mantener la calidad de experiencia de los usuarios.

Para lograr este objetivo, se han realizado esfuerzos para optimizar los protocolos web y de transporte existentes, diseñar nuevos protocolos de transporte de baja latencia e introducir mejoras en la capa MAC de WiFi. En los últimos años, se han introducido varias mejoras en el protocolo HTTP que dan como resultado el estándar HTTP/2 que permite un uso más eficiente de los recursos de la red y una menor percepción de la latencia. El protocolo de transporte QUIC es otro ejemplo de estos esfuerzos ambiciosos. Inicialmente desarrollado por Google como un experimento, el protocolo ya ha hecho grandes avances, gracias a su soporte en los servidores de Google y el navegador Chrome.

Esta tesis proporciona un estudio exhaustivo sobre el comportamiento, uso y rendimiento de HTTP/2 y QUIC, y los mejora mediante la implementación de varias optimizaciones. Primero, para comprender el comportamiento del tráfico HTTP/1 y HTTP/2, analizamos los conjuntos de datos de mediciones pasivas recopiladas en varias redes operativas y descubrimos que tienen características muy diferentes. Esto requiere una reevaluación de los modelos de tráfico, así como los métodos de simulación y evaluación comparativa del tráfico HTTP que se desarrollaron en el estudio hecho anteriormente sólo considerando el tráfico HTTP/1, y que ya no sean válidos para el tráfico web moderno. Desarrollamos un método basado en aprendizaje automático compatible con los sistemas de monitoreo de flujo existentes para la clasificación del tráfico web encriptado en las versiones HTTP. Esto permitirá a los administradores de red identificar los flujos de HTTP/1 y HTTP/2 para las tareas de administración de red, como la configuración del tráfico o la priorización. También investigamos el comportamiento de la multiplexación de flujos HTTP/2. Diseñamos una metodología para el análisis de grandes conjuntos de datos de tráfico de red que comprende más de 200 millones de flujos para cuantificar el uso de la multiplexación HTTP/2 y para comprender sus implicaciones para la infraestructura de red.

A continuación, mostramos con la ayuda de las emulaciones que HTTP/2 muestra un rendimiento deficiente en escenarios adversos, como por ejemplo, una gran pérdida de paquetes o la congestión de la red. Confirmamos que el uso de una sola conexión a veces perjudica el rendimiento de la aplicación de HTTP/2 e implementamos una optimización en el navegador Chromium para hacerlo más robusto en tales escenarios.

Finalmente, recopilamos y analizamos el tráfico de QUIC y TCP en una red de malla inalámbrica en producción. Nuestros resultados muestran que, si bien QUIC supera a TCP en redes fijas, puede presentar un rendimiento significativamente menor que TCP cuando hay enlaces inalámbricos en la ruta de extremo a extremo. Para ver por qué ocurre, examinamos cuidadosamente cómo las variaciones de retardo, que son comunes en las redes inalámbricas, afectan el control de congestión y los algoritmos de detección de pérdida de QUIC. También exploramos la interacción de QUIC con las características avanzadas de la capa de enlace de WiFi, como la agregación de tramas. Ajustando QUIC en función de nuestros hallazgos mostramos que puede conseguirse un notable aumento en el rendimiento.

Contents

List of Figures

List of Tables

List of Acronyms

ABR	Adaptive Bit Rate
ALPN	Application Layer Protocol Negotiation
CDN	Content Distribution Network
DNS	Domain Name System
DPI	Deep Packet Inspection
ECMP	Equal-Cost Multipath Routing
FQDN	Fully Qualified Domain Name
HTTP	Hypertext Transfer Protocol
H2	HTTP/2
H1	HTTP/1
HOL	Head-of-line
IETF	Internet Engineering Task Force
ISP	Internet Service Provider
MAC	Media Acces Control
MPDU	MAC Protocol Data Unit
MSDU	MAC Service Data Unit
MPTCP	Multipath TCP
NPN	Next Protocol Negotiation
NAT	Network Address Translation

OS	Operating System
PLT	Page Load Time
Polito	Politecnico di Torino
QMP	Quick Mesh Project
QoS	Quality of Service
QUIC	Quick UDP Internet Connections
RFC	Request for Comments
RTT	Round-Trip Time
SNI	Server Name Identification
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UPC	Universitat Politècnica de Catalunya
WMN	Wireless Mesh Network

Chapter 1

Introduction

The Internet has become an essential part of our daily lives. Social networking, online shopping, e-banking, instant messaging and video streaming are just a few examples of the web services that we use on a daily basis. These web applications and services vary in complexity and demands, e.g., online gaming and video conferencing are sensitive to latency while online high-definition video streaming requires good bandwidth. The exponential growth in the adoption of mobile phones and widespread availability of 3G/4G and WiFi networks has caused a paradigm shift in the way these services are accessed. Today, 5G is on the horizon and WiFi networks are commonplace in all sectors including homes, offices, shopping malls, restaurants, hospitals and university campuses. This indicates that wireless traffic will further grow in the future. In a recent Cisco white paper it is predicted that wireless and mobile device traffic will exceed that of PCs, and comprise more than 63 percent of total IP traffic by 2021. This has resulted in a new trend where traditional applications are getting migrated to the cloud, e.g., Microsoft Office 365, Google Apps, etc. As a result, modern web content has become extremely complex. This complexity requires efficient web delivery protocols to maintain users' experience regardless of the technology they use to connect to the Internet and despite variations in the quality of users' Internet connectivity.

In the recent years, several improvements have been introduced in the application layer Hypertext Transfer Protocol (HTTP) to adapt it to modern web content. The Internet Engineering Task Force (IETF) has standardized HTTP/2 (H2) which allows more efficient use of network resources and a reduced perception of latency. It solves several performance issues that are present in its predecessor HTTP/1 (H1). Currently H2 is supported by all major desktop and mobile browsers including Chrome, Firefox and iOS Safari, and popular web servers including Apache, Microsoft IIS, Caddy, nginx and LightSpeed. Major Content Distribution Networks (CDNs) including Akamai, Cloudflare, Microsoft Azure and AWS CloudFront also support H2. Even though H2 has provided great improvements over H1, it still suffers from other issues present in the underlying Transmission Control Protocol (TCP) layer.

Meanwhile Google designed a new transport protocol known as Quick UDP Internet Connections (QUIC) as an alternative to TCP. QUIC has already made phenomenal strides, thanks to its support in Google's servers and Chrome browser.

1.1 Purpose Statement

The growing adoption of H2 and QUIC is a testament to the fact that they will define the future of the Internet. However there is lack of sufficient understanding and evaluation of these new protocols across a range of environments, which opens new research opportunities on the usage, adoption and performance of these protocols. State-of-the-art in the empirical evaluation of these protocols in wireless networks is mostly constrained.

"The purpose of this thesis is to provide a comprehensive study on the behavior and performance of H2 and QUIC across a range of environments and develop optimizations where required."

To this end, the following research questions need to be addressed.

Q1: How H2 has changed web traffic and how can we identify it using encrypted network flows for network management?
Understanding the behavior of web traffic has far-reaching implications with respect to network resource utilization, traffic scheduling, congestion control, etc. The development of new features in H2 and the reorganization of servers and clients to profit from such features has lead to the speculation that the behavior of H1 and H2 traffic may be very different. This may cause the old web traffic models and traffic generation tools to be obsolete. In order to identify the characteristics of H1 and H2 traffic, analysis on large and diverse spatiotemporal datasets is required.

Once this is established, the next question is how to enable network administrators to identify H1 and H2 flows for network managements tasks such as traffic shaping or prioritization. Since *HTTPS Everywhere* is becoming a standard practice these days, any practical identification method must operate without inspection of payload. Secondly, since flow monitoring (NetFlow, IPFIX, etc.) is the most widely used technique for getting visibility into network bandwidth utilization and applying Quality of Service (QoS) policies, the identification method has to rely only on flow-based attributes without inspecting more complex features such as information from the TLS handshakes.

Q2: How can we quantify the impact of H2 stream multiplexing on network infrastructure in the wild?
Stream multiplexing is one of the most prominent features introduced in H2. It solves the head-of-line blocking issue of H1 and also eliminates the need to open a large number of concurrent connections to servers because all requests and responses can now be multiplexed on streams inside a single TCP connection. Thus, large scale adoption of H2 can have enormous benefits for the user experience and it can reduce load on the servers and network. A major concern however is that there is lack of research studies on the quantification of the impact of H2 stream multiplexing on network infrastructure in the wild.

Q3: How H2 and QUIC deal with network congestion and random losses and can they be improved?
Congestion and packet losses are common phenomena in the Internet. While H2 and QUIC have demonstrated significant performance gains over other protocols under good network conditions, their performance in networks with congestion is not well studied. While some researchers do study the behavior under adverse conditions and identify H2 to be particularly vulnerable in WiFi networks under high random packet losses, no practical solution to improve its performance is provided.

Q4: What are the factors that impact the performance of QUIC in WiFi networks and how to improve it?
The exponential growth in adoption of mobile phones and widespread availability of wireless networks has caused a paradigm shift in the way we access the Internet. However wireless networks are prone to errors due to obstacles, moving objects, whether condition, noise and interference with other wireless sources, etc. Since QUIC is relatively new and still under development, its performance in real-world wireless networks has not been rigorously investigated. In particular, the interplay between transport layer attributes like packet acknowledgements and congestion

window growth, and MAC layer features such as frame aggregation can significantly affect the overall throughput. Research in this direction is critical to improve the performance of QUIC in WiFi networks.

1.2 Contributions

In this section, the major contributions of the thesis are summarized and mapped to the associated research question. The four major contributions of the thesis are as follows:

C1: A first study on the characterization of H1 and H2 traffic and a machine learning-based method for classification of encrypted HTTP traffic
This contribution reveals characteristics of H1 and H2 traffic using datasets of passive measurements collected in various operational networks. Our findings affirm that H1 and H2 flows are very different both in terms of duration and size. We also observe a rapid adoption of H2 among popular web services. Next, we present a lightweight method for the classification of encrypted web traffic into appropriate HTTP versions. In order to make the method practically feasible, we use machine learning with basic information commonly available in aggregated flow traces (e.g., NetFlow records). We show that a small labeled dataset is sufficient for training the system, and it accurately classifies traffic for several months, potentially from different measurement locations, without the need for retraining. Therefore, the method is simple, scalable, and applicable to scenarios where Deep Packet Inspection (DPI) is not possible.

C2: A methodology for analysis of H2 stream multiplexing
The methodology performs an analysis on large datasets of network traffic comprising over 200 million flows collected during a 4 month period, to detect the extent of usage of a single multiplexed connection in H2 protocol. Contrary to popular belief, our experiments show that a significant number of H2 accesses are performed using several parallel connections to the same domain. We investigate the possible reasons for this behavior and discuss its implications for the future of the Internet.

C3: A performance assessment of H2 and QUIC in adverse network conditions and development of an optimization technique for H2
This contribution provides performance evaluation of H2 and QUIC in scenarios with

network congestion or high random packet loss rate. We argue that the design choice of using a single connection can result in poor application performance under these scenarios. We design and implement an optimizing technique in the source code of the open source Chromium browser that establishes two parallel connections with the server. This approach has similar outcome as Multipath TCP (MPTCP) but it has the advantage of being implemented at the application layer. The evaluation of our proposed optimization shows that it can reduce H2 page load time from roughly 17 s to 8 s for large web transfers.

C4: Identification of factors impacting QUIC performance over wireless links and optimization of the protocol
This contribution highlights the determining factors of QUIC's performance in WiFi networks. Experiments are performed in a production Wireless Mesh Network (WMN) and machine learning techniques are applied to the collected network traces. The interaction of QUIC transport with the advanced features of WiFi such as frame aggregation are investigated. Furthermore, the impact of high delay variations on the congestion control and loss detection algorithms of QUIC are evaluated. We implement several optimizations in QUIC and show up to 52% increase in throughput.

1.2.1 List of Publications

[1] Manzoor, J., Drago, I. & Sadre, R. *The curious case of parallel connections in http/2* in *2016 12th International Conference on Network and Service Management (CNSM)* (2016), 174–180

[2] Manzoor, J., Drago, I. & Sadre, R. *How http/2 is changing web traffic and how to detect it* in *2017 Network Traffic Measurement and Analysis Conference (TMA)* (2017), 1–9

[3] Manzoor, J., Sadre, R., Drago, I. & Cerda-Alabern, L. *Is There a Case for Parallel Connections with Modern Web Protocols?* in *2018 IFIP Networking* (2018)

[4] Manzoor, J., Cerda-Alabern, L., Sadre, R. & Drago, I. *Improving Performance of QUIC in WiFi* in *2019 IEEE Wireless Communications and Networking Conference (WCNC)* (2019)

[5] Manzoor, J., Cerda-Alabern, L., Sadre, R. & Drago, I. *On the Causes of Performance Issues of QUIC over Wireless Links* in *To be submitted to Ad Hoc Networks Journal* (2019)

1.2.2 Thesis Outline

This thesis is organized into 7 chapters. Fig. 1.1 depicts a general overview of the current networking protocol stack. The research work done in each chapter of this thesis is superimposed along with pointers to the portions of the protocol stack that are involved.

In this chapter we provide an introduction of the research domain and present the research questions and contributions. We give a detailed background of the concepts and protocols involved in our research in Chapter 2. We also present an in-depth review of the state-of-the-art in measurement and performance evaluation of these protocols and explain where prior studies fall short.

Chapter 3 and 4 focus on measurement and analysis of H1 and H2 protocols. Chapter 3 presents a first study on the comparison of H1 and H2 traffic characteristics and a machine learning-based framework for classification of encrypted HTTP traffic. Chapter 4 particularly focuses of the analysis of H2 multiplexing behavior in the wild. A divergent behavior of H2 connection establishment is revealed and investigated in this chapter along with its implications for the future of the Internet.

Chapter 5 and 6 provide performance evaluation and optimization of H2 and QUIC protocols. Chapter 5 presents a performance comparison of various application and transport protocols and validates that H2 exhibits poor performance in some adverse network conditions. An optimization is then implemented for H2 protocol which significantly increases the performance. Chapter 6 presents an in-depth evaluation of QUIC in WiFi networks. It is shown that QUIC has achieves much lower throughput than TCP in WiFi networks and the root causes of this problem are identified. Next, several optimizations are implemented and evaluated.

Chapter 7 concludes the thesis and provides future research direction related to our research.

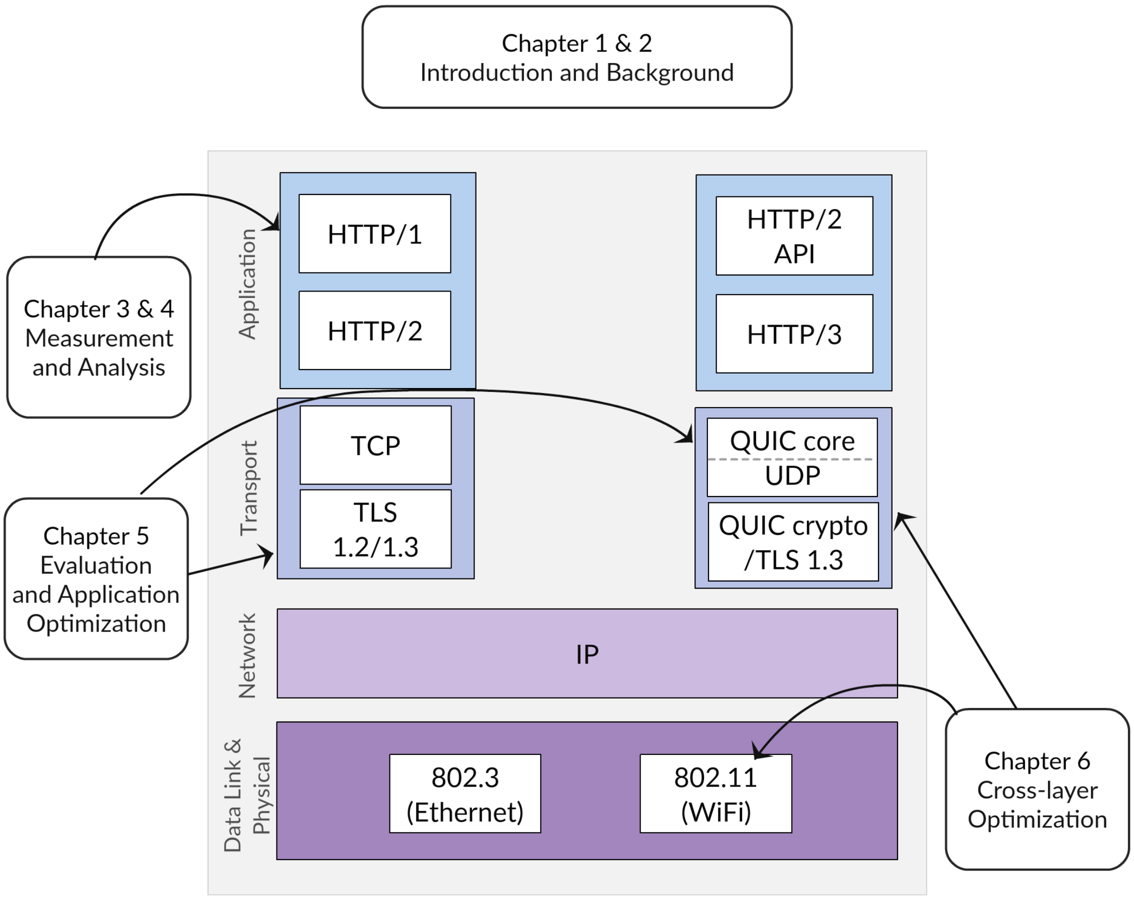


Figure 1.1: Outline of thesis

Chapter 2

Background and Related work

2.1 Evolution of HTTP

HTTP is the most popular protocol for delivering web content. HTTP has occupied a dominant position among Internet traffic, and it is expected to maintain such position in the future as more and more applications are getting migrated to the web. It was developed by Tim-Berners-Lee in 1989 for delivering simple documents and other web resources over the Internet [6].

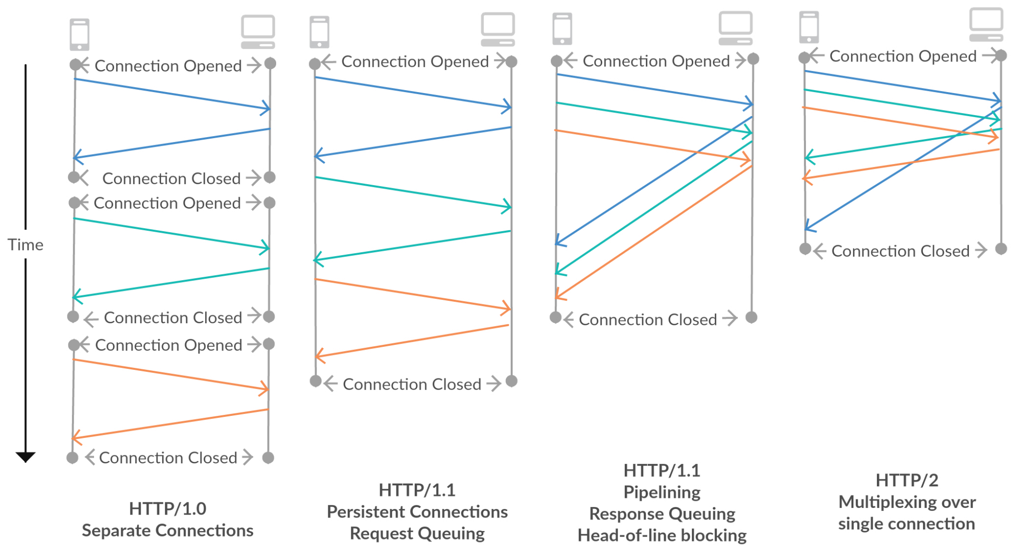


Figure 2.1: Comparison of HTTP versions

2.1.1 HTTP version 1 (H1)

In HTTP version 1.0, a separate network connection is required to request each resource from the same server. This became a problem for large web pages consisting of many objects because a full three-way handshake is required to establish the underlying TCP connection, resulting in long delays. HTTP version 1.1 introduced the concept of *persistent connections*, so that the same connection is reused to request additional resources from the same server, reducing expensive round trips for connection setup. To further improve speed, *pipelining* allows the client to asynchronously send multiple requests to the server without waiting for the response. However, the specification requires that the server sends the responses in the same order in which it received the requests. This can result in so-called Head-of-line (HOL) blocking, where a big response at the head of the queue holds up all following responses. In practice, many popular web browsers do not use pipelining by default because of poorly behaving servers.

Browser vendors have reacted to these inefficiencies throughout the years by deploying ad-hoc optimizations to speed up Page Load Time (PLT) [1]. One such optimization is the opening of several persistent TCP connections towards each web server when retrieving pages. Browsers can issue requests in parallel in multiple connections, reducing the effect of HOL blocking. As a side effect, they compete for resources with other applications in the network more aggressively. For example, while the transfer rate of a single TCP connection is limited by the small congestion window (cwnd) during TCP slow start, multiple connections sum up their cwnd, resulting in faster startup rates. The fierce competition among browser manufacturers has pushed browsers to open a large number of parallel connections in an attempt to speed up page rendering [7]. The first specifications of H1 stated that a single-user client *SHOULD NOT* maintain more than two connections with any server or proxy. However, modern browsers implement different limits on the number of parallel connection with up to 13 connections per hostname and up to 20 maximum connections [8] as shown in Table 2.1. On the server side several other hacks and tweaks like *spriting*, *inlining* and *domain sharding* are used but the management of these tweaks is a difficult task.

[1]Page Load Time is the time from when a user fires a web page request (e.g., by clicking on a link) until the page is fully loaded by the browser.

Table 2.1: Parallel connection limit of different web browsers.
(Source www.browserscope.org [2016])

Browser Name	Connections per Hostname	Max Connections
Chrome 50	6	10
Firefox 45	6	17
IE 11	13	17
Safari 9	6	17
Opera 36	6	10
Chrome Android 48	6	10
Chrome for iOS 44	6	17
Safari for iOS 9	6	17
Firefox Mobile 43	10	20
Opera Mobile 32	6	10

2.1.2 HTTP version 2 (H2)

As web pages got more complex over the years, these inefficiencies started to hurt the PLT and researchers started developing new protocols to replace H1. Google developed an experimental protocol called SPDY which gained a lot of popularity and was deployed on some of the most popular websites like Google, Facebook and Twitter. The successful deployment of SPDY over Transport Layer Security (TLS) opened the way for the HTTP evolution, triggering the standardization of H2 [9] by the IETF in 2015 with Request for Comments (RFC) 7540. H2 borrows many of SPDY's principles and solves several shortcomings of H1. In particular, H2 *multiplexes* requests in a single TCP connection, eliminating the HOL blocking bottleneck. The evolution of the HTTP protocol is shown in Figure 2.1. H2 enables a more efficient use of network resources and reduces latency by introducing several new features.

- Multiplexing: One of the most important features of H2 is multiplexing. It allows the client and server to send multiple HTTP requests and responses asynchronously in multiple streams on a single TCP connection. Responses can be interleaved, avoiding in this way HOL blocking of H1.
- Compression: The header data is compressed to reduce the overheads of redundant header fields. HPACK [10] is used for compression.
- Server Push: It allows the server to proactively push resources to clients. When a client requests a resource, the server predicts the other resources related to it, and pushes them along with the response of the original request. This helps in improving the page load time.

- Content priority: Some resources are more important than others when rendering a page. A client is allowed to specify the importance of each resource, so that it can be transferred by the server in the preferred order.

As of January 2017, around 5.3M domains support H2 compared to just over 600 domains in May 2015 [11]. H2 is supported by all major desktop and mobile browsers including Chrome, Firefox and iOS Safari, and popular web servers including Apache, Microsoft IIS, Caddy, nginx and LightSpeed [12]. Major Content Delivery Networks (CDNs) including Akamai, Cloudflare, Microsoft Azure, AWS CloudFront, etc. also support H2.

Even though H2 has solved many issues that were present in H1, it still suffers from problems due to the underlying TCP layer. TCP guarantees reliable, in-order delivery of packets. If a packet gets lost in transit, TCP will pause all subsequent packet until the lost packet is successfully received. This causes HOL blocking issue at the transport layer resulting in performance degradation.

Another issue is switching between networks, e.g., WiFi and LTE. Since TCP uses the IP address and port numbers to identify a connection, change of network results in termination of the old connection and establishment of a new one.

2.2 Advent of QUIC transport protocol

In 2012 Google started working on an experimental UDP-based transport protocol called QUIC. It was deployed both in Chrome browser and most of Google services, including search engine and Youtube. QUIC is an alternative to TCP, which is the most popular transport protocol that has been driving the Internet for the last four decades. The design goals of QUIC include rapid evolution and deployment, multistreaming, latency reduction, flexible congestion control, connection migration and resilience to NAT-rebinding. To meet these goals, QUIC is developed as a user-space transport protocol running on top of UDP. It provides several cross-layer enhancements, covering the weaknesses of TCP for transporting web content.

In 2016 the IETF chartered the QUIC working group which used Google's QUIC as starting point and created a set of drafts. Due to several important changes introduced during the last two years, the IETF version and Google version have diverged. TLS 1.3 has replaced Google's crypto libraries. A new header compression scheme QPACK has been designed and included to replace HPACK, H2's header

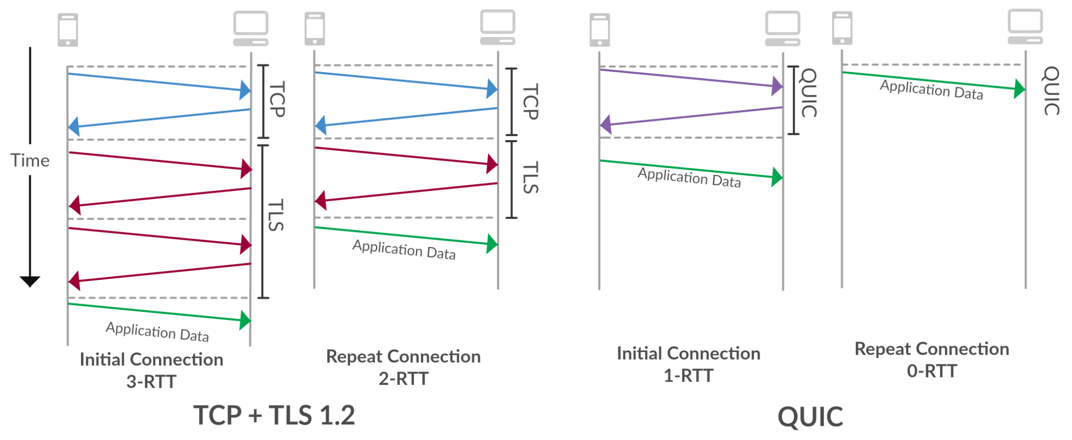


Figure 2.2: Comparison of TCP and QUIC connection establishment

compression scheme. The IETF is also working on HTTP/3 which will use QUIC transport protocol instead of TCP. Several implementations of IETF and Google QUIC are available and a comprehensive list is available at [13].

The following are some prominent features of QUIC:

- Multiplexing
  Stream multiplexing was implemented in H2 over TCP to solve to the head-of-line blocking at the application layer. However, the problem was only partially solved and the head-of-line blocking was passed down to the transport layer. When a packet is lost, all H2 streams are blocked and TCP buffers any subsequent packets until the successful re transmission of the lost packet. QUIC design avoids head-of-line blocking at both application and transport layer and a QUIC connection can still make forward progress on some streams while others are paused due to packet loss.

- Connection Establishment
  Establishment of a secure connection usually requires several roundtrips. Round-trip latency can make a big difference on long distance links or WiFi and cellular networks. QUIC combines the transport and crypto handshake and reduces the number of roundtrips required for setting up a connection. Fig. 2.2 shows the comparison of connection establishment of TCP and QUIC with encryption. It takes 3 Round-Trip Times (RTTs) for TCP and TLS 1.2 to establish the initial connection with an unknown server. First roundtrip is required for the three-way TCP handshake. Second roundtrip is required to negotiate the TLS version and ciphersuite. In the third roundtrip key exchange is initiated which is used to establish the symmetric key for the following session. In case of

resumed connections only 2-RTTs are required. This procedure is improved in TCP with TLS 1.3 where it takes 2-RTT for initial and 1-RTT for resumed connections.

QUIC further reduces this latency and takes only takes only 1-RTT to establish a secure connection with an unknown server using inchoate ClientHello (CHLO) and starts application data transfer in the next RTT along with complete CHLO. Repeat connections are started with 0-RTT by sending complete CHLO along with the encrypted request.

- Congestion control
  QUIC has pluggable congestion control which allows different congestion control algorithms to be used, e.g., Cubic, NewReno, BBR. The default congestion control algorithm in QUIC is a reimplementation of TCP Cubic with some changes. An important change in QUIC's congestion control algorithm is that it is less aggressive than TCP in reducing cwnd size. When packet loss is detected by QUIC's loss detection algorithm it reduces the cwnd size by a factor which is half that of TCP and in this way it emulates the behavior of two connections. QUIC also has richer signaling, is more resilient to packet reordering, and has better RTT estimation and loss recovery than TCP.

- Connection Migration
  QUIC supports connection migration e.g., from WiFi to cellular because QUIC connections are identified by a 64-bit connection ID which remains the same across these migrations. On the other hand a TCP connection is identified by a 4-tuple (source and destination IP address, source and destination port number). Therefore, a TCP connection does not survive IP address changes and NAT re-bindings.

- Header and Payload Encryption
  QUIC hides most of its state information by using header and payload encryption by default. While it helps in avoiding network ossification and pervasive monitoring, it comes at the cost of complicating network operations and management. Routine tasks of network operators such as detecting anomalies, capacity planning, and traffic engineering become harder due to transport layer header encryption.

2.3 TCP and QUIC over wireless links

2.3.1 Transport protocol enhancements for WiFi

Given the popularity of TCP, several researchers have investigated different ways to improve the performance of TCP in wireless networks. One of the proposed enhancements is reducing the acknowledgement frequency and performing delayed cumulative acknowledgements. A key factor affecting TCP performance in wireless networks is the contention and collision between ACK and data packets. Reducing the number of ACKs saves wireless resources and reduces interference with other packets. Moreover, lowering the ACK frequency increases burstiness of traffic as the sender releases a micro burst of packets after receiving the cumulative ACK. This reduces the inter-packet time increasing the opportunities for frame aggregation at the 802.11 Media Acces Control (MAC) layer. Altman et. al [14] investigated the impact of increasing the TCP delayed acknowledgement mechanism to more than two segments as recommended by RFC 1122. Singh et. al [15] propose TCP with adaptive delayed acknowledgement, which aims to reduce the number of ACKs to one per congestion window. Oliveira et. al [16] propose Dynamic Adaptive Acknowledgement where the delay window is adjusted according to the channel condition. In [17], the same authors provide an improved delay window strategy for robustness against losses.

Xylomenos et. al [18] point out that the performance of TCP can degrade over wireless links due to spurious timeouts and spurious fast retransmits caused by TCP's inability to distinguish between acknowledgments for original packets and retransmissions. The *Eifel scheme* uses TCP timestamps to avoid these issues. Scharf et. al [19] study the sensitivity of TCP to sudden delay variations in mobile networks and quantify the risk of spurious TCP timeouts caused by changing round-trip times. Gurtov et. al [20] also report that long sudden delays during data transfers are not uncommon in wireless WANs and can lead to unnecessary retransmissions and low throughput. They show that an aggressive TCP retransmission timer may trigger a chain of spurious retransmissions. Similar research studies to understand the interaction of QUIC's congestion control and loss detection algorithms with wireless networks are largely missing.

2.3.2 MAC layer enhancements for WiFi

WiFi technology based on the IEEE 802.11 standards has undergone an enormous evolution in the recent years. A recent measurement study [21] with millions of Cisco Meraki access points (APs) shows that around 99% of the APs use the 802.11n and 802.11ac standards. These standards introduce enhancements to increase data rates. Among them, *frame aggregation* is a simple method to enhance throughput.

The wireless medium has a high overhead, which includes the MAC and PHY headers, ACKs, backoff time and inter-frame spacing. For ACKs and small segments, the overhead in terms of bytes can be higher than the actual payload. The frame aggregation scheme amortizes this overhead and achieves high data throughput by combining multiple data frames into a single transmission unit.

Aggregation in WiFi MAC architecture is supported at two layers. In the first layer multiple MAC Service Data Units (MSDUs) are aggregated into an A-MSDU. In the second layer multiple A-MSDUs are combined to form an aggregated MAC Protocol Data Unit (MPDU). A detailed description of these concepts can be found in [22]. The impact of frame aggregation on TCP throughput has also been extensively evaluated [22–24] in the past. However, no such evaluation has been performed for QUIC. One of the objectives of our research is to investigate whether QUIC is able to profit from the advanced WiFi features.

2.4 State-of-the-art in performance measurement

2.4.1 H1, SPDY and H2

The constant evolution of the web has resulted in various updates to HTTP. Since its first version, the behavior of HTTP in the wild has been intensively studied and the gained insights have been used, for example, by web developers to improve the performance of web applications [25] and by researchers to build traffic models for simulation and benchmarking [26].

Given that HTTP has become the *de facto* protocol for deploying new applications and services, there are many studies on H1 web traffic, e.g., [26–28]. With the rise of modern protocols including SPDY and H2, researcher have developed frameworks to track the adoption of these protocols. Varvello et al. [29] build a measurement platform that actively monitors H2 adoption. They also provide content analysis and page-level characterization of H1 and H2 traffic using active measurements. They observe that H2 does not currently manage to serve a page using a single TCP connection. They show that half of the websites using H2 today use at least 20 TCP connections. The reason is that most websites have implemented H1 optimization techniques like domain sharding (splitting resources across multiple domains) which causes the browser to create a separate TCP connection for each domain. Zimmermann et al. [30] also investigate H2 adoption. They show that around 12.5% of Alexa top-million domains provide full H2 support, with a 66% increase in H2 enabled domains between Sep 2016 and Jan 2017. They also study H2 performance, but in contrast to our work, they focus on the impact of H2 server push functionality, showing that some websites profit from the feature to speed up PLT.

Other researchers evaluate the performance improvements promised by these new protocols. Thomas et al. [7] compare SPDY's single long *elephant* flow to highly concurrent short-lived HTTP *mice* flows. They also point out that SPDY's single-connection approach has a disadvantage because of inequitable TCP backoff compared to H1 which uses multiple TCP connections. For example, a backoff algorithm that reduces bandwidth by 50% will result in only half of the bandwidth available to a single SPDY or H2 connection. On the other hand the available bandwidth of an application using 12 parallel connections will be reduced by only 4%. The authors discourage connection proliferation, but recommend using a small number of concurrent SPDY connections as a short-term mitigation strategy. Elkhatib et al. [31] identify the impact of network characteristics and website infrastructure on the performance of SPDY using simulations. They conclude that SPDY's default use of a single connection might be unwise in high bandwidth networks because it leads to lower performance compared to HTTPS which can exploit higher throughput by opening parallel connections. Wang et al. [32] perform extensive tests with both synthetic and real web pages. They find that when there is little network loss, SPDY has high performance, but in high loss scenarios the single connection hurts performance. They propose a solution for SPDY inefficiencies by tuning TCP by increasing initial window, increasing receive window and reducing backoff rate in case of packet loss.

Saxce et al. [33] perform experiments on clear-text traffic of H1 and H2, referred to as H1C and H2C respectively. They focus on cellular networks and find that apart from network conditions, PLT depends on the website structure and content. Erman et al. [34] focus on mobile browsing. They measure PLT for the top-20 Alexa websites using SPDY and H1 proxies in a 3G network and find that SPDY performs poorly due to the large number of retransmissions and TCP backoff. Zarifis et al. [35] explore the PLT differences between H1 and H2 using data collected from real users of the Akamai CDN. They find that in around 60% of the time H2 has lower PLT than H1.

The poor performance of H2 in scenarios with high packet loss rate has been discussed in many prior research works. This is particularly problematic in wireless networks that are prone to network errors causing random losses. However, no practical solution has been provided in any prior work to solve this issue. Only Wang et al. [32] provide a solution that requires changes to the TCP protocol. We argue that this solution is not practical since making changes to TCP is very hard as it is implemented in the Operating System (OS) kernel. Even if some changes are eventually implemented in the kernel, it can take many years for those changes to be globally deployed as it requires OS upgrade by end users. Our research focuses on implementing an application layer solution to this problem which can rapidly be deployed. To this end we implement our solution in the source code of the open source Chromium browser and show that our solution can significantly improve the performance of H2 in adverse network conditions.

2.4.2 MPTCP

There is another group of studies that investigate the use of newer transport protocols such as MPTCP [36] to alleviate the performance degradation of H2 and SPDY in presence of packet losses. MPTCP is an enhancement of TCP that can utilize multiple paths simultaneously. This allows bandwidth aggregation and improved reliability. The socket interface is the same as in TCP and the applications or upper-layer protocols remain unaware of the multiple paths. Paasch et al. propose different MPTCP modes to be used by mobile devices for Mobile/WiFi handover [37]. Chen et al. [38] and Ferlin et al. [39] evaluate MPTCP performance over WiFi and cellular networks. Deng et al. [40] create a custom android application and test throughput with congestion-control algorithms, choices of the MPTCP primary subflow, etc. These works however focus on general MPTCP performance, neglecting its impact on higher-layer protocols such as H2. They download different sized files ranging from 8 kB to 16 MB using wget on the client. They use performance metrics such as download time, loss rate, out-of-order delivery and round trip times.

Han et al. [41] provide the first measurement study of mobile web performance over MPTCP using SPDY and H1. They download 25 websites from Alexa top-100 list and measure PLT by combining LTE and WiFi with MPTCP. They show that MPTCP helps in mitigating performance penalties of SPDY under packet loss. In [42] the same authors provide a cost-benefit analysis of MPTCP in terms of improved user experience and energy consumption on mobile devices. The assumption in these studies is that clients are multi-homed – e.g., WiFi and LTE can be used simultaneously. Our work has a different scope: we check whether MPTCP has an impact on H2 performance even on single-homed devices, e.g., laptops or PCs with WiFi only. We show that the usage multiple connections with MPTCP helps in reducing the impact of packet losses and congestion in the network.

2.4.3 QUIC

QUIC aims to provide cross-layer enhancements to improve web performance. Due to its growing popularity and rapid adoption it has been the focus of many research studies in recent years. Rüth et al. [43] provide a broad assessment of QUIC usage in the wild. They probe the entire IPv4 address space and about 46% of the DNS namespace to detect QUIC support. Their findings reveal that the number of QUIC-capable IPs has more than tripled between 2016 and 2017 and around 161K domains are hosted on QUIC-enabled infrastructure.

Several recent papers explore the security of QUIC and particularly the implications of 0-RTT connection establishment. Jager et al. [44] describe attacks which transfer the potential weakness of TLS 1.2 to TLS 1.3 as well as QUIC. An attacker can pre-compute the signature which can have substantial impact since it is essentially equivalent to retrieving the long-term secret key of the server. Fischlin et al. [45] develop a model to reason about the security of multi-stage key exchange protocols and use this model to analyze the key exchange mechanism of QUIC. They show that QUIC meets the suggested security properties of the designers. However, the case of 0-RTT is different. In this case the client speculates that the server still uses a previously known public key. If the server public key changes it is assumed that some data of the 0-RTT connection will be sent under a key that was computed using the obsolete server public key. An active adversary can possibly exploit this. Lychev et al. [46] reveal that an adversary can degrade QUIC performance by making it fall back to TCP or creating an inconsistent view of the client and server handshake which results in a failed connection.

There is also ongoing work towards providing multipath support for QUIC. Apart from the QUIC working group charter which forsees this, independent researchers are also working on such prototypes. De Coninck et al. [47] design and evaluate Multipath QUIC that allows the usage of different network paths such as multiple paths with ECMP routing, WiFi and LTE on smartphones, or IPv4 and IPv6 on dual-stack hosts. While multipath transport protocols provide many benefits, factors such as path asymmetry are known to introduce performance issues. Rabitsch et al. [48] propose a proof-of-concept algorithm for a stream-aware packet scheduler for Multipath QUIC. This algorithm schedules stream data in such a way that slower paths do not delay the completion of individual streams.

There is a growing interest in investigating whether QUIC can effectively deliver multimedia traffic which comprises more than 70% of global Internet traffic. Streaming applications typically make use of HTTP adaptive streaming, e.g., MPEG DASH running over TCP while interactive applications generally use RTP running over UDP. Bhat et al. [49] adapt various DASH players to QUIC and investigate the QoE performance of the DASH algorithms with TCP in various environments. Their findings show that these algorithms perform better with TCP as compared to QUIC. Perkins et al. [50] present a minimal set of extensions to QUIC to support real-time media.

Other concurrent research studies investigate the performance aspects of QUIC. McQuistin et al. [51] study whether explicit congestion notification (ECN) can be used with UDP-based protocols such as QUIC. They test reachability of 2500 servers from the public NTP server pool and their findings suggest that ECN is broadly usable with UDP traffic.

Megyesi et al. [52] provide a comparative study of performance of QUIC, SPDY and H1 to see how they affect page load time. They test QUIC v20 by hosting web pages of different sizes and number of objects on Google Sites server and use a shaper server between the client PC and Google Sites to emulate different bandwidth, RTT and packet loss rates. They host four web pages having different size and number of images. They show that with packet loss, SPDY performs the worst, followed by QUIC and H1. In case of high bandwidth and large page size, QUIC's PLT is three times larger than H1 and SPDY. Carlucci et al. [53] investigate performance of QUIC v21 on emulated network environments using synthetic pages. They report that QUIC performs worse than H1, but better than SPDY with large web pages and 2% random packet loss rate. Without packet loss, QUIC performs better than H1 and SPDY for small and medium web pages, but worse for large pages due to the usage of only six parallel streams. They show that QUIC achieves higher goodput than TCP in under-buffered networks. Cook et al. [54] perform experiments in a local testbed, as well as using a 4G network card. They observe that QUIC and H2 have similar performance, but QUIC connections are less sensitive to delay and packet loss than H2. In case of 4G link, QUIC outperforms H2. Kakhki et al. [55] evaluate QUIC v34 and TCP performance of web page loading and video streaming. They show that QUIC generally outperforms TCP, but there are some scenarios where QUIC's performance is reduced, e.g., high packet reordering in network and large number of small objects in a web page. QUIC is however unfair to other protocols, taking more than 50% of the bottleneck bandwidth when competing with 2 or even 4 TCP connections.

In contrast to these performance studies which mostly focus on fixed networks, emulated network conditions or naive wireless network setups, our research is particularly focused on QUIC's performance in real-world WiFi networks. For this reason we perform our experiments in a production network. Our objective is to reveal the transport and link layer factors that impact the performance of QUIC. After identifying these factors we optimize QUIC protocol and the evaluation shows up to 52% improvement in throughput. Table 2.2 shows a comparison of selected related work on the performance evaluation of popular web and transport protocols.

Table 2.2: Summary of related work

Ref	Client	Server	Content	Network	Network conditions	Traffic shaping	Protocols
[30]	Chromium, Selenium	Mixed	Live websites	Fixed network	Typical	n.a	H1, H2
[31]	Chrome HAR capturer	Apache	Live websites, Cloned pages	Fixed network	Delay, Loss	Linux tc and netem	H1C,H1, SPDY
[32]	Epload SPDY client	Apache	Cloned + synthetic pages	Fixed network	Delay, Loss	Dummynet	H1, SPDY
[33]	Chromium	H2O	Cloned pages	Fixed network, 3G	Delay, Loss	Linux tc and netem	H1C, H2C
[34]	Chrome	SPDY and Squid proxy	Live websites	3G	Typical	n.a	H1, SPDY
[38]	Wget MPTCP v0.86	Apache2, MPTCP v0.86	8KB to 16MB sized files	WiFi, 3G, 4G	Typical	n.a	TCP, MPTCP
[41]	Chrome MPTCP v0.89	Squid and nghttpx proxy, MPTCP v0.89	Cloned pages	WiFi, LTE	Delay, Packet Loss	Dummynet	H1, SPDY, TCP, MPTCP
[52]	Chrome HAR capturer	Google Sites	Pages with diff. object sizes	Fixed network	Delay, Loss	Linux tc and netem	H1, SPDY, QUIC
[53]	Chromium	Chromium dummy server	Pages with diff. object sizes	Fixed network	Loss, Buffer size	Netshaper	H1, SPDY, QUIC
[54]	Perfy	Go-quic server	Cloned web pages	Fixed network, 4G	Delay, Loss	Linux tc and netem	H1, H2, QUIC
[55]	Chrome HAR capturer	Apache 2, QUIC server	Pages with diff. object sizes	Fixed network, 3G, LTE	Delay, Loss, Jitter	Linux tc and netem	TCP, QUIC

Chapter 3

Web traffic characterization and classification

With the advent of H2 and its adoption by the biggest Internet companies, it is expected that the traffic share of coexisting web protocols will change. As a consequence, our knowledge about the behavior of web traffic has to be updated, since these protocols may differ significantly in terms of key metrics, such as request/response sizes, flow duration and number of connections. However, identifying the HTTP version in large-scale traffic measurements is not trivial. First, since H2 traffic is almost always encrypted, and H1 is fast being migrated to HTTPS as well, any practical identification method has to operate without payload inspection. Second, whereas fields in TLS handshakes still provide hints about the used HTTP version as we will discuss later, this information is not available in basic flow-level statistics [56] (e.g., NetFlow records) that are widely used in real deployments. Thus, operators and researchers have no means to monitor the protocol adoption at large scale, or to study practical differences of the protocols from the network point-of-view.

In this chapter we investigate these issues. First, using datasets of flow-level statistics collected in operational networks, augmented with features extracted by means of Deep Packet Inspection (DPI), we provide a comparison of flow-level characteristics of H1 and H2 traffic. We confirm a rapid increase in H2 adoption among top Internet players. Furthermore, we show that new features introduced in H2, such as header compression, together with likely reorganizations of sites and servers to profit from such features, result in clear differences in the network fingerprints of the protocols.

Second, these differences in basic traffic features motivate the search for methods to identify HTTP versions when DPI is not possible. We propose the use of machine learning with basic features available in NetFlow measurements to classify HTTP traffic – thus, introducing a classifier that is able to operate with existing traces and monitoring equipment, and easily scalable to high-speed networks. We validate the classifier using real network traces where ground truth is known by means of DPI on TLS handshakes. We test various machine learning algorithms and conclude that decision trees are quite suitable for this problem. After training the system with a relatively small set of flows, the classifier can achieve very good accuracy at high speed with only basic NetFlow features. Finally, we show that the system accurately classifies traffic from different locations and for several months without the need for continuous retraining.

3.1 Characterization of H2 traffic

The new features of H2 are expected to result in a different network fingerprint when compared to H1. In this section, we compare flow-level characteristics of H1 and H2 traffic. We first describe our datasets and how we have built the ground truth. Then we discuss the adoption of H2 by popular web services and give a flow-level characterization of the observed traffic.

3.1.1 Datasets

We rely on data exported by Tstat [57] in our evaluation. Tstat acts as an advanced flow exporter, monitoring all TCP connections in the network and exporting more than 100 metrics for each flow. These metrics include all basic features present in popular versions of NetFlow — e.g., IP addresses, bytes and packet counters and flow start/end timestamps. Tstat adopts the classic five-tuple definition for a flow: client and server IP addresses, client and server port numbers and the transport layer protocol define which packets belong to a flow.

We collected traffic traces with Tstat in two different networks from June to December 2016. The *Campus* dataset was collected at border routers of a European university campus network. It includes traffic of around 15 000 users connected through wired network in research and administrative offices as well as WiFi access points. The *Residential* dataset was collected at a Point of Presence (PoP) of a European ISP. This network includes traffic of around 25 000 households that are provided access via ADSL and Fiber To The Home (FTTH).

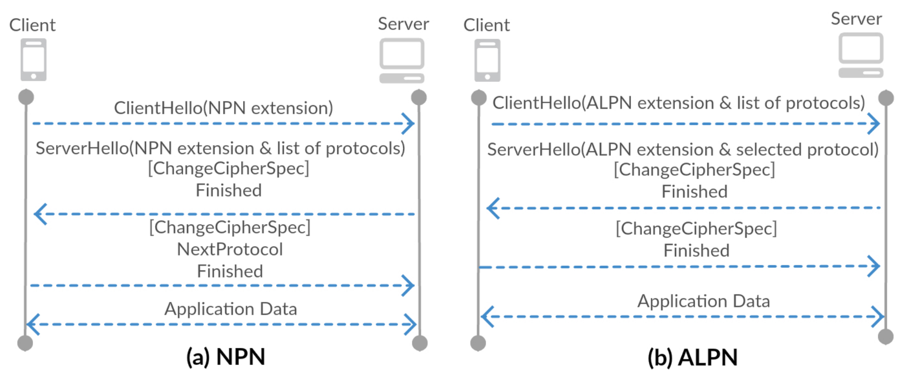


Figure 3.1: TLS handshake with application protocol negotiation.

Although the H2 specification does not require the use of TLS, currently all major browsers only support encrypted H2 traffic. This means that negligible amount of plaintext H2 traffic is expected on TCP port 80. Since we are interested in classifying HTTP flows transported over TLS, we isolate the flows with server port equal to 443, discarding all the remaining flows.

3.1.2 Ground truth

To build the ground truth, we analyze the Next Protocol Negotiation (NPN) and Application Layer Protocol Negotiation (ALPN) information exported by Tstat. NPN and ALPN are TLS extensions that allow clients and servers to negotiate the application protocol to be used in a connection. Since 2016, NPN is deprecated and has been replaced by ALPN, which is an IETF standard. Clients may use both NPN and ALPN extensions, while servers will always use only one of them. The sequence of steps taken in NPN and ALPN negotiations is shown in Figure 3.1.

The negotiation of the application layer protocol using NPN involves the following steps [58]: (i) the client sends the `next_protocol_negotiation` extension in the TLS `ClientHello` message with empty `extension_data`; (ii) the server sends a list of supported application protocols in the `ServerHello` message; (iii) the client selects *one* suitable protocol and sends it back in the `next_protocol` message under encryption before finishing the TLS handshake.

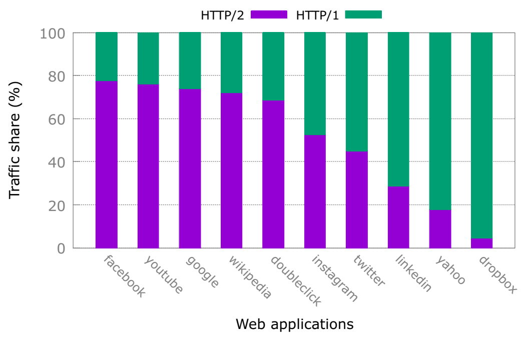
ALPN is instead negotiated in two steps [59]: (i) the client sends a list of supported application protocols using `application_layer_protocol_negotiation` extension in the TLS `ClientHello` message; (ii) the server selects *one* suitable protocol and sends it back in the `ServerHello` message. In this way, the protocol negotiation is done in the clear in a single round trip within the TLS handshake.

Tstat has methods to extract NPN and ALPN information from the TLS handshake. It observes non-encrypted TLS messages and exports the list of protocols offered by clients and servers. We annotate all HTTPS flows with the selected application layer protocol using the NPN and ALPN flags exported by Tstat. Those labels serve as the ground truth to train the machine learning models as well as to validate the classification performance. Remaining features exported by Tstat are discarded from now on. In our datasets the HTTPS flows comprise roughly 68% H1, 27% H2 and 5% other non-HTTP traffic on port 443.

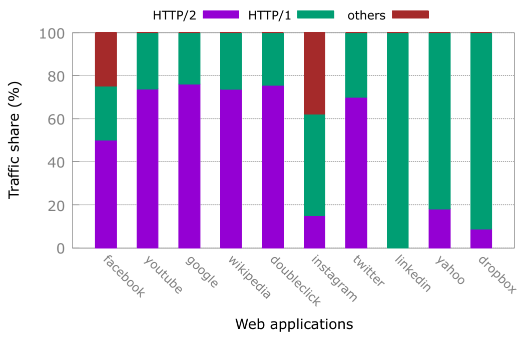
3.1.3 H2 usage evolution in popular web applications

According to [12], the majority ($\approx$ 77%) of the globally used web browsers support H2, therefore the adoption of H2 mainly depends on the support provided by the servers. To find out which HTTP versions are in use by some of the most popular web applications we use information from the TLS SNI extension. This extension allows a client to indicate to the server which hostname it attempts to connect to. Tstat extracts the SNI hostname for each flow and we includes this feature in our ground truth. We aggregate all the flows that belong to the same second-level domain and also assign flows from known CDNs to the respective web application, e.g., *fbcdn* is used by Facebook Inc. Figure 3.2 shows ten popular web applications and their usage of H1 and H2 in June and December 2016. Google, YouTube, DoubleClick ad service and Wikipedia heavily use H2 and it has remained constant in the last six months. Twitter increased the usage of H2 for its services from 45% to 70%. Interestingly Linkedin had around 30% share of H2 traffic in June, but now it has completely switched back to H1. Dropbox still mostly uses H1 and its share of H2 has only slightly increased.

The most interesting change is in Facebook and Instagram traffic. In the past a majority of this traffic was H2, but now only 50% of Facebook and 15% of Instagram traffic constitutes H2. In Figure 3.2b we can see that the 26% of Facebook and 38% of Instagram traffic on port 443 belong to some protocol other than TLS. This is caused by a recent migration of Facebook to a proprietary protocol in mobile devices [60]. The Facebook protocol, called *Zero*, is a zero round-trip (0-RTT) security protocol implemented over TCP and based on QUIC's crypto protocol. At the application level, Facebook still relies on H2 in connections negotiated using the Zero protocol.



(a) June 2016



(b) December 2016

Figure 3.2: H1 and H2 share of ten popular web applications

Facebook claims to have significantly reduced the connection establishment time (and therefore the mobile app's cold start time) using this protocol. Since this traffic does not use standard TLS protocol, it was labeled as other non-HTTP traffic by Tstat during our captures.

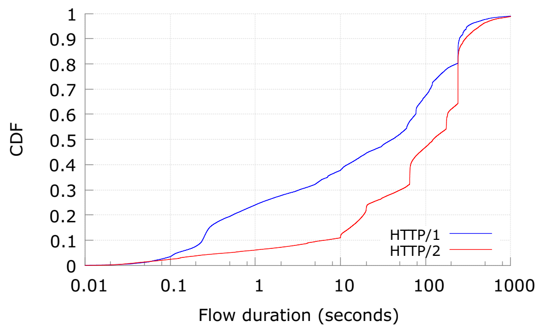
3.1.4 Flow-level characterization

It is expected that H2 will have a different network fingerprint when compared to H1 due to the newly introduced features. For instance, the use of plain-text headers in H1 causes the protocol to (usually) take many round trips to send headers from clients to servers for a single HTTP request. H2 can perform similar operations in a single packet, thanks to header compression, which can reduce the header size by roughly 30% to 80%. Similarly, due to multiplexing in H2, fewer connections can be used to transfer various objects of a web page as compared to H1 [1]. Moreover, it is expected that both browsers and servers have been adapted to exploit new features of H2.

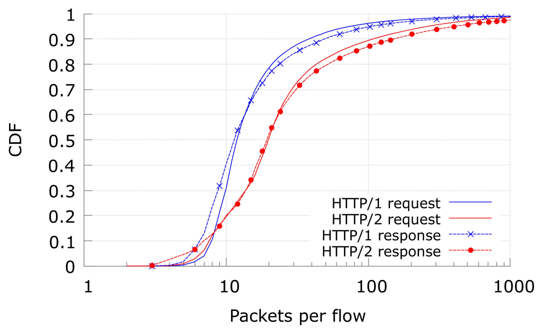
We quantify how all such differences impact HTTP traffic in Figure 3.3 by utilizing our ground-truth traces. We select a subset containing 1 hour of web traffic from June 2016 belonging to the five largest web applications (in terms of number of flows), namely Google, Facebook, Doubleclick, Youtube and Twitter. The subset contains around 700,000 H2 and 300,000 H1 flows.

Figure 3.3a shows the empirical CDF of the flow duration for the two protocol versions. We can see that the majority of H2 flows have longer duration. Around 25% of H1 flows are of very small duration, less than 1s, and another 13% last between 1s and 10s. In H2 only 5% flows are smaller than 1s and another 6% between 1s and 10s.

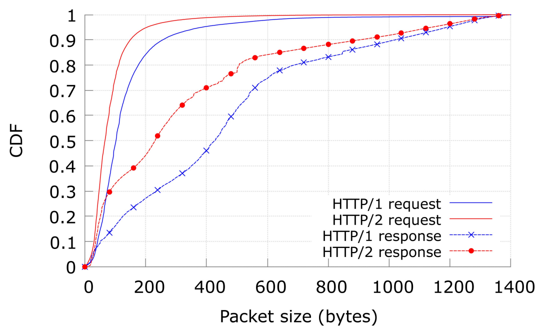
Around 33% H1 and 53% H2 flows have a duration longer than 100s. On average, H2 flows are 36% longer than H1 flows. Furthermore, the duration distribution of H2 flows depicts two pronounced peaks at around 65s and 240s. We investigated it further and found that the flows in the first peak belong to Facebook and those in the second peak belong to Google domains. They are pronounced only in the H2 curve because majority of the flows of Facebook and Google belong to H2 which is evident from Figure 3.2. The reason for the two peaks is probably the different timeout values used at the server side.



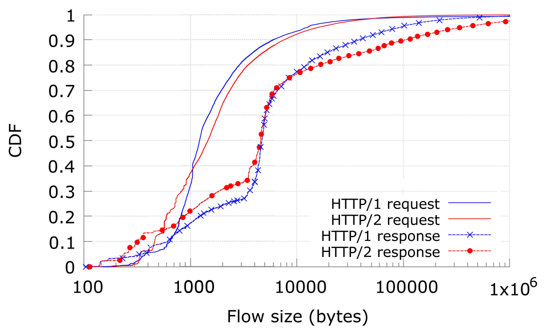
(a) CDF of flow duration



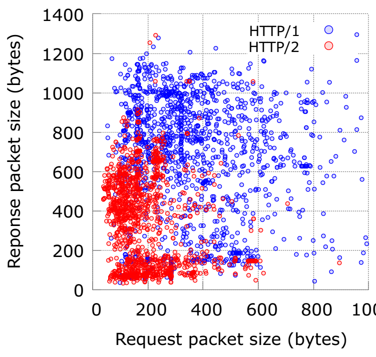
(b) CDF of packets per flow



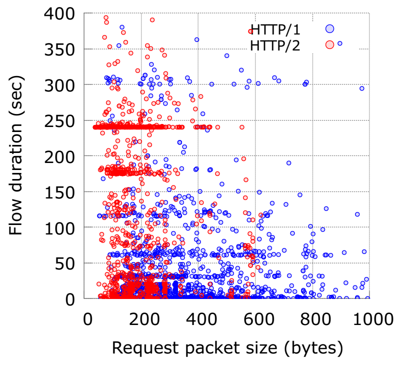
(c) CDF of average packet size



(d) CDF of flow size



(e) Request vs response size



(f) Request size vs flow duration

Figure 3.3: Characteristics of H1 and H2 traffic

We also observe that H2 flows usually have many more packets per flow, likely because they are longer in duration than the H1 ones. This effect is clearly visible in Figure 3.3b. For this figure, the CDF has been calculated separately for the data sent to the server (request) and received from the server (response). The conclusion is however similar for both traffic directions: 50% of the H2 flows have 20 or more packets in the traffic direction, whereas only 22% of H1 flows carry 20 or more packets.

Interesting, while flows are longer and carry more packets, packet sizes of H2 flows are reduced substantially. Figure 3.3c gives the empirical CDF of the *average packet size* of H1 and H2 flows. The average packet size is calculated by dividing the total amount of bytes by the number of packets in a flow. Again, independent lines are plotted for request and response packets. H2 request and response packets are 40% and 29% smaller in size than in H1, respectively. While our data does not prove causality, we conjecture that most of this effect is caused by header compression in H2. Around 80% of H2 and 55% of H1 request packets are smaller than 100 bytes. The advantage of header compression is clearly more prominent in smaller response packets (<600 bytes).

Figure 3.3d quantifies overall effects in flow sizes, by showing the empirical CDF of bytes per flow for the two protocol versions. Here a much more complicated figure emerges, in which many different effects are likely to interplay. Focusing on lines depicting response flow sizes, notice in the tails of the distributions that large flows (e.g., larger than 10000 bytes) are more common in H2 than in H1. The longer duration of H2 flows (e.g., due to client and server implementations) are again a reasonable explanation for that effect. On the other hand, observe that flows with very limited number of bytes (e.g., less than around 4000 bytes) are slightly more frequent in H2 than in H1 as well, probably thanks to the header compression feature of the former.

Finally, we take a random sample of 1500 flows from each HTTP version and depict their properties in scatter plots. Figure 3.3e shows the request and response packet size. In case of H2 most flows are clustered between 400 bytes request size and 800 bytes response size. For H1 the request size is evenly spread throughout the range while the response size is mostly above 600 bytes. H1 and H2 flows look cleanly separated into different clusters. Similarly, in Figure 3.3f H2 flows are grouped together with higher duration and smaller request size while the majority of H1 flows have smaller duration and larger request size. We next exploit these characteristics to derive a method to identify traffic of each protocol without inspecting payloads.

3.2 H2 flows identification using machine learning

It is clear from Figure 3.3 that H1 and H2 traffic are different. This allows us to develop models that are able to classify traffic according to the HTTP versions without payload inspection. Since classification rules are hard to be manually derived from traffic, we opt to follow a machine learning approach.

In machine learning, classification is the task of assigning a *class* to unknown instances, based on features that describe the instances and a sample of instances with known classes. In our case, the instances to be classified are flow records exported by measurement devices, such as NetFlow exporters. The features that characterize the instances are the metrics exported by NetFlow (e.g., client and server IP addresses, bytes and packet counters, etc.), as well as features derived from these data (e.g, average flow throughput). The classes to be assigned to flows are the possible HTTP versions (i.e., *H1* or *H2*) or *Others* for all remaining traffic.

Generally, the classification follows two steps in what is called *supervised learning*. Firstly, a dataset of instances with known classes is presented to the algorithm for *training*, i.e., the algorithm learns relations between features and the classes of the problem. Secondly, using the model built during the training phase, the algorithm assigns classes to unknown instances during the *classification* phase.

3.2.1 Classification algorithms

Our goal is to assess the machine learning approach for the identification of HTTP versions behind network flows. Among the many classification algorithms found in the literature, we take four different options and check their performance with our data. We are not interested in performing an exhaustive assessment of which algorithm is the best one, but instead we want to coarsely select an algorithm that delivers good performance for the particular problem we are solving.

We consider four algorithms: Bayesian Networks (Bayes Net), Naive Bayes Tree (NB Tree), C4.5 Decision Tree and Random Forest. These algorithms are appealing for being fast and for requiring little parameter tuning. For the sake of brevity, we refrain from providing details of the classification algorithms. We use the implementations offered by Weka, and readers can refer to [61] for further information about the algorithms and the toolset implementing them.

3.2.2 Evaluation methodology

We use the datasets described in Section 3.1.1 for training and testing the machine learning algorithms. Flows are labeled as H1, H2 or Others based on TLS handshakes (see Section 3.1.2). This enables us to compare actual and predicted classes and calculate performance metrics.

We evaluate five classic machine learning performance metrics [62]:
(i) *Accuracy* – the overall number of instances that are correctly classified;

$$Accuracy = \frac{\sum True\ positives}{Total\ population}$$

(ii) *Precision* – the number of instances correctly classified as belonging to a particular class;

$$Precision = \frac{\sum True\ positives}{\sum True\ positives + False\ positives}$$

(iii) *Recall* – the number of instances belonging to a particular class that are correctly classified;

$$Recall = \frac{\sum True\ positives}{\sum True\ positives + False\ negatives}$$

(iv) *F-measure*:

$$F\text{-}measure = 2 \times \frac{Precision \times Recall}{Precision + Recall}$$

(v) *Kappa statistic*:

$$Kappa = \frac{P_o - P_e}{1 - P_e}$$

where

- P_o = Observed Accuracy computed in the experiment
- P_e = Expected Accuracy, i.e. probability of accuracy by random chance.

Accuracy is the most popular metric, but it can be misleading particularly when there is large class imbalance. Precision and Recall are measures of exactness and completeness in respect to a particular class, whereas F-measure is a balance between the two. The Kappa statistic is considered a more robust metric because it takes into account the correct classifications that occur by chance.

The training set contains flows that are used to build classification models while an independent testing set contains flows representing the unknown traffic that we want to classify. To create training and testing sets, we use stratified 10-fold cross-validation, in which the original dataset is divided into 10 equal subsets. One subset is used for testing while the remaining are used for training. This process is repeated 10 times and each time a different subset is used for testing. The results are averaged over all repetitions.

3.2.3 Feature selection

The features used for training a classifier are usually key for the classification performance. It is however hard to know upfront which features are informative for classification. Therefore, the training phase in supervised learning is usually accompanied by a *feature selection phase*, in which a large set of features is analyzed for shortlisting those with high discriminating power. This is achieved, firstly, by defining and extracting the largest possible number of features from the input data (e.g., using domain knowledge). Then, feature selection algorithms determine the most informative ones.

From the salient characteristics of the HTTP versions and results presented in previous sections, we create a set with 17 candidate features, including: the overall flow duration and, for both client to server communication and vice-versa, the number of packets, number of bytes, average number of bytes per packet, average packet throughput, average byte throughput, and whether the flows have SYN, ACK or FIN flag set.

Our goal of building a classifier that can operate with NetFlow data constraints the set of features we can extract. In particular, none of the above features requires access to packet payload, nor they require information beyond the transport layer (e.g., from TLS handshakes). Moreover, to keep the classifier lightweight, we do not use features that require correlation of multiple flows (e.g., the number of connections between client and server to load a web page), as it would introduce additional processing steps on raw NetFlow data.

We then apply correlation-based feature subset selection and best-first search algorithms to rank the features. Again, we use the implementation available in Weka [61]. We select the best subset of seven features with highest predictive power for training our classifiers: (i) Number of client bytes; (ii) number of server bytes; (iii) number of client packets; (iv) number of server packets; (v) average client bytes per packet; (vi) average server bytes per packet; and (vii) flow duration.

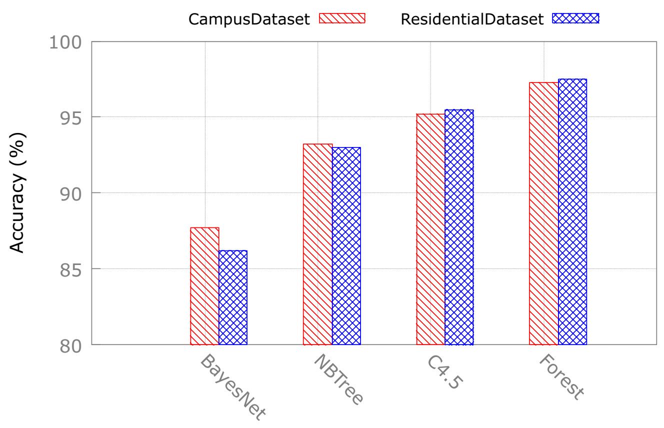


Figure 3.4: Accuracy of the algorithms for different datasets

3.2.4 Classification performance

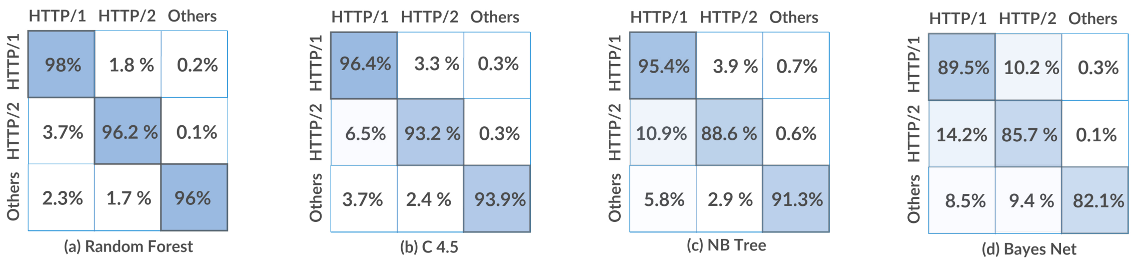
In order to find the optimal size for the training set, we first test the accuracy of learning algorithms using training sets of increasing sizes – ranging from 1 K to 1 M flows. We use the Campus dataset in this experiment. We notice that accuracy improves for all algorithms, provided that at least 100 K flows are in the training set, and it practically stalls for training sets containing more than 500 K samples. Therefore we conclude that a training set of limited size (e.g., 500 K instances) is sufficient for training the classifier.

We then perform 10-fold stratified cross-validation using both Campus and Residential datasets. The results are in Figure 3.4. Random Forest has the highest accuracy – i.e., 97.5% – followed by C4.5 and NBTree having roughly 95% and 93% accuracy. Overall, numbers are very similar in both datasets for these classification models, reinforcing conclusions. BayesNet has the lowest accuracy of 88% in Campus and 86% in Residential dataset. Table 3.1 shows that F-measure (average for the three classes) and Kappa coefficient values are quite high for all algorithms, except BayesNet. This shows the good classification power of the flow-based approach and confirms that the high accuracy is not just by chance.

To further explain results, confusion matrices for the campus dataset are shown in Figure 3.5. Rows mark the actual classes of instances, columns mark the predicted classes, and cells are normalized by number of instances of each actual class – i.e., each row sums up to 100%. Cells in the diagonals thus report the recall of the class, and other cells show how instances are misclassified. Focusing on Random Forest at Figure 3.5a, which has the best results, we notice how it consistently classifies

Table 3.1: Evaluation of the algorithms

Dataset		Campus	Residential
F-Measure	Random Forest	97.3	97.5
	C4.5	95.2	95.5
	NB Tree	93.2	93
	BayesNet	87.7	86.2
Kappa	Random Forest	94.8	94.7
	C4.5	90.8	90.4
	NB Tree	86.4	87
	BayesNet	76.5	70



(a) Random Forest

	HTTP/1	HTTP/2	Others
HTTP/1	98%	1.8 %	0.2%
HTTP/2	3.7%	96.2 %	0.1%
Others	2.3%	1.7 %	96%

(b) C 4.5

	HTTP/1	HTTP/2	Others
HTTP/1	96.4%	3.3 %	0.3%
HTTP/2	6.5%	93.2 %	0.3%
Others	3.7%	2.4 %	93.9%

(c) NB Tree

	HTTP/1	HTTP/2	Others
HTTP/1	95.4%	3.9 %	0.7%
HTTP/2	10.9%	88.6 %	0.6%
Others	5.8%	2.9 %	91.3%

(d) Bayes Net

	HTTP/1	HTTP/2	Others
HTTP/1	89.5%	10.2 %	0.3%
HTTP/2	14.2%	85.7 %	0.1%
Others	8.5%	9.4 %	82.1%

Figure 3.5: Confusion matrices of the algorithms

instances of all classes with recall greater than 96%. Few errors are observed in particular for H2 flows that are mistakenly marked as H1.

3.2.5 Per service performance

There is usually a handful of web applications, such as video streaming and social networking, that generates the majority of traffic in any kind of network. It is crucial to verify that the classification accuracy of the proposed method is due to the characteristics of the protocol versions and not the applications.

We use the SNI field available in the ground-truth to select the top-100 domain names according to the number of flows. We then compute the percentage of flows that are correctly classified for each domain name. Fig. 3.6 presents the accuracy per domain name (left y-axis) of top-100 domain names (x-axis) and their share of flows (right y-axis). The red curve represents the accuracy per domain name, while the blue curve represents the traffic share for each domain. The black horizontal line marks the overall average accuracy. As expected, we see that the top-5 domain names

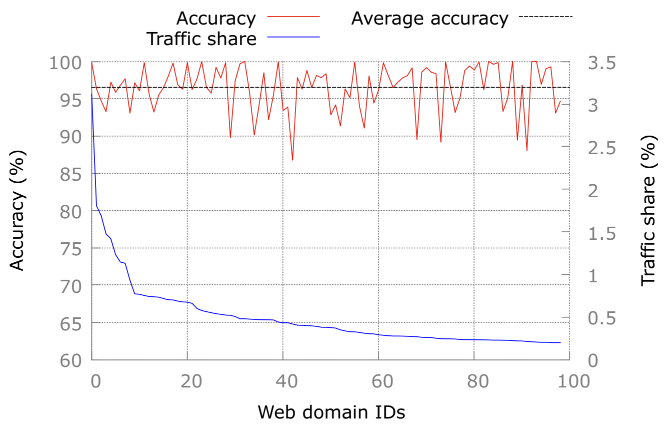


Figure 3.6: Classification accuracy vs traffic share of top-100 domain names

account for the biggest traffic share, with a long tail of less popular domain names. More important, it is evident from this graph that the traffic of most domain names has been classified accurately(>90% accuracy). It verifies that the learning ability of the algorithms is not impacted by specific characteristics of popular applications and the traffic of the applications with very small share is also classified with high accuracy.

3.2.6 Temporal stability

Ground truth collection for training machine learning algorithms is not an easy process but frequent retraining is essential to maintain a certain level of accuracy in most cases because the old model becomes outdated after a certain amount of time. The retraining process can be performed manually under human supervision or automatically [63], but in either case it is not convenient for system administrators to frequently use DPI based methods on network traffic due to legal and privacy reasons. Therefore, the longer the model stays stable after initial training, the better it is. The temporal stability measures how accurate a classification method will remain over time. To test the temporal stability of our system, we used 1 hour of traffic trace from start of June 2016 as training set containing roughly 650K flows to build a classification model with the Random Forest algorithm.

We used seven test sets, each selected from a random day and time from each month from June to December 2016. The details of the test sets and the resulting classification accuracy are shown in Table 3.2. From October onwards we see a big drop of 4-5% in classification accuracy. Upon investigation we found that this is caused by the use of the *Zero* protocol by Facebook and Instagram mobile apps

Table 3.2: Long-term classification accuracy

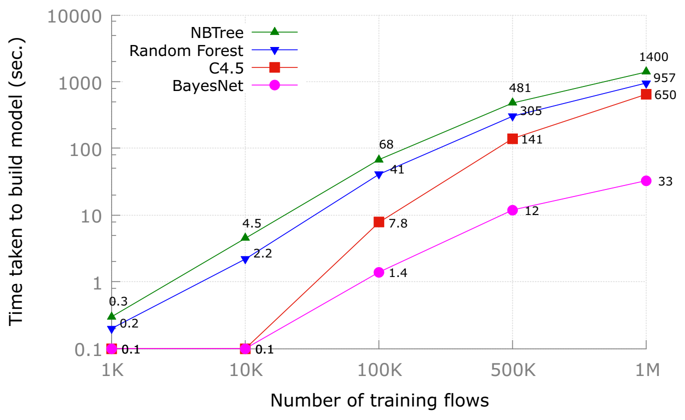
Date	Test Instances	Accuracy (%)
Jun	800K	95.8
Jul	700K	94.8
Aug	400K	95.9
Sep	1.5M	95.5
Oct	2.1M	91
Nov	2.2M	90.4
Dec	1.4M	90.8

which we have explained in Section 3.1.3. Since this traffic does not use standard TLS protocol, it is labeled as other non-HTTP traffic by Tstat. It constitutes a big portion of around 28% of *other* traffic and therefore results in the increase of misclassification. In the future, Facebook is going to subsume this protocol into their implementation of TLS 1.3. To fix this issue for the time being, we labeled this traffic of the *Zero* protocol as H2 instead of *Other* for the December test set. We repeated the classification using the same model that was built from the training set from June and the accuracy increased to 92.3%.

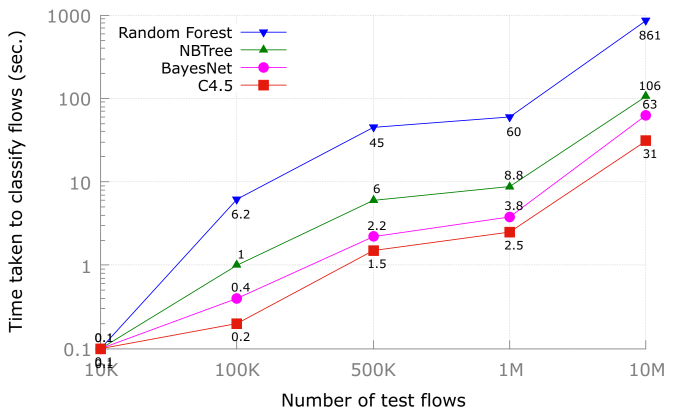
We can see that the classification model built in June maintains a minimum accuracy level of at least 92% for six months, which shows that the system has high temporal stability and can continue to provide accurate classification for several months without the need for retraining.

3.2.7 Spatial stability

The spatial stability measures how classification models perform across different networks. We tested the spatial stability of our system by using datasets from two totally different networks. We trained the Random Forest algorithm using traffic flows from the Campus dataset and used this model to classify flows from the Residential dataset and vice versa. 500K instances each were used for both training and testing set. The accuracy in both cases was around 92%, which is 4% lower than in the scenarios where traffic from the same network dataset is used for both training and testing. This is probably due to usage of different services in these networks. The traffic in the campus network will naturally have more educational and scientific content which might not be there in a residential network. On the other hand, the residential network traffic usually has more entertainment-related content. As the different usage profile is not captured by the model, it may be the reason for the increase in misclassifications. However, the accuracy is still good enough for most



(a) Training speed



(b) Classification speed

Figure 3.7: Comparison of training and classification speed of the algorithms

scenarios and shows that this method can be extremely useful in cases where labeled datasets are not available for a particular network. In such cases a small labeled training set from another network can be conveniently used.

3.2.8 Computational performance

We calculated the time required for training and classification by varying the size of the training set from 1K to 1M entries and the size of the testing set from 10K to 10M. The experiments were performed on a PC with 2.5GHz Intel Core i7 processor and 8GB RAM. Figure 3.7a and Figure 3.7b show the training time and classification time. BayesNet is the fastest algorithm in training and takes only 12s for 500K instances. It is followed by C4.5, Random Forest and NB Tree that take 141s, 305s and 481s respectively. C4.5 is the fastest classification algorithm and takes only 31s to classify 10 million flows ($\approx$ 323,000 classifications/s). It is followed by BayesNet and NB Tree that take 63s and 106s respectively. Random Forest is the slowest algorithm with 861s.

These results show that Random Forest and C4.5 algorithms are best suited for this particular classification problem. C4.5 can be used in online classification systems where high speed is required while Random Forest is a better choice for offline systems as it provides slightly higher accuracy.

3.3 Summary

This chapter presented a comparison of H1 and H2 traffic. We studied basic statistics that are usually exported by NetFlow-like measurement devices and found that differences in the protocols, likely coupled with client and server adaptations to exploit new H2 features, result in very distinct network fingerprints. This calls for a reappraisal of traffic models, as well as HTTP traffic simulation and benchmarking approaches. Differences in network fingerprints motivated us to develop a classifier that is able to identify the HTTP version based on basic traffic features. The proposed method is lightweight and uses only those features that are available in NetFlow data. We demonstrated that decision trees are suitable for this problem, and once the model is built on a relatively small training dataset, it stays valid for several months, thus eliminating the need for frequent ground truth collection and retraining. Our work is a step forward in the direction of understanding modern web traffic. It provides a methodology to identify H2 traffic in flow traces where no information about application layer protocols is available. We expect that our work will improve visibility into network traffic and help in analyzing the adoption of H2 from passive traces.

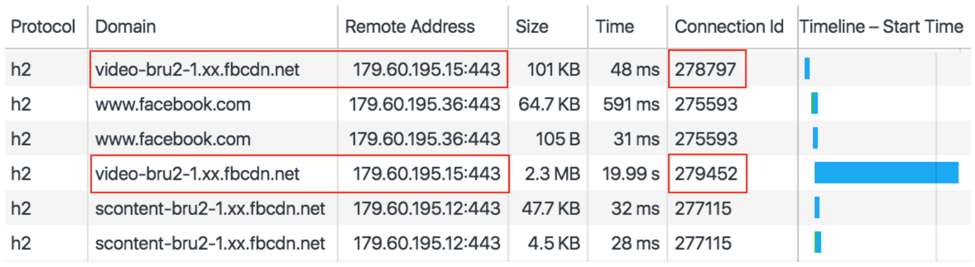
Chapter 4

Analysis of H2 multiplexing

H2 allows the client and server to multiplex HTTP requests and responses on one single TCP connection, while avoiding the problem of head-of-line blocking of H1. Therefore wide adoption of H2 will not only improve the latency of web services, one can also expect that it will have great benefits for the web servers due to a significant reduction of concurrent TCP sessions, and similarly also for network operators due to a reduced resource usage in firewalls, Network Address Translation (NAT) tables, or flow monitors. Although multiplexing over a single TCP connection is one of the main features of H2, nothing actually prevents a client from opening multiple TCP connections to a server. The fact that H2's usage of one single connection can also hurt performance in high loss scenarios, as noted for example by [32], makes this all the more likely. Due to the fierce competition, various web browser vendors aim to provide the most optimal solution with respect to performance.

There is lack of measurement studies to verify that there is no gap between the intended and actual usage of these new H2 features in the wild. This provides the motivation to study the behavior of H2 traffic and examine how H2 multiplexing is used in practice.

In this chapter, we study the behavior of H2 and SPDY clients when establishing and maintaining connections to web servers. We include SPDY because at the time of this study SPDY has much larger share of traffic as compared to H2. This is because H2 has only recently been standardized by IETF and it will take some time for the clients and servers to shift completely to H2. Secondly, since H2 is heavily based on SPDY and the multiplexing feature is exactly the same in both protocols, it is fair to equally treat the traffic of both protocols.



Protocol	Domain	Remote Address	Size	Time	Connection Id	Timeline – Start Time
h2	video-bru2-1.xx.fbcdn.net	179.60.195.15:443	101 KB	48 ms	278797	
h2	www.facebook.com	179.60.195.36:443	64.7 KB	591 ms	275593	
h2	www.facebook.com	179.60.195.36:443	105 B	31 ms	275593	
h2	video-bru2-1.xx.fbcdn.net	179.60.195.15:443	2.3 MB	19.99 s	279452	
h2	scontent-bru2-1.xx.fbcdn.net	179.60.195.12:443	47.7 KB	32 ms	277115	
h2	scontent-bru2-1.xx.fbcdn.net	179.60.195.12:443	4.5 KB	28 ms	277115	

Figure 4.1: Concurrent H2 connections observed using Chrome developer tools

To this end, we collect traffic data in two different operational networks and implement a method to detect concurrent TCP connections established between a pair of client and server using H2 or SPDY. We show that clients indeed open multiple connections and that they actively use those connections. We also show that the number of concurrently opened connections vary with the type of web application. Considering the fact that prior studies have evaluated the performance of H2 under the assumption of a single connection, we believe that this new insight into the real usage of H2 will lead to more accurate traffic models for planning and management tasks.

4.1 Experiments in a Controlled Environment

We start with a simple experiment to illustrate the phenomenon we want to study in this chapter. We visit popular sites from Alexa top 500[1] with various web browsers and observe the established HTTP connections using Developer Tools [64] for Chrome on Linux and Mac OS X, HttpWatch [65] for Firefox on Windows, and Remote debugging for Chrome on Android [66]. These tools allow us to examine all requests and responses between the end points and provide valuable information including server IP, destination port, domain name, connection ID, etc. We observe that, at some point, all of these browsers create multiple connections using H2.

A sample of concurrent connections to a Facebook domain observed using Chrome Developer Tools is shown in Figure 4.1. We can see that there are two parallel connections to the Facebook domain *"video-bru2-1.xx.fbcdn.net"*. We see similar results for other websites and other browsers. However, to quantify this behavior we need to perform large scale tests with traffic generated by thousands of users over a long duration.

[1]`http://www.alexa.com/topsites`

Table 4.1: Summary of datasets

Dataset		Campus	Residential
SPDY	Flows	38 million	160 million
	Bytes	7.3 TB	31 TB
	FQDNs	66,952	268,827
	2nd Level Domains	6,207	8,399
H2	Flows	4 million	21 million
	Bytes	1.3 TB	6 TB
	FQDNs	2,432	4,663
	2nd Level Domains	208	298

4.2 Datasets and Methodology

4.2.1 Datasets

We rely on Tstat [57] to perform passive measurements and collect data related to SPDY and H2 usage in different networks. Tstat monitors each TCP connection, exposing flow level information, including: (i) client and server IP addresses, (ii) timestamps of the first and the last packets in each flow, (iii) the amount of exchanged data, (iv) Server Name Identification (SNI) strings found in TLS handshakes, and (v) the Fully Qualified Domain Name (FQDN) that the client resolved via Domain Name System (DNS) queries prior to opening connections [67]. The latter two fields are instrumental in identifying traffic, since they expose names of websites contacted by the users. We explore those fields to characterize the usage of parallel connections by SPDY and H2 considering different services.

Table 4.1 summarizes our datasets that are collected in a campus and a residential network. At the time of writing, the version of Tstat deployed in the evaluated networks is only capable of identifying SPDY traffic and H2 traffic up to its draft version 14. Thus, we miss deployments relying on the final H2 version standardized by the IETF in 2015.

In total, we have observed 198 million SPDY flows and 25 million H2 flows, during 4 months of data collection (Jan-Apr 2016). Table 4.1 lists the number of server FQDNs contacted by means of each protocol. We can see that users in the monitored networks reach more than 5 k (250 k) FQDNs using H2 (SPDY), or 200 (8,000) names when considering only second-level domains. Thus, our datasets provide a large-scale view in terms of number of users and services. Both protocols together account for more than 45 TB, which is roughly equivalent to 6% of the overall web traffic in the networks, regardless of HTTP/SPDY versions.

Finally, in the rest of the chapter, we will evaluate SPDY and H2 flows always together, since we are interested in knowing whether the protocols open a single connection when contacting servers. Our references to H2 flows implicitly include all SPDY and H2 flows in Table 4.1.

4.2.2 Methodology

This section explains the methodology to detect the parallel connections in H2 traffic.

Log format
We use TCP logs generated by Tstat [57]. Each row in the log corresponds to a different TCP connection and each column represents a specific measure. A TCP connection is started when the first SYN segment is observed, and is ended when either, the FIN/ACK or RST segments are observed or there is no data packet from client or server for a default timeout of 10 seconds after the opening SYN segment or 5 minutes after the last data packet. All the connections for which the proper three-way handshake is not seen or the connection is not correctly closed are ignored.

Domain name extraction
SNI is a TLS extension used by the client in “ClientHello” message at the start of the handshake to indicate which hostname is it trying to connect to. This allows a server to provide multiple certificates on the same underlying IP address on the same port 443 and hence allows multiple websites. Tstat has a field which contains the host name indicated in SNI and we use this field as the domain name in our experiments.

Flow definition
We adopt the classic definition for a flow, which is also implemented by Tstat – i.e., client and server IP addresses, client and server port numbers and transport layer protocol define a flow. When discussing H2 and SPDY, we will however focus only on those flows transported over TCP/TLS, ignoring for instance QUIC and

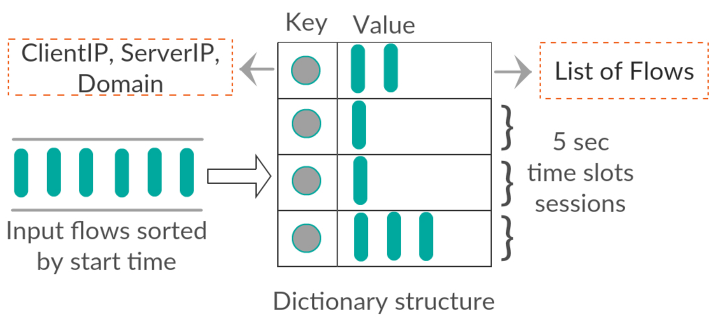


Figure 4.2: Methodology for identification of parallel flows

non-encrypted H2, since the former is by far more common in current traffic.

Parallel flow definition

Our goal is to quantify how many flows are opened in *parallel* by clients when communicating with a server. Therefore, we derive a methodology to estimate the number of parallel flows. We define parallel flows those cases where at least two flows (i) have the same client and server IP addresses, (ii) share the *domain name* coming from SNI strings, (iii) present different client port numbers, and (iv) start close in time. The latter requirement has been imposed to reduce the possible influence of NATs, as we will discuss next.

Algorithm for parallel flow detection

We apply a simple method to mark flows as *isolated* or *parallel* in our traces. We first filter traces to separate only H2 flows. After that, we aggregate flows using the following algorithm which has two phases. In the first phase our algorithm reads a batch of flows sorted by start time. It maintains a set of active "sessions" between clients and servers in a dictionary structure. Each session is represented by an entry in the dictionary, where a session is identified by a key which comprises *client IP address*, *server IP address*, *domain name* and *start time*. The latter is the start time of the first flow of the session. The value stored against the key is a set of all connections (one or more) that belong to the session. When a flow f is read, the algorithm checks whether its key matches an active session. If such a session exists and the start time of f is no later than the start time of the session plus a fixed window size `MAX_GAP`, the flow f is added to that session. Otherwise, the old session (if any) is closed and a new session is started with f as its first flow.

Algorithm 1 Identification of flows belonging to a session

```
 1: Process batches of flows
 2: for each flow in trace do
 3:     key ← clientIP, serverIP, domainName of the flow
 4:     value ← startTime, clientPort of the flow
 5:     if dictionary contains key then
 6:         list[startTime,clientPort] ← dictionary.get(key)
 7:         for each entry in list do
 8:             if flowTime - startTime $\leq$ MAX_GAP then
 9:                 add clientPort to this entry
10:             else
11:                 create new time entry and add clientPort to it
12:             end if
13:         end for
14:     else
15:         add this key and value to dictionary
16:     end if
17: end for
```

Algorithm 2 Measurement of isolated and parallel flows

```
 1: for each key in dictionary do
 2:     list[startTime,clientPort] ← dictionary.get(key)
 3:     for each entry in list do
 4:         if entry contains more than 1 clientPort then
 5:             single++
 6:         else
 7:             multiple++
 8:         end if
 9:     end for
10: end for
```

When the first phase has finished for an entire batch, we execute phase 2 in which we go through each key in the dictionary and get its value. If there are more than one client ports, we count this sessions as having parallel connections, otherwise we count it as having only a single connection. We call a flow *isolated* if it is the only flow in its session, otherwise it is a *parallel* flow. The pseudo-code for the algorithms used in phase 1 and phase 2 are shown in Algorithm 1 and Algorithm 2.

4.2.3 Aggregation gap and impact of NAT

The usage of NAT is very common in both residential and campus networks. Most broadband customers have WiFi-enabled NAT home gateways. In [68] the authors find that 10 % of DSL lines have more than one host active at the same time and IP addresses can be problematic unique end-host identifiers. Our analysis could also be affected by NATs. That is, when a large number of users are connected from a single IP address, the probability that different users will simultaneously contact the same server increases. Connections of the different users would therefore be marked as parallel by our algorithm.

As a first step, we manually remove IP addresses of large NATs in the Campus dataset, based on our knowledge of the campus network topology. For instance, some IP addresses that we know to host WiFi access points relying on NATs are manually removed from the analysis. This step considers around 12.5 million flows (i.e., ≈30% of the raw dataset).

In order to estimate the probability that two users behind a NAT contact the same server at the time (or, at least, close enough that our algorithm would assign their flows to the same session), we examine the User Agents found in our datasets. User Agents are strings containing information about the client browser, which are sent out by clients when contacting URLs using the HTTP protocol. We leverage the HTTP monitoring plug-in of Tstat [57] to export information present in unencrypted H1 headers for every HTTP request seen in the networks. This collection of User Agents has been performed simultaneously to our flow-level captures.

We have counted how often different User Agents are observed active per IP address when dividing our data traces in time bins of 1 min. We have found that 90% of those time bins have only 4 or less different User Agents. Collisions are even rarer: In less than 1% of the time bins two User Agents access the same server (via unencrypted H1).

Note that the User Agent is *not* a unique identification for browsers. It however contains many details, such as the browser and operating system versions. Since our datasets contain only small NATs, e.g., aggregating few users' devices in households, we believe that the odds that different users rely on exactly the same browser configuration and contact a single server simultaneously are negligible.

We therefore adopt a gap of 5 seconds for our algorithm in the experiments, since it is a reasonable value for browsers to open parallel connections when loading a

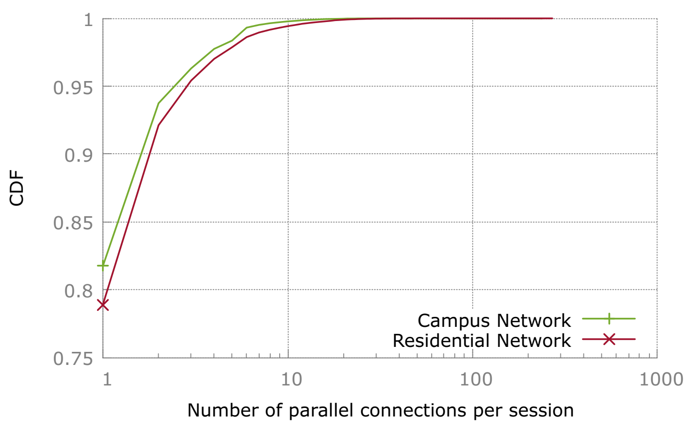


Figure 4.3: CDF of number of flows per session

web page. Compared to the 1-minute bin size studied above, this value is a rather conservative choice and it is likely that we under-estimate the real number of parallel connections.

4.3 Results

4.3.1 Observed parallel flows

We have applied our algorithm to the datasets of the two monitored networks. Table 4.2 gives a summary of the results.

For the campus network, we observe that around two thirds of the studied H2 and SPDY flows are isolated, i.e., the algorithm could not identify parallel flows for them according to the rules described in Section 4.2.2. In contrast, one third of the flows are parallel flows. Figure 4.3 shows the CDF of the number of flows per session. On average, sessions with more than one flow contain around 3 flows.

Similar proportions can be observed for the residential network. Again, around two thirds of the studied flows are isolated and one third of the flows are parallel flows.

Note that browsers that want to send requests to a server need to first discover whether the server supports either SPDY or H2 using NPN or ALPN. Once the connection is established, the browser may remember the supported protocols of the server for a fixed *timeout*, and skip the NPN/ALPN negotiation when opening further connections. Since we only consider flows where the negotiation is observed

Table 4.2: Summary of results

Data set	Campus Network	Residential Network
Ratio of isolated flows	67.1%	65.3%
Ratio of parallel flows	32.9%	34.7%

in the TLS handshake, our results may underestimate the actual number of parallel flows per session.

The reason for the large number of sessions with only one (isolated) flow is not completely clear. One reason could be the aforementioned caching of NPN/APLN negotiations which prevents us from identifying some flows as H2. Furthermore, manual tests have revealed that browsers do not always open parallel connections, suggesting that implementation-specific mechanisms have an influence on the client behavior.

As already stated, one of the main advantages of H2 is its capability to multiplex multiple requests and responses over one single TCP connection. By opening multiple connections to a server, a client is approaching a communication pattern similar to H1.1. Therefore, the results in Table 4.2 raise the question of a possible reason for this behavior. We have discussed our finding with experts from the industry associated with Google Chrome, Mozilla and the IETF HTTP working group. Their responses allow two possible explanations.

4.3.2 Potential reasons for parallel flows

Hypothesis 1 When a browser is oblivious to the protocols supported by the server, it prepares a number (typically 6) of connections that it could use for H1 just in case if the server does not support H2. Later on when the browser discovers that the server speaks H2, it closes the superfluous connections. Exactly how the browser does this may differ between various implementations but this should be done rather quickly so that the extra connections don't consume resources for long.

Hypothesis 2 Although H2 allows to multiplex all requests to a server on one single connection, a web browser still creates multiple connections for performance reasons. Unlike in hypothesis 1, these connections are actively used and not closed quickly. By doing this, the browser avoids the performance problems that might occur with single connections in scenarios with high packet losses. Furthermore, in some situations the TCP congestion window (CWND) negatively affects the performance

Table 4.3: Example of overlapping H2 connections

Flow ID	Start Time	End Time	Duration(ms)
F1	1456822931376	1456822966686	35310
F2	1456822931485	1456822931907	442
F3	1456822931493	1456822975759	44266
F4	1456822931498	1456823080442	148944
F5	1456822931621	1456822931858	237
F6	1456822931622	1456822931860	238

because it places limits on the amount of unacknowledged data which can be in transit between two endpoints at a given time on a single connection. Therefore, the browser chooses to use more connections to avoid performance bottlenecks.

We verify these hypotheses in the following sections.

4.3.3 Degree of time-overlap of parallel flows

Let $\{f_1, \ldots, f_n\}$ be a session consisting of $n > 1$ parallel flows between a client and a server, as identified by our algorithm. If hypothesis 1 is correct, we expect that a session mainly consists of one long flow and several shorter flows (corresponding to the superfluous connections that are closed once the server has been identified as H2-capable). To verify this behavior, we select the longest flow f_l of the session and the flow f_o in the session that overlaps for the longest duration with f_l. We calculate the percentage of overlap by

$$p = \frac{duration\ of\ overlap}{duration\ of\ f_l} \times 100$$

For hypothesis 1, we expect that the sessions in our datasets have a small overlap, while a high overlap would support hypothesis 2.

In Table 4.3 we give an example of six parallel flows. The longest flow is flow F4 and the most overlapping flow is F3 with an overlap of 29.7%.

Figure 4.4 shows the CDF of the degree of overlap for all sessions with parallel flows in our datasets. We can see that in 50% of those sessions, the percentage of overlap is at least 75% in the campus network and 63% in the residential network. Furthermore, in more than 20% of the cases in the campus network and 30% in the residential network, there is nearly complete overlap. In absolute numbers, the average overlap duration in the campus network and the residential network is 58.5 seconds and 29

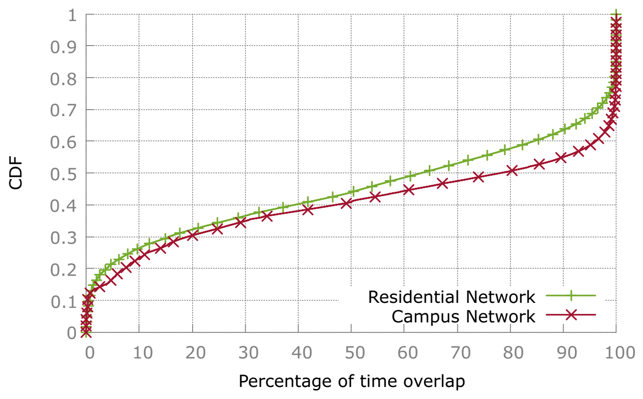


Figure 4.4: CDF of degree of time overlap

seconds, respectively. This suggests that many sessions have at least two parallel connections that are not quickly closed by the browser.

4.3.4 Degree of time-overlap vs flow duration

The large degree of overlap reported in the previous section might be a coincidence. One could argue that most flows are extremely short and hence overlap or, quite the opposite, most overlapping flows are just very long flows that the browser keeps alive and never close.

To get a better understanding of the data, we have divided the sessions according to their longest flow f_l in four categories. Figure 4.5 shows the percentage of sessions with their longest flow having a duration less than 10s, between 10 s and 60 s, between 60 s and 600 s, and greater than 600 s. As expected, sessions with f_l longer than 600 s are relatively rare and there are many sessions with f_l shorter than 10s, but we also see a significant number of sessions with f_l between 10s and 600 s. We also observe a clear difference between flows in the campus network and the residential network. This difference will be discussed later.

In Figure 4.6a and 4.6b, we show again the CDF of the percentage of overlap, this time with the sessions classified according to the duration of their longest flow.

In both networks, we can see that sessions with duration > 60 s tend to overlap for significantly less time compared to shorter sessions. This speaks against the theory that long parallel flows with high overlap are “forgotten” flows. A large fraction of sessions with duration < 10 s have a high overlap but we also see a comparably high overlap for sessions in the range 10s to 600 s.

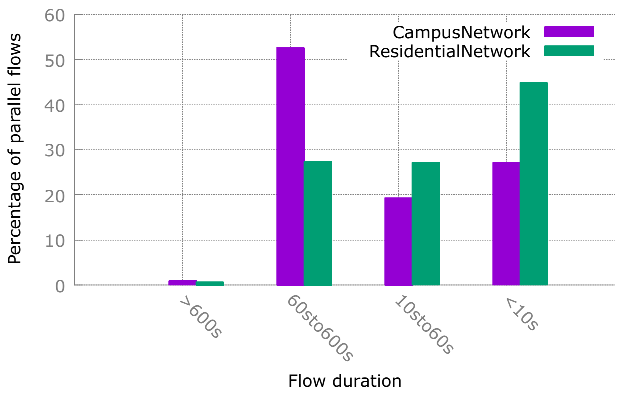
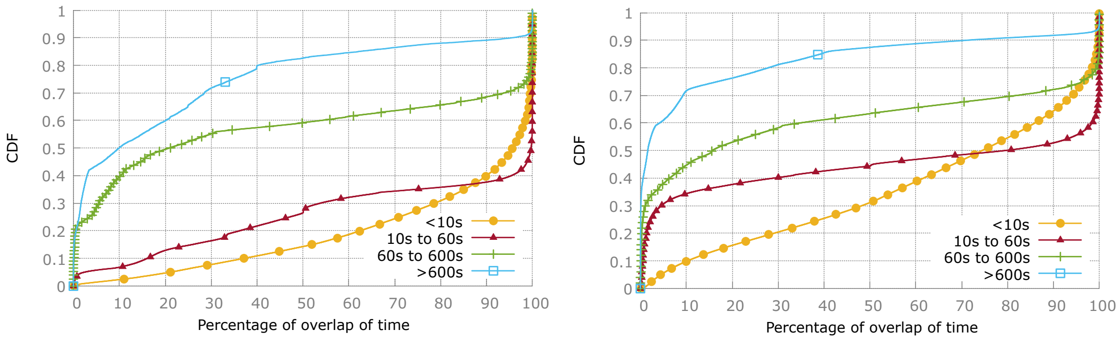


Figure 4.5: Share of flows of various lengths



(a) Campus network

(b) Residential network

Figure 4.6: CDF of degree of time-overlap for flows of different durations

4.3.5 Degree of data-overlap of parallel flows

Although we have shown in the previous experiments that clients open parallel connections with overlapping life times, it does not necessarily mean that the client and the server are actively using these connections for data transmission. To measure how the data is distributed over parallel connections inside sessions with at least two flows, we calculate for each session the ratio

$$r = \frac{total\ session\ size - size\ of\ f_s}{size\ of\ f_s} \times 100$$

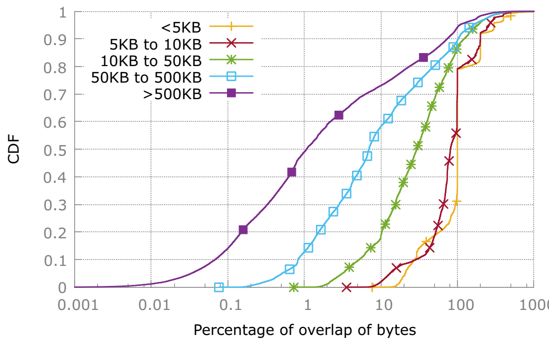
where f_s is the largest flow of the session (in number of bytes) and the total session size is the sum of the sizes of all flows in the session. A ratio of 100% would mean that the largest flow transports the same amount of bytes as all other parallel flows combined. Larger ratios would mean that the largest flow does not carry the majority of bytes.

We show in Figures 4.7a and 4.7b the CDF of r for the sessions in the two networks. We again create different intervals to highlight the behaviour of small and large sessions. We split the data into five groups, based on the size of the largest flow in the session. For the residential (campus) network, 11% (18%) of the sessions fall in the category < 5 kB, 21% (30%) in the category 5–10 kB, 32% (25%) in the category 10–50 kB, 27% (22%) in the category 50–500 kB, and 8% (6%) in the category > 500 kB.

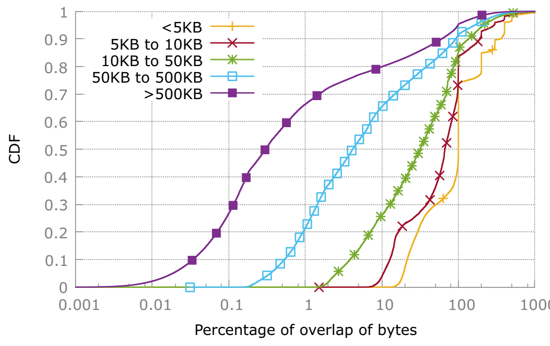
We see a significant number of sessions where bytes are spread over multiple connections. Notice, for instance, that for around 20% of the sessions where the largest flow carries between 10–50 kB we have a ratio $r \geq 100\%$. That means that the remaining connections carry at least the same amount of bytes as the largest connection.

For very small sessions, i.e., largest flow carrying less than 5 kB, we see that the ratio r is very concentrated around 100%. This is not surprising since those sessions mostly contain TLS handshakes and barely any payload. On the other extreme, very large sessions seem to concentrate more bytes in the largest connection. However, even for those cases, we still see a non-negligible number of sessions where bytes are spread over multiple connections.

This suggests that the parallel flows are actively used by the browser, thus hinting that our second hypothesis is true.



(a) Campus network



(b) Residential network

Figure 4.7: CDF of degree of data-overlap for flows of different sizes

4.4 Summary

In this chapter we studied the behavior of clients when opening SPDY and H2 connections in the wild. We quantified how many connections towards a server each client usually opens, and concluded that the number of parallel connections is surprisingly high. Whereas the H2 specification advises browsers to open a single connection and multiplex requests, we found that browsers open on average 3 connections per server.

Researchers evaluating H2 should, therefore, be careful in assuming that implementations follow RFCs' recommendations, even if the protocol standardization is rather recent. Regarding more general implications for the Internet, H2 comes with a promise of reducing the number of connections between clients and servers. This would have a positive impact on servers and on network infrastructure. For instance, it could reduce the workload on devices such as routers, firewalls and intrusion detection systems, which usually keep state per TCP connection in the network. However, our results showed that this promise is rather far from being realized. Although prior studies have shown that there is a reduction in number of connections in H2 when compared to H1, we are still far from the point where only one connection will be required to load a web page.

Note

The network traffic traces that we collected did not enable us to precisely pinpoint the reasons behind this behavior. Later we analyzed the source code of Chromium web browser and found out that it is a client side implementation issue that allows multiple connections to be established with the servers. The Chrome network stack limits number of preconnects to a server that supports H2, but this limit is not implemented if there are simultaneous requests going to the server. In this case up to 6 connections can be opened with an H2 supported server. This issue was found a year after our research by one of Chromium developers in May 2017 and the bug report is available here [69]. We found a similar issue in Mozilla Firefox as well.

Chapter 5

Assessment of H2 and QUIC performance in adverse network conditions

The usage of H2 and QUIC has increased manifold in recent years due to the performance improvement to end users. However, both protocols use a single connection by design, which may result in poor application performance under adverse network conditions, in particular when different protocols compete for limited resources. For example, H2 is known to be particularly vulnerable in WiFi networks with random packet losses. Equally, whereas QUIC uses a different congestion control strategy that reduces the effects of random packet losses, the implications of QUIC's use of a single connection – e.g., during congestion in load-balanced links – are not fully understood yet.

In this chapter we evaluate the performance of browsing using modern web protocols in some adverse network scenarios. We use both active measurements with live websites and emulations in a testbed. We first confirm that H2 (and to a lesser extent QUIC) suffers more than H1 with multiple TCP connections in the tested scenarios. The use of a single connection partly explains the results. We develop a solution which we refer to as H2-Parallel. It is implemented inside the source code of the Chromium browser as an optimization for H2 protocol. We compare H2-Parallel (using 2 TCP connections) with standard H2 (using a single TCP connection), H1 (over TLS), H1C (cleartext), QUIC (using single UDP connection), as well as H2 over Multipath TCP, hereafter called H2-MP. With H2-MP 2 parallel TCP connections are created at the transport layer opaque to the application layer. Note that H2 and QUIC always employ encryption by default.

We make the following contributions:

- We identify scenarios that challenge H2 and QUIC performance, namely (i) packet losses in wireless networks and (ii) congestion in Internet Service Provider (ISP) networks with load balancers, a scenario not addressed in prior work yet.
- We implement *H2-Parallel*, a Chromium-based user agent that fans out H2 requests to a destination over multiple TCP connections.
- We compare *H2-Parallel* against the major web protocols. Our work differs from previous studies by (i) including all relevant web protocols, instead of only a subset of them; (ii) considering real websites and browsers instead of simplistic downloads or TCP transfers, thus giving a view on how users perceive performance while browsing; (iii) testing the latest protocol versions (e.g., QUIC 39).
- We evaluate whether the protocols are fair to one another when competing for bandwidth and find that *H2-Parallel* and QUIC behave similarly for long transfers.

5.1 Considered scenarios

5.1.1 Random packet losses in WiFi networks

Random packet losses in WiFi networks are common under real-world conditions due to various factors like noise and the distance from clients to the access point. There are efforts to reduce the impact of these factors on performance. Optimization techniques have been introduced at both transport and link layer to improve the performance, however the problem cannot be totally eliminated.

5.1.2 Load balancers and congestion in ISP networks

A large percentage of the Internet traffic between source-destination pairs traverses multiple paths due to deployment of load-balancing routers [70, 71] in ISP networks. These routers split packets across multiple paths using techniques like Equal-Cost

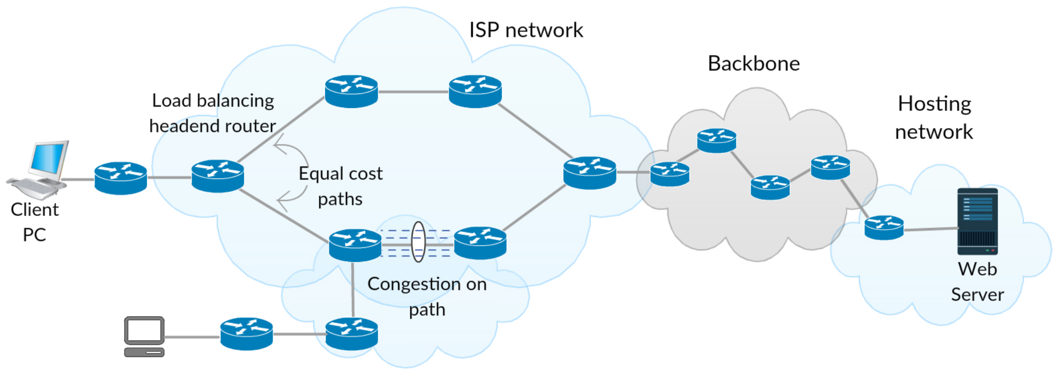


Figure 5.1: Congestion on one of the load-balanced paths in ISP network

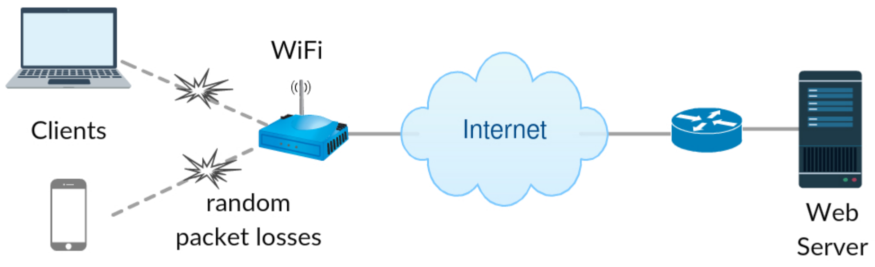


Figure 5.2: Random packet losses in WiFi network

Multipath Routing (ECMP). Since traditional Internet measurement tools like traceroute may fail to identify these paths, alternative tools such as Paris traceroute have been developed to quantify multipath routing in the Internet. Augustin et al. [72] performed a large scale measurement using over 68 thousand destinations and showed that around 70% of paths between source and destination networks traverse a load balancer.

Three different load-balancing schemes exist: packet-based, destination-based and flow-based. Packet-based algorithms distribute all incoming packets evenly on all network paths, e.g., in a round robin fashion. Since different paths can have different delay, this approach may result in massive packet reordering and hence out-of-order delivery of the packets [73]. Therefore it is rarely used in practice. The destination-based scheme routes all traffic destined to the same host over the same path. This scheme can lead to uneven traffic distribution. The flow-based scheme is more popular. It defines a flow using different fields in the packet header, such as source and destination IP addresses, port numbers and protocol. All packets belonging to the flow according to the chosen definition are sent over the same path.

However, optimal load balancing is hard. Figure 5.1 shows a load-balancing headend router with two equal-cost paths in an ISP network where one of the paths is shared by traffic from other nodes. Despite evenly distributing traffic using ECMP on the headend router, the shared link may become overloaded and cause congestion. A number of large scale measurements [74–77] show that congestion predominantly occurs in ISP networks and the described scenario happens quite often. In such a scenario H2 and QUIC may perform poorly as the browser will open a single connection, and that connection may be routed over the congested path. Hence they cannot exploit the underlying path diversity of the network. H1 on the other hand establishes multiple connections which may be distributed across the available paths by the load-balancing routers. Therefore, the performance is not significantly affected while downloading pages with H1 in such a scenario. This is an important scenario, yet it has not been investigated in prior studies. To quantify the extent of severity of this scenario by performing experiments in the Internet is an interesting topic for future work, but is out of scope for this research.

5.1.3 Fairness among competing connections

In the recent years the traffic share of H2 and QUIC has rapidly increased and today, connections belonging to H1, H2 and QUIC co-exist in web traffic. These connections essentially compete for the bottleneck bandwidth. Maintaining fairness is very important for the network as unfairly taking bandwidth share from other protocols may lead to substantial performance degradation for some applications. While it is clear that H1 is unfair to H2 because of its aggressive use of connections, it is more interesting to see how QUIC competes with H1 and H2. In a recent study [55] on QUIC (v. 34) it was shown that QUIC is unfair to 2 and even 4 competing TCP connections. Since QUIC is evolving rapidly with major changes and improvements in each new version, we want to observe whether this behavior has changed in the most recent version (v. 39). Moreover, we want to verify how QUIC compares to our H2-Parallel implementation.

5.2 Methodology

We now explain the design of H2-Parallel and H2-MP, our testbed, and measurement of acPLT and cwnd of TCP and QUIC.

5.2.1 H2-Parallel

H2-Parallel is our Chromium-based user agent that fans out H2 requests over parallel TCP connections to mitigate the negative impact of a single connection on H2 PLT. Our objective is to verify that allowing the user agent to open parallel TCP connections for H2, similar to what most user agents do for H1, would improve the PLT.

Chromium browser maintains a single H2 session per domain, in accordance with the H2 specification. H2 sessions are tracked by a key consisting of the destination host–port pair. Each H2 session goes on a separate socket. In order to allow *two* TCP connections per domain, we have modified Chromium such that it stores two keys for each host-port pair. When issuing a new request, the state of the H2 session is controlled. First we check if we already have two keys for destination host-port pair of the current request. If not, we create a session with the new key and initialize the TCP connection. For each subsequent request, we call a function that returns one of the two available connections and use it for the request.

To keep the modifications as straightforward as possible, we have implemented a basic scheduler that assigns requests in a round robin fashion to one of the two connections. Note that requests assigned to the same connection are still multiplexed by H2. For the server, the two connections look like two regular H2 sessions from the same source IP. Despite this approach being simple, it distributes the requests fairly equally over the two connections.

5.2.2 H2-MP

H2-MP uses MPTCP to create parallel subflows to the servers to load a web page. We use H2-MP to compare the performance of creating parallel connections at transport layer versus application layer(as implemented by H2-Parallel). MPTCP is an enhancement of TCP that allows bandwidth aggregation and improved reliability by utilizing multiple paths simultaneously. It provides the same socket interface as TCP and spreads the data across several subflows without requiring applications or upper-layer protocols to be aware of the multiple paths. An MPTCP connection is initiated with the usual TCP 3-way handshake over one path. The handshake however includes a `MP_CAPABLE` message in the options field of the `SYN`, `SYN/ACK` and `ACK` packets. Further (sub-)flows can be added to the MPTCP session by sending

Table 5.1: Statistics of cloned web pages. The columns *HTML*, *CSS*, etc. show the number of objects of that respective type

Website	HTML	CSS	JS	Image	Other	Total	Size (kB)
Baidu	1	1	1	6	1	10	50
Google	2	1	3	5	1	12	56
Live	2	2	2	2	0	8	262
Twitter	6	1	4	2	3	16	421
Wikipedia	1	1	2	20	1	25	441
Reddit	4	2	5	26	2	39	470
Yahoo	16	13	5	48	4	86	839
VK	4	1	14	3	1	23	920
Taobao	2	2	7	38	4	53	1 320
Instagram	3	1	7	25	1	37	1 409
QQ	15	6	19	115	6	161	1 728
Sohu	13	11	33	167	4	228	2 056
YouTube	8	3	5	113	20	149	2 911
Facebook	1	1	8	123	1	134	3 560
Amazon	5	2	14	41	2	64	3 723

Table 5.2: Statistics of the pages in live websites

Website	Objects	Size	Domains	Connections		
		(kB)		H2	H2-Parallel	H1
Google	17	286	1	1	2	4
Bing	32	421	1	1	2	2
Wikipedia	36	882	2	2	4	4
Mozilla	37	931	2	2	4	8
Poloniex	19	1 028	2	2	4	7
Paypal	64	1 415	2	2	4	12
Instagram	35	1 785	3	3	6	14
Blogger	61	2 061	2	2	4	11
Twitter	18	2 429	2	2	4	5
Facebook	86	4 266	2	2	4	12

`MP_JOIN` in the option field of additional 3-way handshakes regardless of the path used to open the flow.

We use stable release v0.91 of MPTCP and use the *ndiffports* path manager with the number of subflows set to 2, which creates two subflows between the same pair of IP-addresses by modifying the source port. We set the *default* scheduler which starts by sending data on the subflow with the lowest RTT. When its cwnd is full, it sends data on the subflow with the next lowest RTT. This is the recommended scheduler as it is known to provide the best performance. Hence two TCP connections are established between client and server without any modification of the browsing applications and having a complete view of the state of the connections at the transport layer. However, it requires MPTCP-compatible network stacks in the client and in the server.

5.2.3 Mininet testbed setup

We use Mininet [78] version 2.3 to emulate the three scenarios described in Section 5.1. Mininet emulates a large network comprising multiple hosts, links and switches running real kernel and application code. We run Ubuntu Linux kernel 4.1.38 with the stable release v0.91 of MPTCP on the client and server and use Cubic congestion controller on both sides. We use Chromium browser version 60 on the client which supports H1, H2 and QUIC(v. 39). The server node hosts H2O web server[1] which provides an open-source implementation of H2, and quic-go web server which is an implementation of the QUIC protocol in Go[2]. We use Linux's Traffic Control *(tc)* and Network Emulation tools to configure network path characteristics such as bandwidth, delay and packet loss.

5.2.3.1 Emulating ECMP and congestion

For the scenario of ECMP with congestion, we emulate a typical home network shown in Figure 5.1 where a client's home router is connected to the headend router of an ISP with a 10 Mbps link [79]. The link from the ISP to the web server is configured with 1 Gbps bandwidth capacity. The headend router at the ISP performs load balancing using two paths. We emulate the congested bottleneck link in the

[1]`https://h2o.examp1e.net/`
[2]`https://github.com/lucas-clemente/quic-go`

lower path by generating traffic on it with 90% of its bandwidth capacity using iPerf with 8 connections.

We use flow-based routing in the load-balancing router for the reasons explained in Section 5.1.2. Flow-based ECMP routing is not available in the latest MPTCP-capable Linux kernel that we use in our experiments. Therefore, we build a custom Linux kernel and implement a flow-based routing algorithm where the next hop is selected by hashing the flow 5-tuple, i.e., source address (SA), destination address (DA), source port (SP), destination port (DP), and protocol type (PT) of a connection. Our hash function H is defined as

$$H = SA \oplus DA \oplus SP \oplus DP \oplus PT$$

where $\oplus$ is the bitwise XOR function. We calculate $H \bmod 2$ to select either the first path or the second path.

This design avoids any informed decision at the router on how the multiple flows of a single H1, H2-Parallel or MPTCP session are routed through the paths. For instance, the MPTCP or H2-Parallel flows may all take the congested or the non-congested paths during emulations. Obviously, each protocol will react to path choices differently. For instance, MPTCP is able to detect congestion and move traffic to non-congested paths, if at least one subflow is routed to the non-congested path. H2-Parallel instead will blindly schedule requests on the multiple connections.

5.2.3.2 Emulating random losses in WiFi

We emulate the network shown in Figure 5.2. The WiFi link has 7 Mbps bandwidth and 50 ms delay representing realistic network conditions based on large-scale measurement study [80] and also used in prior studies [41, 42]. We perform tests without and with packet loss in the WiFi link. For the latter, we inject random packet losses using *netem*. We use 2% packet loss rate as suggested in prior work [32, 52, 53]. We also perform experiments using other loss rates but the results are not shown due to space limitation.

5.2.4 Measuring page load time

We have selected 15 websites from Alexa's top 100 list and downloaded their landing pages or other publicly available pages onto the H2O server. The selected websites are a mix of social networking, online shopping, news and search. The main characteristics of the cloned pages are summarized in Table 5.1. Chromium browser is used on the client to load the pages from the server. We configure dnsmasq [3] on the client to ensure that all hostnames resolve to the IP address of the server and do not leave the testbed. We have also selected 10 popular H2 enabled websites for *live* experiments listed in Table 5.2. The key requirement for this selection is that *all* of the content must be delivered by the server using H2.

To automate the page loading we create a script that uses *Chrome-HAR-capturer*[4] to connect to the browser via its remote debugging API and load each page multiple times with cold cache. When the experiment ends, an HTTP Archive (HAR) file is created, containing detailed performance data. Our script parses the HAR file, extracts PLT for each of each run and calculates the arithmetic mean of all runs. We load the web pages using H1C, H1, H2, H2-Parallel, H2-MP and QUIC.

5.2.5 Measuring cwnd changes

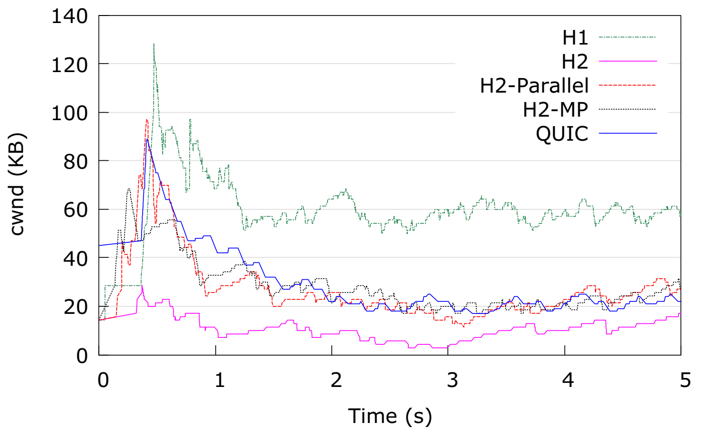
We monitor the changes in the cwnd size for TCP using the *tcpprobe*[5] module. In case of H1 we calculate the sum of the cwnd sizes of all individual connections. In case of QUIC we instrument the source code of quic-go web server to collect logs that allow tracking of the cwnd size on each ACK.

5.3 Measurement Results

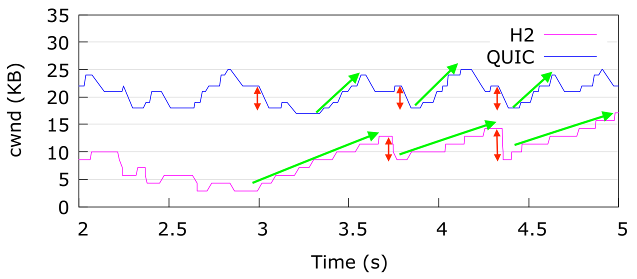
5.3.1 Impact of packet loss on single connection

We start our experiments by determining the impact of packet losses on the performance of various protocols by monitoring changes in cwnd size while loading a web page. We perform an experiment where we host a static web page comprising several JPG images on our web server and load it on the client using Chromium browser via H1, H2, H2-Parallel, H2-MP and QUIC. We configure the bottleneck link with 7 Mbps bandwidth, 50ms RTT, 45 kB buffer size and inject 2% random packet losses into the network path using *tc* and *netem*. We log the changes in cwnd sizes.

[3]`http://www.thekelleys.org.uk/dnsmasq/doc.html`
[4]`https://github.com/cyrus-and/chrome-har-capturer`
[5]`https://wiki.linuxfoundation.org/networking/tcpprobe`



(a) Comparison of cwnd of all protocols



(b) 3-second zoom comparing cwnd of H2 and QUIC

Figure 5.3: Timeline showing the impact of packet losses on congestion window size during a page load

Figure 5.3a shows a 5-second zoom of the cwnd size comparison of the protocols. We can see that H1 has a much higher cumulative cwnd size than others. This is because browsers usually maintain up to 6 parallel connections to each server for H1 transfers and only some connections may be affected by random losses at a time. So the sum of cwnd size of all individual connections remains high. On the other hand, since browsers establish only one connection to the server when using H2, the same connection keeps experiencing the losses resulting in continuous reduction of cwnd. This limits the cwnd to a very small size which in turn results in very low throughput and long page load time. We can also see that QUIC has almost twice the size of cwnd as compared to H2 although both use a single connection and face the same rate of packet losses. There are several reasons for this behavior. First, note that the initial window size in QUIC is around 45 kB (32 segments) while for H2 (and others based on TCP) the size is around 15 kB (10 segments). Second, QUIC has an advantage over H2 because it uses its congestion controller to emulate the behavior of two TCP connections over UDP (in QUIC v39). In other words, in the event of packet loss the cwnd is reduced at half the rate of H2, depicted by red double-headed arrows in Figure 5.3b. Finally, QUIC recovers more quickly from packet losses than H2, which can be observed by a steep upslope as shown by green arrows in Figure 5.3b.

In case congestion in network with load-balancers, the packet losses are dynamic, however, the results that we observe are almost the same and are not shown here. In such a scenario, the single connection of H2 and that of QUIC suffers losses when it is on the congested path, while H1, H2-Parallel and H2-MP are able to use the non-congested path simultaneously for part of their traffic.

5.3.2 Impact of packet loss in WiFi networks

5.3.2.1 Packet loss in live websites

When visiting a live website, several aspects influence the PLT perceived by the user, e.g., delays of DNS queries or of server-side operations to prepare the content. Furthermore, a live website might consist of objects coming from different domains (e.g., due to domain sharding) that are not delivered from the same host.

We study the performance of H1, H2 and H2-Parallel with live websites. We do not perform experiments with MPTCP since none of the top websites support it at the server side. We also skip QUIC as it is only available for Google services and we cannot compare it with other websites. The only parameter that we will vary in our experiments with live websites is the random packet loss rate. To this end, we have selected 10 H2 enabled websites. The page characteristics are shown in Table 5.2, where we give the number of objects per tested page, the total size in kBytes and the number of domains delivering the objects. The latter determines the number of TCP connections opened by the browser (also shown in the table). For each domain Chromium opens one TCP connection with H2, two with our H2-Parallel implementation, and up to six connections with H1. Note that the pages loaded from live websites are not exactly identical to those we cloned onto our testbed.

The RTT to the web servers is in the 15–165 ms range. We load each page 15 times with an empty cache. Figure 5.4 shows the average PLT (and its standard deviation) when using H1, H2 and H2-Parallel without packet loss and with 2% packet loss.

Focusing on Figure 5.4a, we notice how the performance of H2 is mostly better than H1. Whereas differences are not extremely large, these results are significant if we consider the number of TCP connections opened by the browser for each protocol (see Table 5.2). Our implementation of H2-Parallel achieves similar performance as H2 when network conditions are good, although a small overhead for opening and managing the extra TCP connections are visible in some cases.

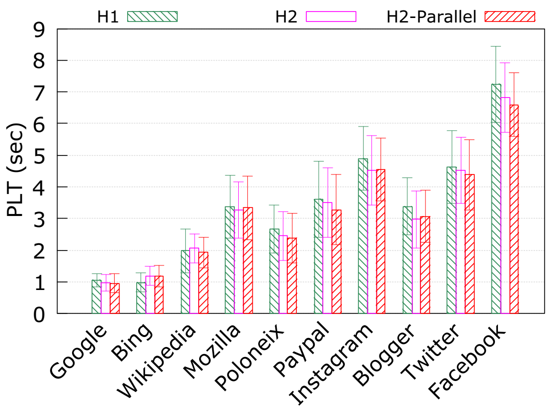
Obviously, the PLT increases significantly when packet loss is introduced – see Figure 5.4b (note the different scale of the *y*-axes). However, the increase of PLT with H1 is less pronounced than with H2, thanks to the use of multiple TCP connections by the former. H2 suffers severely under the packet loss. We notice how the PLT for Facebook reaches almost 12 s on average for H2, whereas it is around 8.5 s for H1 with 2% packet loss. The figure also shows that H2-Parallel achieves similar performance to H1 thanks to its second TCP connection. We are able to achieve 53% reduction in PLT on average for all websites by using H2-Parallel.

Figure 5.4c shows the number of connections established with the server to load a website (depicted with different colors) and the percentage of bytes transferred with each connection (depicted by the size of the bar of each color) using H2 and H2-Parallel. We can see that with H2, while in some cases multiple connections are established due to multiple domains on the server side, most of the data (83% on average for all tested websites) is still transferred using only one connection. In case of H2-Parallel, a single connection carries 52% of traffic on average, thus distributing the load more evenly across multiple connections and reducing the probability of a single connection experiencing packet losses repeatedly.

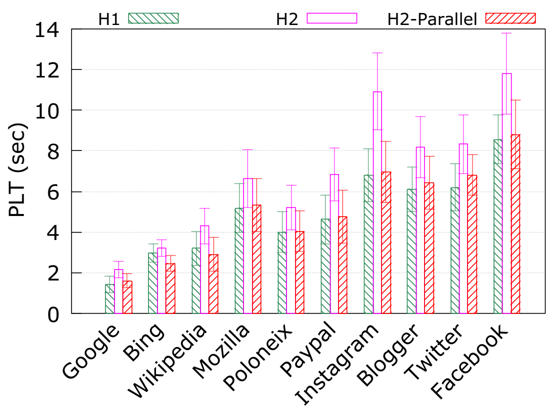
5.3.2.2 Packet losses in emulated environment

We perform emulations in a controlled mininet environment for reproducibility of results. In this experiment we measure the effect of random packet losses on the performance of the different web protocols using cloned web pages. Since the web server is under our control we can test QUIC and MPTCP and compare them with other protocols for the same pages, which was not possible with live websites. We load each page 30 times with empty cache using automated scripts. Note that the performance of H1 and H2 in scenarios with packet losses in WiFi networks has been studied in [31–34, 41]. We confirm results from the previous works and evaluate to what extent H2-Parallel and H2-MP improve performance.

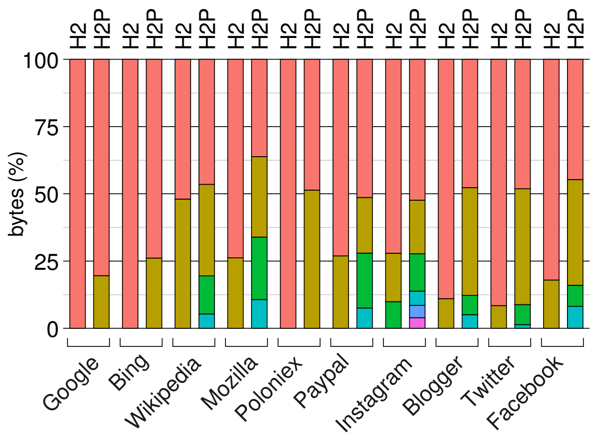
In the following, box-whisker plots show the arithmetic mean (small circle), the median (middle horizontal line), the first and third quartiles (upper and lower box edges) and the minimum and maximum (whiskers) of the PLT over all 15 web pages. In our experiments, minimum and maximum will typically be several seconds apart.



(a) No packet loss



(b) 2% packet loss

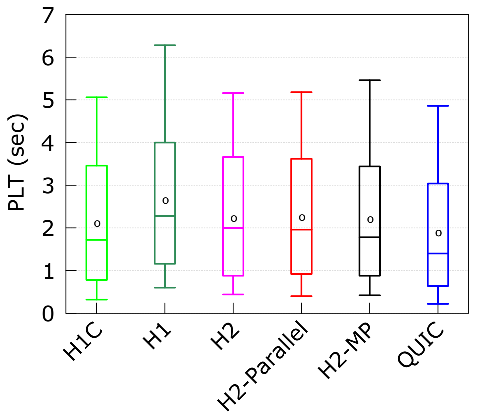


(c) Percentage of data transferred on each connection with H2 and H2-Parallel (H2P)

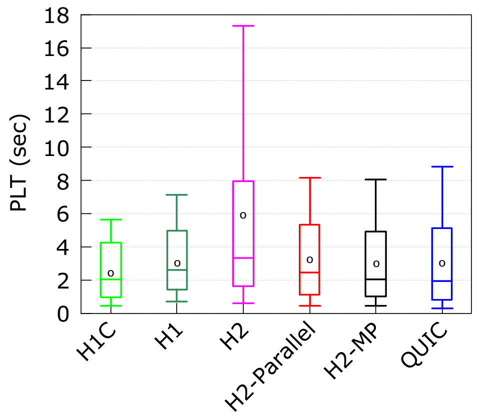
Figure 5.4: Performance comparison of live websites with and without injected packet loss.

This is due to the large difference in characteristics of the selected web pages, i.e., small web pages like Google have very small PLT (represented by the lower end of the whisker) while large web pages like Facebook or Amazon have very large PLT (represented by the upper end of the whisker).

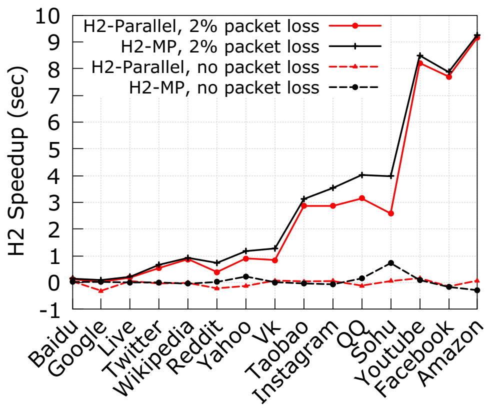
Figures 5.5a and 5.5b show plots of the PLT measured over all web pages without and with 2% packet loss, respectively. As expected, H2 performs better than H1 when there is no packet loss, and H2-Parallel and H2-MP do not show significant differences under good network condition. However, *with* packet loss, H2's PLT increases greatly while the other protocols see only a moderate increase by around 20%. Again, H2 is affected the most due to the use of a single TCP connection by the browser. Another disadvantage using H2 is that when there is a packet loss event,



(a) WiFi with 0% packet loss rate



(b) WiFi with 2% packet loss rate



(c) H2 speedup against a single connection

Figure 5.5: Performance comparison in WiFi network emulation. Note differences in y-axes

all streams get stalled until packet recovery due to the in-order delivery guarantee of TCP. Using QUIC, only the stream related to that packet gets blocked while others keep functioning normally. QUIC also maintains larger cwnd size as compared to H2 as shown in Figure 5.3a. Due to these reasons its performance is not affected as severely as H2.

Both H2-Parallel and H2-MP are able to reduce the performance penalty of packet losses and achieve a performance similar to H1 by increasing the cumulative cwnd size.

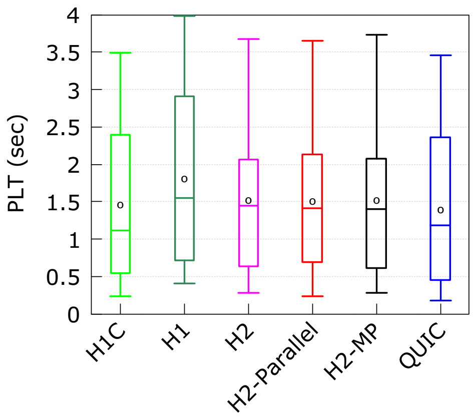
Figure 5.5c shows the average speedup (in seconds) that H2-MP and H2-Parallel achieve relative to regular H2 for each tested website (sorted by their size, with the smallest on the left). It can be seen that the speedup relative to regular H2 with packet loss is particularly pronounced for large web pages.

5.3.3 ECMP and network congestion in emulated environment

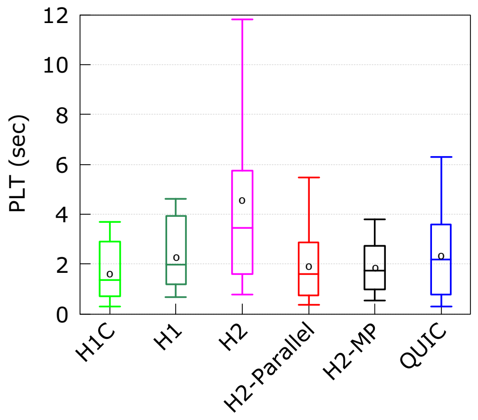
We now emulate the ECMP scenario with Mininet using H1C, H1, H2, H2-Parallel, H2-MP and QUIC. For each considered protocol, the client loads each web page 30 times with an empty cache. Remember that we do *not* actively control how the multiple flows are load-balanced in the available paths to emulate realistic scenarios. That is, in some experiment rounds, the multiple MPTCP or H2-Parallel flows may both take the congested or the non-congested path by chance. We can see in Figure 5.6a that, without congestion, H2 performs slightly better than H1 thanks to its various optimizations and new features. Not a surprise, H1C is faster than H1 because of the TLS overhead in the latter. Using two connections (H2-Parallel and H2-MP) in good network conditions brings no noticeable advantage while QUIC performs slightly better than others in the mean and median case.

However, the situation changes drastically in the presence of congestion. H2's PLT shoots up, with some pages taking as much as 12 s on average and up to 20 s in the worst case (not shown) to be fully loaded. H1C, H1, H2-Parallel and H2-MP are only slightly affected thanks to the parallel connections, which may be routed in the two available paths. In fact H2-MP performs the best as it can route the traffic away from the congested path on the fly and move it to the good path, which is not possible with any other protocol. QUIC is not affected as severely as H2 because its congestion controller reduces the cwnd size less aggressively when dealing with packet losses due to congestion. However, it still performs worse than H2-Parallel and H2-MP because in many cases it cannot take advantage of the non-congested path due to the use of a single connection. QUIC is 34% slower than H2-Parallel and H2-MP on average for medium and large websites but the situation is different in case of small websites. QUIC loads small websites quite fast due to 0-RTT connection establishment, and even with a single connection it performs slightly better than H2-Parallel and H2-MP. In fact for small websites creating parallel connections doesn't provide much benefit because most of the data is already transferred on the first connection before the second connection gets its turn.

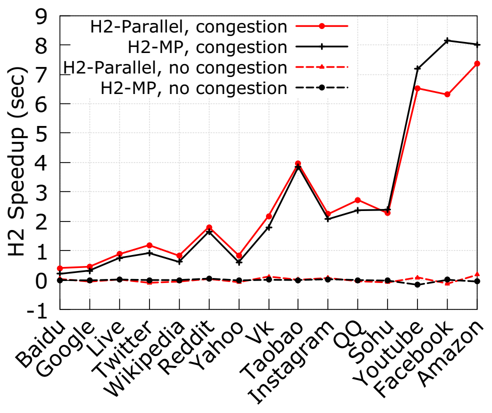
Finally, in Figure 5.6c we can see that there is no speedup using H2-Parallel and moderate speedup using MPTCP when both network paths are congestion-free. When congestion is created on one path, both H2-Parallel and MPTCP achieve impressive speedups particularly for large pages.



(a) No congestion on any path



(b) Congestion on one path



(c) H2 speedup against a single connection

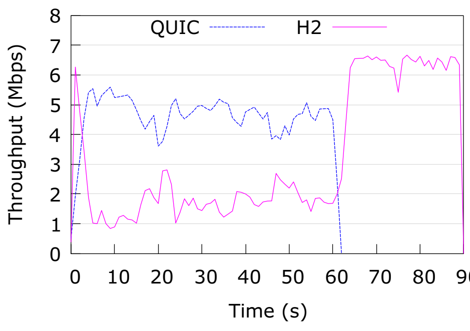
Figure 5.6: Performance comparison in emulation of ECMP routing in network. Note differences in y-axes

Among the tested protocols, QUIC looks the most promising. Although it does suffer from performance degradation in the above scenario, we believe that using two connections with QUIC (similar to H2-Parallel) instead of emulating two connections using the congestion controller could improve QUIC performance in this scenario.

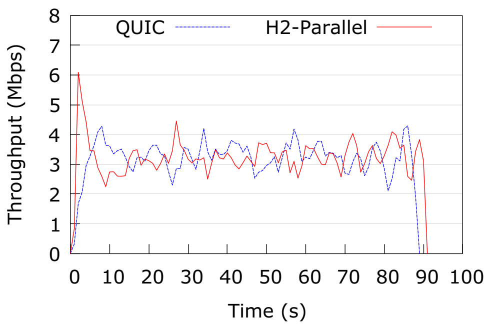
5.3.4 Fairness comparison

So far we have measured PLT of various protocols while running in isolation. Now we investigate their behavior while competing with one another. For this experiment we use two clients connected to two servers using the same 7 Mbps bottleneck link. We host a synthetic web page with large JPG images on the web servers and both clients load the web page at the same time, but using a different protocol. H2 and QUIC use a single connection, H2-Parallel uses two connections while H1 uses four connections to load the page. We measure the throughput of each protocol using tcpdump. Figure 5.7 shows the bandwidth share of competing connections per protocol pair.

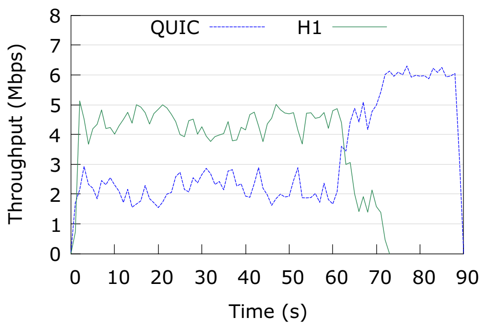
In Figure 5.7a we can see that QUIC gets twice as much bandwidth share as compared to H2 as it emulates two connections using its congestion controller. It has been shown in a recent study [55] that QUIC v34 is unfair to TCP even when competing against 2 or even 4 TCP connections. However, we do not observe such behavior in our experiments with QUIC v39. Figure 5.7b clearly shows that H2-Parallel using 2 TCP connections and QUIC v39 get an equal share of bandwidth, as expected from QUIC's congestion controller. Figure 5.7c shows that H1 using 4 TCP connections is more aggressive than QUIC and thus has an unfair advantage when competing with both H2 and QUIC.



(a) QUIC competing with H2



(b) QUIC competing with H2-Parallel



(c) QUIC competing with H1

Figure 5.7: Comparison of fairness of bandwidth share among competing connections of different protocols

5.4 Summary

In this chapter we presented a performance evaluation of H2 and QUIC in adverse real-world scenarios. We confirmed that H2 exhibits suboptimal performance in such scenarios and suffers from unfairness when competing with other protocols due to its use of a single TCP connection. Results showed that QUIC is not as severely affected, because it implements a congestion controller that emulates the behavior of two TCP connections over UDP. We implemented and evaluated a solution to improve H2 performance, called H2-Parallel, which lets browsers open multiple TCP connections for H2 as they usually do for H1. Our experiments with popular live websites as well as controlled emulations show that H2-Parallel reduces PLT when compared to H2 over a single connection. In a scenario with around 2% of packet loss, H2-Parallel with only two parallel connections reduces the average PLT of H2 by 55%, and practically makes the performance of H2 similar to what is obtained by H1 with several parallel connections. H2-Parallel has interesting advantages: it presents performance similar to QUIC, profits from parallel Internet paths similar to H2-MP, and requires changes only in the client browser thus easing deployment. Although QUIC is not affected by packet losses as severely as H2 over TCP thanks to its new congestion control strategy, QUIC can still benefit from the use of parallel connections in scenarios with network congestion.

Chapter 6

On the Causes of Performance Issues of QUIC over Wireless Links

The exponential growth in adoption of mobile phones and widespread availability of wireless networks has caused a paradigm shift in the way we access the Internet. Today WiFi networks are commonplace across the board including homes, offices, shopping malls, restaurants, hospitals and university campuses. Wireless Mesh Networks (WMN) are also getting popular and some examples include Guifi.net, MadMesh, Merkai and Google WiFi. Thus wireless traffic continues to grow at an unprecedented rate. In a recent Cisco white paper [81] it is predicted that wireless and mobile device traffic will exceed that of PCs, and will comprise more than 63 percent of total IP traffic by 2021.

However wireless networks are prone to errors due to obstacles, moving objects, whether condition, noise and interference with other wireless sources, etc. Nonetheless, technological advancements have allowed to tackle these issues and provide better quality of experience. On one hand, WiFi technology, which is based on the IEEE 802.11 standards has undergone a lot of improvement in the recent years [82]. For instance, the 802.11n standard, which is predominant nowadays, allows coding schemes (MCS) that support data rates up to 600 Mbps. It also includes a high throughput enhancement called frame aggregation, which consists of combining two or more data frames into a single transmission, thus reducing the fixed overhead associated with each frame transmission. On the other hand, there are efforts to improve the transport layer to better deal with the inherent issues of wireless networks. TCP is the most important most widely used transport protocol and it is not a surprise that there are vast amounts of research on TCP improvements. TCP has reigned the Internet for the last four decades and only recently we have seen the development of a new low-latency transport protocol called QUIC that aims to provide better user experience. QUIC started as an experimental protocol designed by Google and has emerged as a serious alternative to TCP. In a recent measurement study [83], it was estimated that around 7% of the Internet traffic is carried by QUIC.

Both academia and industry are showing great interest in QUIC research. Two of the most popular web services, namely Google Search and Youtube use QUIC and Snapchat has started to adopt QUIC. IETF has chartered the QUIC working group to work on standardization and HTTP/3 which is going to be the third official version of HTTP protocol is expected to use QUIC instead of TCP. Keeping in view the exceptional growth in wireless traffic and the expeditious adoption of QUIC, our objective is to perform a comprehensive study of QUIC in WiFi networks. To this end we collect and analyze QUIC and TCP traffic in a production WMN. We explore the interaction of QUIC transport with the advanced features of 802.11 such as frame aggregation. We carefully examine how delay variations which are common in wireless networks impact the congestion control and loss detection algorithms of QUIC. Based on our experimental findings we develop an optimized version of QUIC by making the necessary changes to the source code. We call this version BQUIC and evaluate the performance improvement it provides.

6.1 Methodology and Evaluation

In this section we describe our methodology to evaluate the performance of QUIC and TCP in WiFi networks. We provide details about the testbed, the way in which the server and clients are organized along with their specifications, and the performance metrics used in the experiments. We compare throughput of TCP and QUIC using different wireless links in a production network by analyzing the collected packet at both client and server side.

6.1.1 Wireless community network

We perform experiments in a production wireless community network deployed in a neighborhood of the city of Barcelona (Spain) called Sants [84]. The network was started in 2009 and in 2012 was joined by nodes installed at Universitat Politècnica de Catalunya (UPC) within the EU CONFINE project [85]. The network is operative since 2009. The nodes use the linux/openwrt [86] based distribution provided by the Quick Mesh Project (QMP) [87], which runs the BMX6 mesh routing protocol [88]. From now on we will refer to this network as *QMPSU*. QMPSU is part of a larger community network started in 2004, which has more than 30.000 operative nodes called Guifi.net [89]. At the time of writing QMPSU has around 80 active nodes. Fig. 6.1 shows the geographic location of active nodes and links, using distinct colors to represent wireless links configured with different channels. In QMPSU there are 2 gateways that connect QMPSU to the rest of Guifi.net and the Internet.

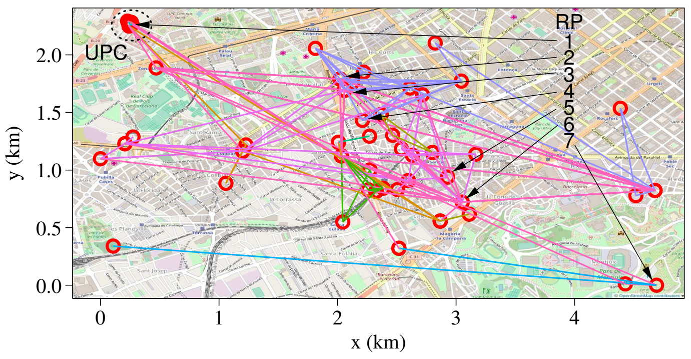


Figure 6.1: Geolocation of clients in QMPSU topology. Colors indicate links configured in the same WiFi channel

QMPSU is 802.11an-based and the most common hardware is the Ubiquiti NanoStation M5, equipped with a sectorial antenna and running QMP firmware. There are also a number of point-to-point links using Ubiquiti parabolic antennas running the original manufacturer firmware. QMPSU also has a live monitoring web page updated hourly. A detailed description of QMPSU can be found in [90], and a live monitoring page updated hourly can be accessed on-line [91].

QMPSU has been deployed by its own users. Its unplanned spread out using heterogeneous WiFi devices in an urban area has produced a high diversity on the quality of the links. Thus, it offers a very realistic testbed to evaluate the performance of QUIC under a variety of conditions.

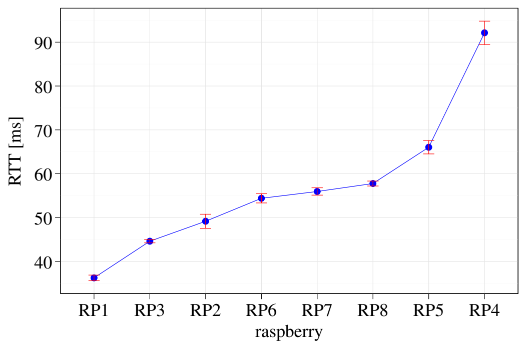


Figure 6.2: Mean RTT between various clients and the server

Table 6.1: Characteristics of the client locations

RP	Name	W-hops
$RP1$	UPC	0
$RP2$	BCCanBruixa20Rd6	1
$RP3$	BCNevaristoarnus5Rd3	2
$RP4$	BCNMelciorPalau62	5
$RP5$	GS26gener10	3
$RP6$	GSgV-rb	1
$RP7$	BCNJardiBotanicSants186	5
$RP8$	UPC-EETAC	5

6.1.2 Experimental setup

We deploy eight Raspberry PI 3 (RPi) clients attached using the Ethernet port, one in UPC and the others to the premises of different volunteers across the QMPSU mesh network. Clients have quad-core ARM Cortex-A53 CPU and 1 GB RAM and run Debian 9. Chromium browser v69 is used on the client node in headless mode to load a web page from the server in the campus of Politecnico di Torino (Polito), Italy. The server has a dual-core Intel Core i5 CPU running Ubuntu 16.04. The default congestion controller in the server is TCP CUBIC. We use Nginx web server for TCP and quic-go server supporting QUIC v43.

The clients in the WMN first reach the gateway node located in UPC using wireless links and then reach the server in Polito via the Internet. Fig. 6.1 shows the geographical location of the raspberries, marked as *RP*. One of the raspberries, RP8, is not shown in Fig. 6.1 because is located geographically very far away in another campus of UPC located in Castelldefels, a village around 15km from Barcelona. There is a WiFi point-to-point link of around 15km to reach this campus. Tab. 6.1 shows the number of wireless hops (W-hops) in the WMN taken by the traffic downloaded from the server in Polito to reach the RPis. RP1 is located inside UPC and is directly connected to the fixed network without any WiFi links. Note that many of these hops use different channels and thus are not interfering with each other. Figure 6.2 shows the average RTT between the clients and server and 95% confidence intervals. The RTT was measured averaging over a large number of pings. The figure shows that RTT is in the range from 36 ms for RP1 (the raspberry located at UPC) to 90 ms for RP4, the raspberry in the WMN having the worst connection.

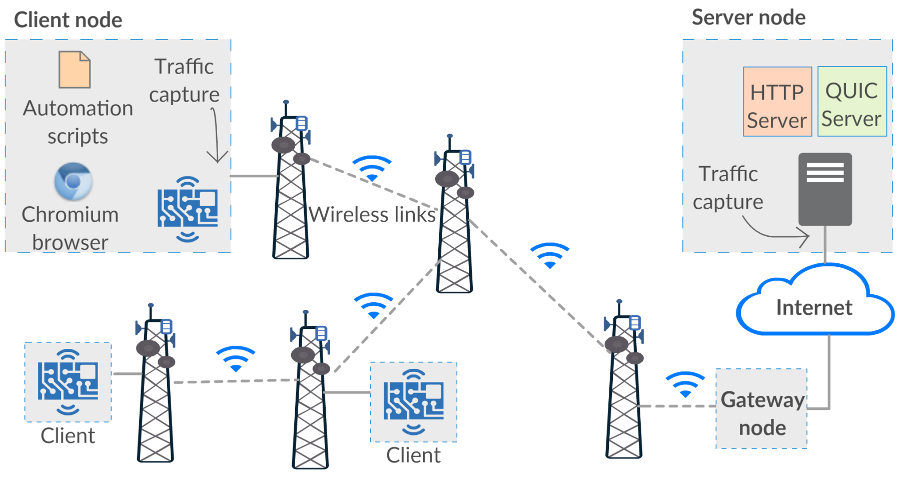


Figure 6.3: Experimental setup

6.1.3 Measurements and performance metrics

We load a 10 MB file hosted on our web server in Turin from the clients in Barcelona using three transport protocols, i.e., TCP, default Chromium QUIC implementation and our optimized BQUIC implementation which we will describe later in Sec.6.3. To automate the experiments we deploy scripts that connect to remote clients in the WMN hourly, to load the pages sequentially using every protocol under test. Traffic is captured at both client and server using tcpdump and saved to a separate timestamped file. The experiments where done during the first week of February, 2019.

We parse the captured traffic from each individual file and calculate various average metrics such as the total page transfer time, throughput, and packet inter-arrival time, etc. Each average was calculated using at least 100 points. Throughput is computed by dividing the amount of bits sent in the payloads of the connection over the time of the transfer, ignoring connection establishment time. We also analyze the time sequence of data segments and ACK packets. We log TCP cwnd using

Table 6.2: Comparison of QUIC and TCP performance under stress

	TCP Throughput (Mbps)			QUIC Throughput (Mbps)		
Device	Normal	Stressed	Decrease	Normal	Stressed	Decrease
RP2	58.5	53.47	8.6%	38.8	17.7	54.3%
RP7	31.8	29.6	6.9%	21.2	16.6	21.6%
RP5	10.2	9.89	3%	7.3	6.1	16.4%

tcpprobe kernel module[1]. In case of QUIC we instrument the web server to log the cwnd size on each ACK.

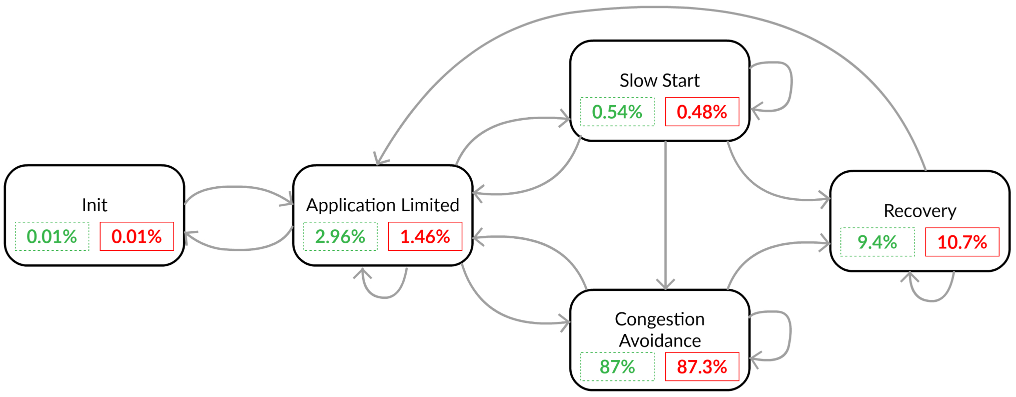


Figure 6.4: QUIC v43 state transitions on RPi (green dotted box) vs PC (red solid box). The numbers show the percentage of time spent in each state. The behavior is very similar on both devices

6.1.4 Impact of userspace implementation of QUIC on RPi

TCP is implemented in the OS kernel while QUIC is implemented in userspace. This can have negative impact on performance, particularly on resource-constrained mobile devices. With the exponential growth of smartphones in recent years, this aspect becomes even more crucial. This impact was first shown experimentally by Kakhki et al. [55] by developing a state machine for QUIC. They showed that on a mobile phone (MotoG) QUIC v34 spends 58.84% of the time in application limited state as compared to only 7.05% on a PC. In application limited state the cwnd is not increased any further because the client may be slow and unable to process the incoming packets at the sender's rate.

Since we use RPis that have similar hardware specification to modern smartphones, it is crucial to make sure that its hardware does not limit the performance of QUIC.

[1]https://wiki.linuxfoundation.org/networking/tcpprobe

To this end, we develop a state machine for QUIC by instrumenting the source code. We load a web page from the server over QUIC using two clients, an RPi and a PC. Our experiments with version 43 of QUIC shows that the behavior on both RPi and PC is quite similar which is shown in the state machine in Fig. 6.4. There is little difference between RPi and PC in the amount of time spent in various states. This shows that the performance of QUIC has improved over time and its userspace implementation is not a limiting factor on moderately powerful devices. It also validates our choice of using RPi as client.

To see if the userspace implementation of QUIC is a limiting factor on low end devices we stress the CPU of RPi using *Linux stress tool.* We repeat the web page download to record throughput and compute the difference between the stressed and unstressed setting. We perform this experiment on three RPis which have different link quality and capacity and the results are shown in Table 6.2. We can see that stressing the CPU has a big impact on QUIC throughput which drops by up to 54%, while TCP throughput is reduced by a maximum of 8.6% in the worst case. We also observe that the higher the bandwidth the bigger is the reduction. In low bandwidth links the network becomes the bottleneck therefore, stressing the CPU has little impact. These results show that QUIC's userspace implementation can have negative impact on low end devices with limited processing power. However, with moderately powerful devices this is not an issue.

6.1.5 Performance comparison of TCP and QUIC

Our objective here is to compare the performance of QUIC and TCP in WiFi networks. To this end, we compare the average throughput of over 200 experiments performed using each protocol as explained in Section 6.1.3.

WiFi frame aggregation is enabled by default in all antennas. Fig. 6.5 shows the mean throughput with 95% confidence interval achieved by TCP and QUIC on various clients. RP1 is the client located at UPC and is directly connected to LAN *without* any WiFi link. QUIC achieves slightly higher throughput of around 76 Mbps as compared to TCP's 72 Mbps on this wired link. This result conforms to prior studies that also find QUIC to perform slightly better than TCP. However, the outcome is very different with other clients that involve one or more WiFi hops on their end-to-end path and we can see that in almost all cases TCP achieves higher throughput than QUIC. The difference is higher is high-bandwidth links. e.g., in case of RP3 and RP2 TCP's throughput is larger than QUIC by roughly 50%. This

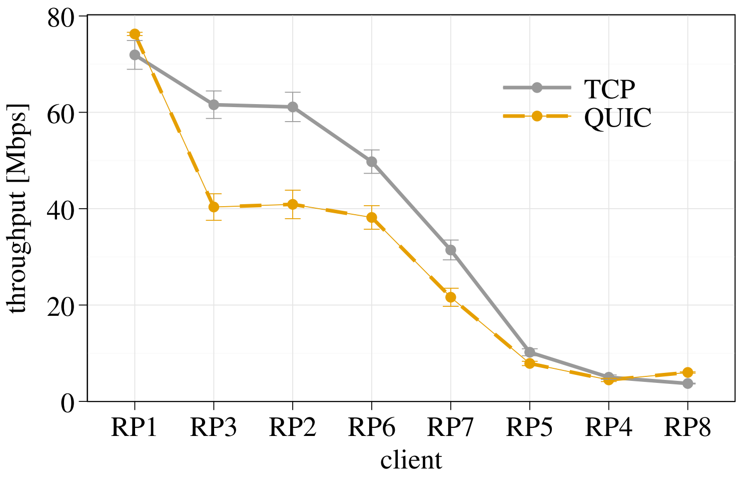


Figure 6.5: Mean throughput comparison (WiFi frame aggregation enabled)

	Throughput [Mbps]			Loss prob. %	
	TCP	QUIC	Gain %	TCP	QUIC
RP1	71.9	76.3	−5.7	0.0054	0.039
RP2	61.1	40.9	49.5	0.21	0.52
RP3	61.6	40.3	52.6	0.29	0.29
RP4	5.0	4.5	12.3	0.42	0.45
RP5	10.2	7.9	29.4	0.49	0.32
RP6	49.8	38.2	30.3	0.19	0.15
RP7	31.4	21.6	45.4	0.16	0.15
RP8	3.7	6.0	−38.1	0.64	0.64

Table 6.3: Comparison of throughput and loss probability of TCP and QUIC

difference is 30%, 45%, 29% and 12% for RP6, RP7, RP5 and RP4 respectively. RP8 is the only client where TCP performs worse than QUIC with WiFi links. Recall that RP8 reaches UPC using 5 WiFi links, one of them a long distance WiFi link of 15km. Table 6.3 shows that RP8 has the highest loss probability (0.64%) among all clients. TCP is known to be severely affected by packet losses when using a single connection as its cwnd size is reduced to a very small value [3, 54, 92, 93]. We consider RP8 as an outlier.

6.2 Determining factors of TCP and QUIC performance

To identify the determining factors of TCP and QUIC performance in WiFi, we apply data mining techniques on the collected network traffic. We extract several features from the collected packet traces at client and server side. These features are derived from information in IP and TCP/UDP headers and logs of cwnd.

6.2.1 List of all features

1. *Server/client mean inter-packet time*: The inter-departure time between packets on the server side and the inter-arrival time between packets on the client side
2. *Server/Client inter-packet time variation*: The variance of inter-packet time of server/client.
3. *Server burstiness/Client burstiness*: We have computed the burstiness using the index of dispersion for intervals as described in [94].
4. *Mean delay*: The mean value of one way delay of all data packets transferred from server to client.
5. *Delay variation*: The variance of one way delay.
6. *Packet loss*: Probability of lost packets. It is defined as the fraction of the total transmitted packets that did not arrive at the receiver. calculated by dividing total packets by the number of lost packets.
7. *Out of order*: Probability of out of order packets. It is defined as the fraction of the total transmitted packets that are received in a different order from which they were sent.
8. *ACK freq*: ACK frequency is the ratio of acknowledgements and data packets.
9. *Mean cwnd*: Mean value of the congestion window measured throughout the data transfer.
10. *Cwnd variation*: Variance in the size of congestion window.
11. *Slow start/ Congestion avoidance/ App limited/ Recovery freq*: The frequency of slow start, congestion avoidance, recovery and application limited phase respectively. This is the percentage of time spent in a particular phase during the data transfer.

6.2.2 Decision tree learning

We use decision tree learning which involves the construction of a decision tree from class-labeled data. The algorithm measures the entropy of various features and tests the expected reduction in entropy caused by partitioning all data points according to a given attribute. This is known as information gain which is essentially a measure of how much information a feature gives us about the class. The attribute with highest

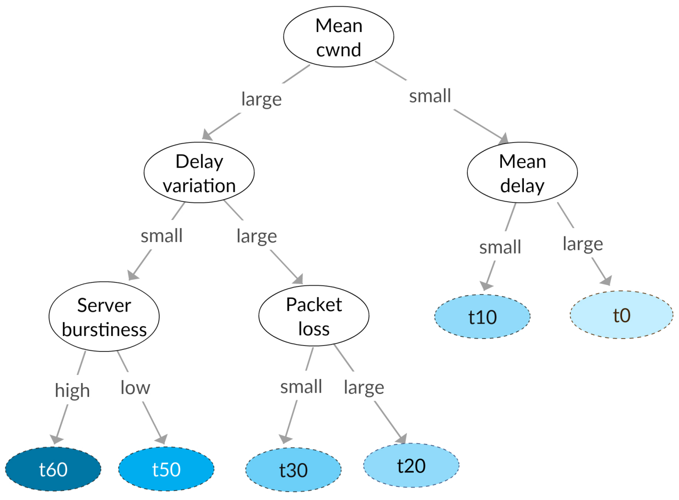


Figure 6.6: Decision tree for TCP

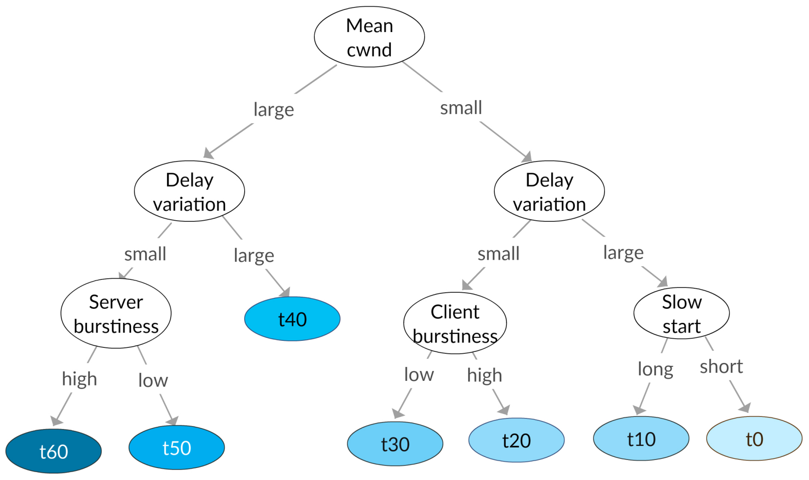


Figure 6.7: Decision tree for QUIC

information gain is split first and the process is repeated for each node which results in the tree. Interior nodes of the tree represent the features and leafs represent the target class. Our target variable is throughput which is the indicator of performance. We divide throughput into discrete bins of 10 Mbps that are classified as *t0*, *t10*, *t20*, *t30*, etc, where class *t0* represents throughput in the range of 0 Mbps to 10 Mbps, class *t10* represents throughput in the range of 10 Mbps to 20 Mbps and so on.

We create a row of features or data point from each individual data transfer performed between a client and server in our experiments. We derive roughly 2000 data points from our experiments and use them in the decision tree learning algorithm. We use the implementation of C4.5 decision tree algorithm offered by Weka [95]. The resulting pruned decision trees for TCP, QUIC and BQUIC are shown in Figure 6.6 and 6.7.

The features with the highest information gain are selected by the algorithm. The mean cwnd lies at the root of all trees indicating that it is the most important feature in determining throughput. The cwnd defines how much unacknowledged data can be in flight before an ACK is received from client. Generally, the larger the mean cwnd size, the higher the throughput. Algorithms such as slow start, congestion avoidance, etc. specify how to grow the cwnd size. The slow start phase is designed to quickly converge on the available bandwidth between the client and the server. During this phase the cwnd increases exponentially on each RTT. Once we are out of the slow start and into the congestion avoidance phase the cwnd grows at a much lower rate. Therefore the longer the slow start lasts, the higher is the overall throughput.

Delay is another factor that impacts the throughput. End-to-end delay directly impacts the growth of cwnd because the sender has to wait for ACK from the receiver before increasing the sending rate. The higher the delay, the lower the throughput. Wireless connections are also particularly susceptible to signal interference which causes network data to be corrupted in transit, increasing delays due to link layer retransmissions. This increase in delay is usually considered a sign of congestion by the congestion control algorithms which react by reducing the sending rate and consequently limiting the maximum throughput of the connection.

Packet losses impact the sending rate in a similar manner. TCP and QUIC are both reliable transport protocols, therefore they react to packet losses by reducing the sending rate and retransmitting the lost packets. However, packet loss has a much greater negative impact on TCP due to its head-of-line blocking issue. The loss of one of the packets in transit leads to all subsequent packets being held in the receiver's buffer until the lost packet arrives at the receiver. Since the application has no visibility into transport layer it has to wait for the full sequence of packets before accessing the data.

Server and client side traffic burstiness are also determined by the data mining algorithm as key players in defining the overall performance. We further investigate how the bursty traffic results in higher throughput and how it exploits WiFi frame aggregation.

6.2.3 Root causes of performance issues of QUIC in WiFi

QUIC is a general purpose protocol optimized for the broader Internet mostly consisting of wired links. However, due to the exponential increase in wireless traffic

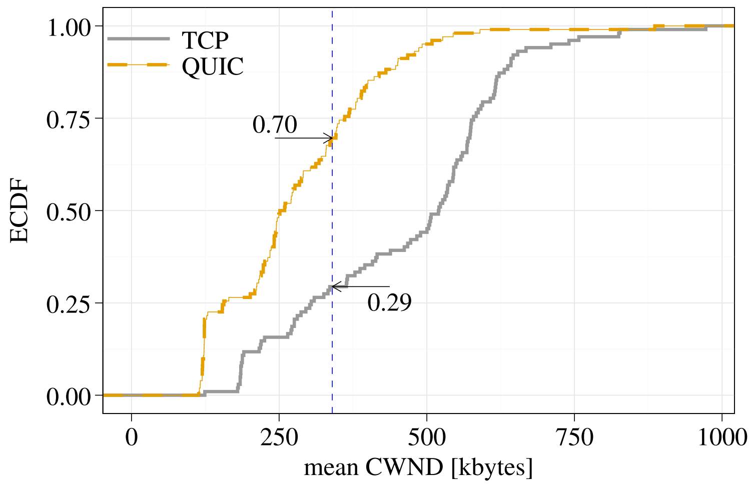


Figure 6.8: Mean cwnd comparison for RP3 client. The blue vertical line is the optimal cwnd size calculated using bandwidth and latency

in the recent years, it is becoming increasingly important to perform careful study of performance of QUIC in wireless networks and optimize it accordingly.

Our experiments show that QUIC has lower performance than TCP using WiFi links. The decision trees created using data mining algorithm also provide us useful information and show that cwnd size, high delay/RTT variation and client/server burstiness, etc. impact QUIC performance. We further investigate these factors and try to identify the root-cause for the problem, finding that some of the design decisions of QUIC impair its performance in WiFi networks. We have identified two potential causes of lower QUIC performance in WiFi.

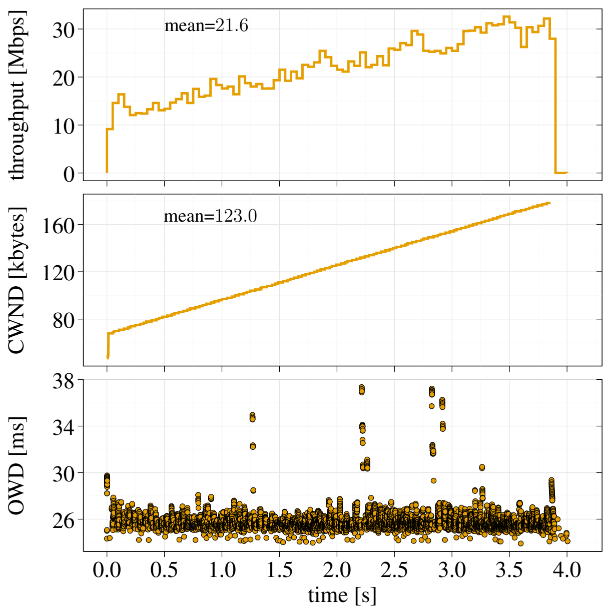
Cause 1: QUIC's congestion control and loss detection algorithms behave abnormally in high RTT variation

Since the cwnd size dictates throughput, we start our investigation by analyzing the cwnd of TCP and QUIC which is logged during all our experiments. We select RP3, the client with WiFi links having the highest throughput, and measure the bandwidth and RTT with the server in Polito using netperf and ping tools respectively. We estimate the optimal cwnd size (minimal window that allows the maximum throughput) using the well known bandwidth-delay product:

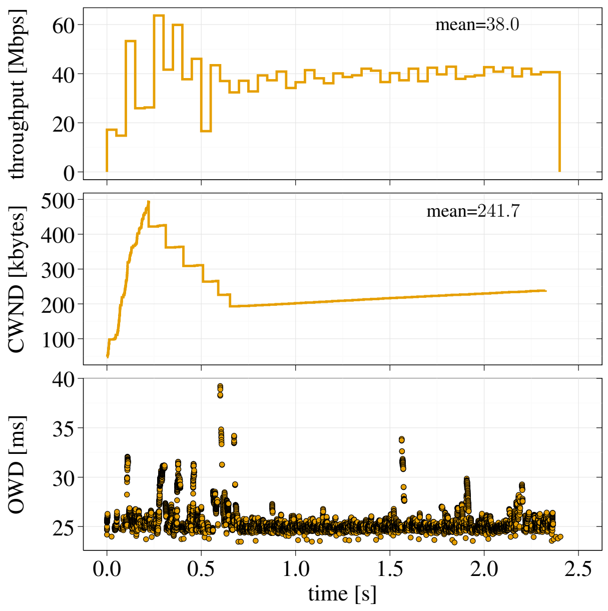
$$Optimal\ cwnd = Bandwidth \times RTT$$

With 62 Mbps of available bandwidth and 44 ms RTT, the cwnd size needs to be at least 340 kB to saturate the link. Next, in Figure 6.8 we plot the ECDF of the mean cwnd size measured during more than 200 experiments each for TCP and QUIC. The blue vertical line represents the optimal cwnd size computed using bandwidth-delay product. We can see that in almost 70% of cases QUIC chooses a cwnd size which is smaller that the optimal value, while this occurs in only 29% of cases with TCP. Furthermore, in around 25% cases the selected cwnd size by QUIC is even smaller than half of the optimal size. This clearly shows why QUIC has poor performance. Our next target is to find out why QUIC selects a sub-optimal cwnd so often. We plot individual cwnd charts for over 200 experiments performed using QUIC and TCP. We observe that two abnormal behaviors in QUIC. First, QUIC often exists early from slow start, resulting in small mean cwnd size. Second, QUIC spuriously detects congestion and reacts by reducing cwnd size.

QUIC exits early from slow start: Slow start which is also known as *exponential growth* algorithm is used to quickly converge on the available bandwidth of the end-to-end path between client and the server. Premature exit from slow start results in underestimation of bandwidth and low throughput. QUIC's congestion control algorithm exits from slow start phase when an increasing delay is detected. It compares the minimum RTT of the current burst of packets relative to the minimum RTT during the session. If the delay is larger than a certain threshold, the slow start phase exits. QUIC uses a *slow start delay factor* of 1/8 which implies that the slow start will exit if the RTT has increased by more than 1/8$^{\text{th}}$. E.g., if minimum RTT is 40ms, an increase of up to 5ms from this value can be tolerated for the slow start phase to continue. This idea works well in wired networks where RTTs are quite stable and a significant increase indicates congestion. However, the same does not hold true for WiFi networks where it is common to have variation in RTT due to factors related to the underlying medium and not due to network congestion. High delay variation in the network at the start of a transfer forces QUIC to exit early from slow start phase. As a result, the cwnd cannot increase to the optimal size and leads to low throughput. This behavior of QUIC is shown in Fig. 6.9a which compares the delay, cwnd size and throughput of QUIC for a single transfer using RP3 where this behavior occur. We can see that right at the start of the transfer the delay varies between 24 ms and 30 ms. This variation of around 6 ms is larger than QUIC's threshold of 3 ms which results in early exit from slow start. Thus in this download QUIC achieves a mean cwnd size of 123 kB, which is much lower than the optimal size of around 320 kB. The corresponding throughput of QUIC in this download is around 21.6 Mbps. In contrast the mean throughput of TCP over all



(a) Cwnd graph shows QUIC's early exit from slow due to high variation in delay in the beginning



(b) Cwnd graph shows QUIC spuriously detects congestion due to high variation in delay and reduces cwnd size

Figure 6.9: High delay variation affects QUIC's slow start and loss detection algorithms

downloads is around 62 Mbps for the same link.

QUIC spuriously detects congestion: Spuriously detecting congestion and declaring packets as lost causes unnecessary retransmissions and may result in performance degradation because the protocol takes action to reduce the sending rate. QUIC's loss detection algorithm tracks the packets that are in-flight and unacknowledged. When it receives ACK for a packet that was sent later than an unacknowledged packet it starts a timer based on the sending time of packet and declares a packet as lost if its acknowledgement is not received within a particular time period. QUIC uses a *time reordering factor* of 1/8 (fraction of RTT), which is the maximum reordering in time space. It means that the timer is set at 1 RTT plus a threshold of $1/8^{\text{th}}$ of the RTT from sending time. As explained earlier, this threshold is quite conservative and causes performance degradation in WiFi networks with high RTT variation. Fig. 6.9b shows this particular behavior. In this particular download QUIC stays in slow start for longer duration and is able to grow its cwnd to a good size, but soon afterwards it starts reducing the cwnd repeatedly and we see a staircase pattern going downwards. This is again caused by high delay variations that are visible in OWD chart. This reduction in cwnd size is triggered by QUIC's loss detection algorithm. However using the IP identification field from the client and server side traffic captures we confirm that there is *zero* packet loss during this transfer.

Cause 2: QUIC does not exploit advanced WiFi transmission methods

QUIC aims to reduce traffic burstiness and, as a consequence, queuing delays and packet losses by using *packet pacing* [96]. While this idea is generally true in wired networks, we observe that due to the characteristics of the WiFi medium and interactions between transport and MAC protocols, bursty traffic might actually be beneficial in WiFi. One apparent reason for this behavior is frame aggregation. As mentioned in Section 2.3.2, frame aggregation in recent 802.11 standards is a key feature to achieve high throughput, and bursty nature of traffic in TCP increases frame aggregation opportunities.

In order to measure the impact of frame aggregation on TCP and QUIC performance we repeat our experiments by disabling frame aggregation in the antenna located at UPC. The traffic from the client RP6 in the WMN passes through this WiFi link to reach the gateway and then the server in Polito. In order to assess the relevance of frame aggregation and to observe how both protocols exploit it, table 6.4 shows the throughput and 95% confidence intervals obtained for RP6 with and without

	Throughput (Mbps)		
	Without frame aggregation	With frame aggregation	Gain(%)
TCP	26.1 ± 1.3	49.8 ± 2.4	90
QUIC	23.7 ± 1.1	38.2 ± 2.4	61

Table 6.4: RP6 client throughput with and without frame aggregation

frame aggregation in the WiFi link. The table shows that with frame aggregation the throughput is increased by almost 90% with TCP and only 61% with QUIC. This shows that TCP can efficiently exploit frame aggregation while QUIC is not able to fully benefit from it.

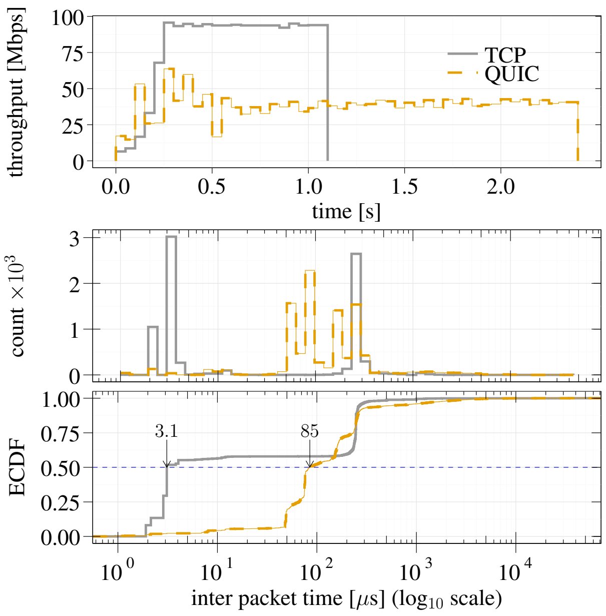
To see why this is the case, we compare the inter-packet time of both protocols. As an example, Fig. 6.10a shows the ECDF and histogram of the inter-packet time of TCP and QUIC for one of the downloads from RP3. This information is retrieved from the server side capture files of two back-to-back experiments done using TCP and QUIC. We can see that around 50% of TCP packets have inter-packet time between 2μs and 4μs while for QUIC this range is between 50μs and 100μs. Large inter-packet time means less chances of exploiting frame aggregation.

Figure 6.10b shows a scatter plot of the throughput vs. burstiness for all downloads from RP3. For TCP the figure shows a clear correlation between throughput and burstiness: the higher is the burstiness the higher is the throughput. However, burstiness is not the only parameter influencing the throughput. This explains why there is a small proportion of points having a relatively low throughput even if the burstiness is high. As expected, Figure 6.10b shows that QUIC is much less bursty than TCP, due to packet pacing.

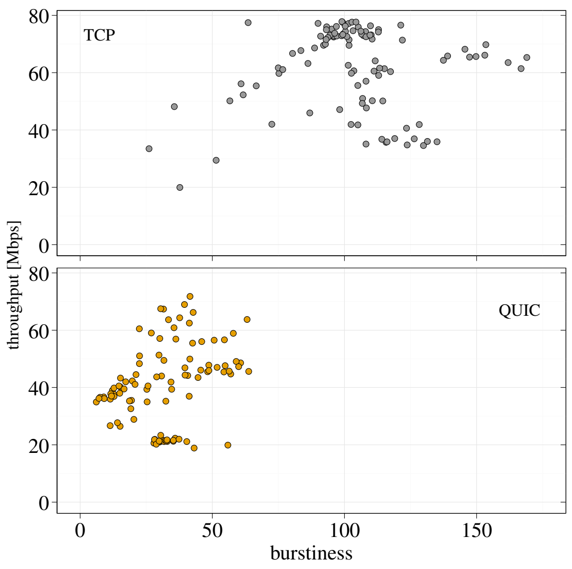
Apart from packet pacing, the main aspect of QUIC implementation which influences how the protocol interacts with 802.11 frame aggregation is the acknowledgment mode. QUIC has two acknowledgment modes:

- **TCP_ACKING**: This mode is similar to TCP delayed acknowledgment, in which an ACK is generated for every 2 received packets in accordance with RFC 1122. This is the default mode of QUIC in Chromium at the time of writing.
- **ACK_DECIMATION**: In this mode acknowledgments are delayed up to a maximum of 10 packets and a cumulative ACK is generated. The maximum duration for which the ACK can be delayed is 25 ms.

We have experimentally observed that TCP dynamically changes the acknowledgment rate according to the network conditions while QUIC's acknowledgment rate is always



(a) Inter-packet time comparison of TCP and QUIC



(b) Impact of burstiness on throughput of TCP and QUIC

Figure 6.10: TCP gets higher benefits from WiFi frame aggregation due to its bursty traffic

static. When a client performs cumulative acknowledgment for several packets, the server sends out a burst of packets that take advantage of frame aggregation in WiFi MAC layer and results in higher throughput.

6.3 Optimizing QUIC

6.3.1 Modifications

Taking into consideration the performance issues of QUIC in WiFi, we evaluate a customized version of QUIC that incorporates the following modifications to the original protocol. (1) We use `ACK_DECIMATION` as default acknowledgment mode which reduces the transport layer acknowledgment frequency, increasing traffic burstiness at the server side and thus resulting in greater aggregation opportunities at the WiFi MAC layer. (2) We make QUIC's congestion controller more tolerant to RTT variation in WiFi networks by increasing the threshold defined for exiting slow start. (3) We make QUIC's time based loss detection algorithm more resilient in WiFi networks by increasing the threshold defined for considering a packet lost.

Since these features is not controllable from the browser configuration, we had to study the source code, make required modifications, and compile a customized version of Chromium browser. We call this version *Bursty QUIC* (BQUIC).

We have carefully proposed these changes in the QUIC protocol design so that tweaking it to make it perform better on one particular link layer does not have negative effects in the general case. We validate the effectiveness of these changes by performing experiments in a production network where the end-to-end path between client and server contains WiFi last hops as well as wired links in the Internet.

6.3.2 Performance evaluation of QUIC and BQUIC

To better understand the difference in behavior of QUIC and BQUIC and their performance, we compare various parameter including throughput, loss rate and delay. Tab. 6.5 presents the mean values calculated for each client. RP1 is the client directly connected to LAN in UPC and has no WiFi hop in its path. We can see that BQUIC achieves much higher throughput particularly in high bandwidth WiFi links. RP8 is an exception where we see around 20.8% reduction in throughput. This

Table 6.5: Performance comparison of QUIC and BQUIC

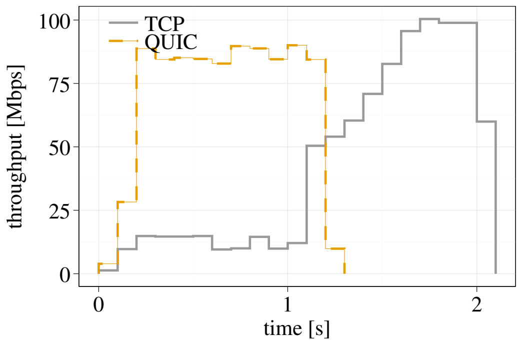
Device	Throughput (Mbps)			Loss rate (%)		Delay (ms)	
	QUIC	BQUIC	Gain	QUIC	BQUIC	QUIC	BQUIC
RP1	76.3	80.3	5.4%	0.03	0.01	22.2	23.8
RP2	40.9	50.6	23.8%	0.51	0.32	22	22.8
RP3	40.3	51.7	28%	0.29	0.25	25.6	26.3
RP4	4.5	4.8	7.3%	0.44	0.39	93.1	86.6
RP5	7.9	9.2	16.3%	0.32	0.29	56.7	58.5
RP6	38.2	42.2	10.7%	0.15	0.33	12.6	13.2
RP7	21.6	26	20.5%	0.15	0.17	41.5	47
RP8	6	4.75	-20.8%	0.64	0.99	57	51

particular client is located in Castelldefels 15km away from Barcelona and has a long distance WiFi link.

The losses have been computed by comparing the identification field of the IP header of transmitted and received datagrams. We have computed the 95% confidence intervals for throughput, and they are small in all cases (less than 10%). The throughput gain of BQUIC over QUIC is in the range of 7% to 28% for different clients. Higher speed links show higher gain, since it take longer for QUIC to reach the peak throughput after spurious reduction in cwnd, while BQUIC avoids this phenomenon. Regarding the losses measured at the transport layer, Tab. 6.5 shows that they are very low. This is normal on a WiFi link of an acceptable quality, since 802.11 retransmits lost unicast frames multiple times before abandoning its transmission. Indeed, the worst connected device (RP8) has a loss of only 0.64% in QUIC and 0.99% in BQUIC. It is interesting to see that in most case the packet loss rate is comparable with BQUIC. It shows that the bursty nature of BQUIC traffic does not increase the packet loss rate in most cases. We also compare the delay duration of the packets to see if high transfer rate of BQUIC causes queues to build up which would result in increase in delay. In most case the difference in delay is negligible. In RP4 we observe that delay decreases from 93.1ms to 86.6ms while in RP7 it increases from 41.5ms to 47ms.

6.3.3 Fairness comparison

Fairness among the competing flows of various transport protocols very important for a network. It has been shown in prior studies that QUIC consumes a greater share of bottleneck bandwidth as compared to TCP. In our previous work [3] we observed



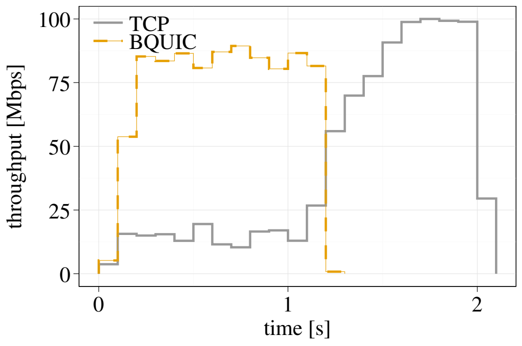
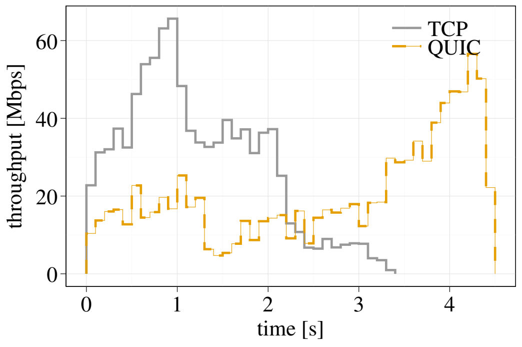


Figure 6.11: Fairness comparison in wired network



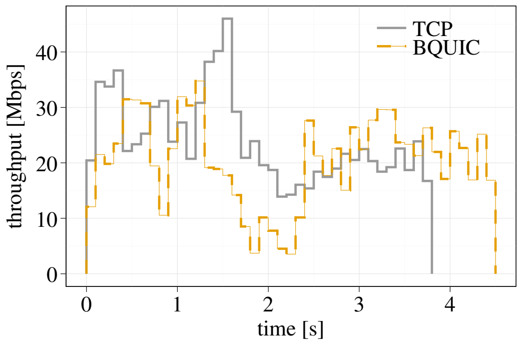


Figure 6.12: Fairness comparison in WiFi network

in an emulated environment that a QUIC v39 flow consumes around than 70% of the bottleneck bandwidth when competing with a single TCP flow and around 50% when competing with 2 TCP flows. [55] have shown even greater unfairness with QUIC v34 where QUIC consumes more than half of the bottleneck bandwidth even when competing with 4 TCP flows.

Here our main objective is to measure the fairness of QUIC v43 and BQUIC with TCP in a WiFi network. We have shown that BQUIC is able to achive higher throughput than QUIC in WiFi but it is crucial to make sure that it does not starve other competing flows. We perform measurements using two different links. First is the wired link between the two university campuses in Spain and Italy. Second is the WiFi link between a client in the WMN and UPC campus. We run a script that starts two parallel threads on the client to fetch a web page from the server using TCP and QUIC. We repeat the experiment using TCP and BQUIC.

Fig. 6.11 shows the results of the wired network. In agreement with prior studies, we can see that QUIC consumes much higher share of the bottleneck bandwidth than TCP. BQUIC also has similar behavior and we do not find it to be more aggressive than QUIC, confirming that our proposed changes do not have a negative impact on fairness in fixed networks.

In case of WiFi network the behavior of QUIC and TCP is very different as shown in Fig. 6.12. Here TCP gets much higher share of bandwidth and QUIC is able to grab more bandwidth only after the TCP flow is about to finish. Interestingly, the competing flows of TCP and BQUIC are quite fair to each other which is a positive thing. Besides providing higher throughput, our proposed optimization also provides better fairness with competing TCP flows.

6.4 Summary

In this chapter we analyzed the performance of QUIC in WiFi, investigating the interactions of the protocol with 802.11 frame aggregation. We first highlighted that QUIC delivers sub-optimal throughput in typical WiFi scenarios. The root-cause is the way QUIC paces packets in the network and its acknowledgment mechanisms as well as high delay variations in WiFi links which is misinterpreted by QUIC as a sign of congestion.

We implemented and evaluated *BQUIC*, i.e., a customized version of QUIC that increases traffic burstiness and optimizes threshold for slow start and loss detection algorithms. We carried out experiments in a production WMN. Results show that BQUIC increases the throughput between 7% to 28% for different clients.

Chapter 7

Conclusion

In this thesis we presented a comprehensive study of H2 and QUIC, which are among the most popular network protocols of current times. We start by providing analysis of the characteristics of H1 and H2 traffic using large and diverse spatiotemporal datasets. Our findings reveal that H1 and H2 traffic has very different characteristics. This calls for a reappraisal of web traffic models, as well as HTTP traffic simulation and benchmarking approaches. Such models were built based on the understanding of H1 traffic and are no longer valid for modern web traffic. Our next objective was to enable network administrators to identify H1 and H2 flows for network management tasks such as traffic shaping or prioritization. Since flow monitoring is the most widely used technique for getting visibility into network bandwidth utilization and applying QoS policies, we developed a machine learning-based system for HTTP traffic classification that only used flow attributes such as byte and packet counters which are commonly available in technologies like NetFlow or IPFIX. The system has very high accuracy and as well as spatial and temporal stability. This system can also be used by researchers to easily track the adoption of H2 protocol in any network and it may be applied to future versions of HTTP protocol as well.

We also provided a detailed study on the real-world usage of H2 multiplexing and revealed a divergent behavior of the protocol where multiple parallel connections are established to the same server. This behavior does not conform to the rules of the RFC. Our finding is critical for the future of the Internet because H2 comes with a promise of reducing the number of connections between clients and servers which would have a positive impact on servers and network infrastructure.

Second part of this thesis focused on providing performance evaluation of H2 and QUIC. We evaluated scenarios with network congestion or high rate of random packet losses and showed that H2 exhibits poor performance in such conditions. We designed and implemented an application layer optimizing technique for H2 in the source code Chromium browser which significantly improved the performance. Finally, we highlighted the determining factors of QUIC's performance in WiFi networks. We investigated the interaction of QUIC transport with the advanced features of WiFi such as frame aggregation. Furthermore, the impact of high variations in delay on the congestion control and loss detection algorithms of QUIC were evaluated. Based on our findings we implemented several optimizations for QUIC which significantly improved its throughput in WiFi networks.

With this research we advance the state-of-the-art in empirical evaluation of H2 and QUIC protocols. This work will enable the readers to get insights into the behavior and performance of modern network protocols and develop a better understanding of how various design decisions impact the performance in various network environments.

7.0.1 Future Work

Performance of QUIC in cellular networks

Our research was focused on performance evaluation of QUIC in WiFi networks and we discovered several factors such as high variation in RTT that resulted in performance degradation of QUIC. Cellular networks have some similar characteristics and will interesting to evaluate TCP and QUIC performance in such networks. The benefits of QUIC's connection migration e.g., from WiFi to LTE can also be studied. Furthermore, mobility is a part and parcel of cellular networks and whether it affects QUIC and TCP differently is an open question for future research.

HTTP3 over QUIC

HTTP/3 is expected to become the third official version of the HTTP protocol and it will run on top of QUIC instead of TCP. Due to the popularity of video streaming some researchers have started investigating how to run DASH-style streaming applications using HTTP/3 over QUIC. While one study [97] finds that these applications see slight performance degradation over QUIC, another [49] shows performance improvement. Further research in this direction is required.

QoE and monitization hurdles for mobile operators with QUIC

Guaranteeing subscriber QoE is crucial for mobile operators. Similarly, creating monitization policies based on type of content is crucial to their business. These days video traffic accounts for the majority of all mobile traffic. Top video sites like YouTube, Netflix and Facebook use Adaptive Bit Rate (ABR) video technology which is designed to select the highest sustainable bit rate. ABR over QUIC creates hurdles for mobile operators as the transport layer headers are also encrypted along with the payload. New techniques are required to detect such traffic and to avoid network congestion and guarantee fairness.

Unreliable transfer over QUIC

Researchers are currently looking into different ways in which real time media can be supported over QUIC. Possible use cases include video conference, augmented reality and virtual reality, etc. There are proposals for extensions to QUIC that allow sending and receiving unreliable datagrams over a QUIC connection. This is a new research direction with a lot of open questions to address.

QUIC for 5G packet core

5G technology is on the horizon and it will play a pivotal role in meeting the exponentially increasing demand of mobile data and new business models based on low latency. In order to guarantee low latency and high bandwidth there is a need for an efficient transport protocol and QUIC is a potential candidate to replace TCP. Evaluation of QUIC with 5G core is an open research direction.

References

1. Manzoor, J., Drago, I. & Sadre, R. *The curious case of parallel connections in http/2* in *2016 12th International Conference on Network and Service Management (CNSM)* (2016), 174–180 (cit. on pp. 5, 27).
2. Manzoor, J., Drago, I. & Sadre, R. *How http/2 is changing web traffic and how to detect it* in *2017 Network Traffic Measurement and Analysis Conference (TMA)* (2017), 1–9 (cit. on p. 5).
3. Manzoor, J., Sadre, R., Drago, I. & Cerda-Alabern, L. *Is There a Case for Parallel Connections with Modern Web Protocols?* in *2018 IFIP Networking* (2018) (cit. on pp. 5, 78, 89).
4. Manzoor, J., Cerda-Alabern, L., Sadre, R. & Drago, I. *Improving Performance of QUIC in WiFi* in *2019 IEEE Wireless Communications and Networking Conference (WCNC)* (2019) (cit. on p. 5).
5. Manzoor, J., Cerda-Alabern, L., Sadre, R. & Drago, I. *On the Causes of Performance Issues of QUIC over Wireless Links* in *To be submitted to Ad Hoc Networks Journal* (2019) (cit. on p. 6).
6. Berners-Lee, T., Fielding, R. & Frystyk, H. *Requests for comments 1945 - Hypertext Transfer Protocol – HTTP/1.0* `https://www.ietf.org/rfc/rfc1945` (cit. on p. 9).
7. Thomas, B., Jurdak, R. & Atkinson, I. SPDYing Up the Web. *Commun. ACM* **55,** 64–73 (Dec. 2012) (cit. on pp. 10, 17).
8. *Web browsers profiling* `http://www.browserscope.org` (cit. on p. 10).
9. Belsche, M., Peon, R. & Thomson, M. *Request for Comments 7540 - Hypertext Transfer Protocol Version 2 (HTTP/2)* `https://tools.ietf.org/html/rfc7540` (cit. on p. 11).

10. Peon, R. & Ruellan, H. *Request for Comments 7541 - HPACK: Header Compression for HTTP/2* `https://tools.ietf.org/rfc/rfc7541.txt` (cit. on p. 11).

11. *HTTP/2 adoption* `http://isthewebhttp2yet.com/measurements/adoption.html` (cit. on p. 12).

12. *Browser support tables* `http://caniuse.com/` (cit. on pp. 12, 26).

13. *QUIC Implementations* `https://github.com/quicwg/base-drafts/wiki/Implementations`. QUIC Implementations (cit. on p. 13).

14. Altman, E. & Jiménez, T. *Novel delayed ACK techniques for improving TCP performance in multihop wireless networks* in *IFIP International Conference on Personal Wireless Communications* (2003), 237–250 (cit. on p. 15).

15. Singh, A. K. & Kankipati, K. *TCP-ADA: TCP with adaptive delayed acknowledgement for mobile ad hoc networks* in *Wireless Communications and Networking Conference, 2004. WCNC. 2004 IEEE* **3** (2004), 1685–1690 (cit. on p. 15).

16. De Oliveira, R. & Braun, T. *A dynamic adaptive acknowledgment strategy for TCP over multihop wireless networks* in *INFOCOM 2005. 24th annual joint conference of the IEEE Computer and Communications Societies. Proceedings IEEE* **3** (2005), 1863–1874 (cit. on p. 15).

17. De Oliveira, R. & Braun, T. A smart TCP acknowledgment approach for multihop wireless networks. *IEEE Transactions on Mobile Computing* **6,** 192–205 (2007) (cit. on p. 15).

18. Xylomenos, G., Polyzos, G. C., Mahonen, P. & Saaranen, M. TCP performance issues over wireless links. *IEEE communications magazine* **39,** 52–58 (2001) (cit. on p. 15).

19. Scharf, M., Necker, M. & Gloss, B. *The sensitivity of TCP to sudden delay variations in mobile networks* in *International Conference on Research in Networking* (2004), 76–87 (cit. on p. 15).

20. Gurtov, A. in *Emerging Personal Wireless Communications* 87–105 (Springer, 2002) (cit. on p. 15).

21. Bhartia, A., Chen, B., Wang, F., Pallas, D., Musaloiu-E, R., Lai, T. T.-T. & Ma, H. *Measurement-based, practical techniques to improve 802.11 ac performance* in *Proceedings of the 2017 Internet Measurement Conference* (2017), 205–219 (cit. on p. 16).

22. Skordoulis, D., Ni, Q., h. Chen, H., Stephens, A. P., Liu, C. & Jamalipour, A. IEEE 802.11n MAC frame aggregation mechanisms for next-generation high-throughput WLANs. *IEEE Wireless Communications* **15,** 40–47 (Feb. 2008) (cit. on p. 16).

23. Kim, B. S., Hwang, H. Y. & Sung, D. K. *Effect of Frame Aggregation on the Throughput Performance of IEEE 802.11n* in *2008 IEEE Wireless Communications and Networking Conference* (Mar. 2008), 1740–1744 (cit. on p. 16).

24. Lin, Y. & Wong, V. W. S. *WSN01-1: Frame Aggregation and Optimal Frame Size Adaptation for IEEE 802.11n WLANs* in *IEEE Globecom 2006* (Nov. 2006), 1–6 (cit. on p. 16).

25. Krishnamurthy, B. & Wills, C. E. Analyzing factors that influence end-to-end Web performance. *Computer Networks* **33,** 17–32 (2000) (cit. on p. 16).

26. Mah, B. A. *An empirical model of HTTP network traffic* in *INFOCOM'97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE* **2** (1997), 592–600 (cit. on p. 16).

27. Sadre, R. & Haverkort, B. R. *Changes in the Web from 2000 to 2007* in *International Workshop on Distributed Systems: Operations and Management* (2008), 136–148 (cit. on p. 16).

28. Ihm, S. & Pai, V. S. *Towards understanding modern web traffic* in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference* (2011), 295–312 (cit. on p. 16).

29. Varvello, M., Schomp, K., Naylor, D., Blackburn, J., Finamore, A. & Papagiannaki, K. in (eds Karagiannis, T. & Dimitropoulos, X.) 218–232 (2016) (cit. on p. 16).

30. Zimmermann, T., Rüth, J., Wolters, B. & Hohlfeld, O. *How HTTP/2 Pushes the Web: An Empirical Study of HTTP/2 Server Push* in *Proceedings of the IFIP Networking Conference* (2017) (cit. on pp. 17, 22).

31. Elkhatib, Y., Tyson, G. & Welzl, M. *Can SPDY really make the web faster?* in *Networking Conference, 2014 IFIP* (June 2014), 1–9 (cit. on pp. 17, 22, 64).

32. Wang, X. S., Balasubramanian, A., Krishnamurthy, A. & Wetherall, D. *How Speedy is SPDY?* in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (USENIX Association, Seattle, WA, 2014), 387–399 (cit. on pp. 17, 18, 22, 39, 60, 64).

33. De Saxcé, H., Oprescu, I. & Chen, Y. *Is HTTP/2 really faster than HTTP/1.1?* in *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)* (Apr. 2015), 293–299 (cit. on pp. 17, 22, 64).

34. Erman, J., Gopalakrishnan, V., Jana, R. & Ramakrishnan, K. K. Towards a SPDY'ier mobile web? *IEEE/ACM Transactions on Networking* **23,** 2010–2023 (2015) (cit. on pp. 17, 22, 64).

35. Zarifis, K., Holland, M., Jain, M., Katz-Bassett, E. & Govindan, R. *Modeling HTTP/2 Speed from HTTP/1 Traces* in *Proceedings of the PAM Conference* (2016), 233–247 (cit. on p. 18).

36. Ford, A., Raiciu, C., Handley, M. & Bonaventure, O. *TCP Extensions for Multipath Operation with Multiple Addresses* `https://tools.ietf.org/html/rfc6824` (cit. on p. 18).

37. Paasch, C., Detal, G., Duchene, F., Raiciu, C. & Bonaventure, O. *Exploring Mobile/WiFi Handover with Multipath TCP* in *Proceedings of the ACM SIGCOMM Workshop on Cellular Networks: Operations, Challenges, and Future Design* (2012), 31–36 (cit. on p. 18).

38. Chen, Y.-C., Lim, Y.-s., Gibbens, R. J., Nahum, E. M., Khalili, R. & Towsley, D. *A Measurement-based Study of MultiPath TCP Performance over Wireless Networks* in *Proceedings of the IMC* (2013), 455–468 (cit. on pp. 18, 22).

39. Ferlin, S., Dreibholz, T. & Alay, Ö. *Multi-path transport over heterogeneous wireless networks: Does it really pay off?* in *2014 IEEE Global Communications Conference* (2014), 4807–4813 (cit. on p. 18).

40. Deng, S., Netravali, R., Sivaraman, A. & Balakrishnan, H. *WiFi, LTE, or Both?: Measuring Multi-Homed Wireless Internet Performance* in *Proceedings of the IMC* (2014), 181–194 (cit. on p. 18).

41. Han, B., Qian, F., Hao, S. & Ji, L. *An Anatomy of Mobile Web Performance over Multipath TCP* in *Proceedings of the ACM CoNEXT* (2015), 5:1–5:7 (cit. on pp. 19, 22, 60, 64).

42. Han, B., Qian, F. & Ji, L. *When Should We Surf the Mobile Web Using Both Wifi and Cellular?* in *Proceedings of the 5th Workshop on All Things Cellular: Operations, Applications and Challenges (ATC)* (2016), 7–12 (cit. on pp. 19, 60).

43. Rüth, J., Poese, I., Dietzel, C. & Hohlfeld, O. *A First Look at QUIC in the Wild* in *International Conference on Passive and Active Network Measurement* (2018), 255–268 (cit. on p. 19).

44. Jager, T., Schwenk, J. & Somorovsky, J. *On the security of TLS 1.3 and QUIC against weaknesses in PKCS# 1 v1. 5 encryption* in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (2015), 1185–1196 (cit. on p. 19).

45. Fischlin, M. & Günther, F. *Multi-stage key exchange and the case of Google's QUIC protocol* in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (2014), 1193–1204 (cit. on p. 19).

46. Lychev, R., Jero, S., Boldyreva, A. & Nita-Rotaru, C. *How secure and quick is QUIC? Provable security and performance analyses* in *2015 IEEE Symposium on Security and Privacy* (2015), 214–231 (cit. on p. 19).

47. De Coninck, Q. & Bonaventure, O. *Multipath QUIC: Design and Evaluation* in *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies* (2017), 160–166 (cit. on p. 20).

48. Rabitsch, A., Hurtig, P. & Brunström, A. *A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths* in *ACM CoNEXT 2018 Workshop on the Evolution, Performance, and Interoperability of QUIC (EPIQ'18)* (2018) (cit. on p. 20).

49. Bhat, D., Rizk, A. & Zink, M. *Not so QUIC: A performance study of DASH over QUIC* in *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video* (2017), 13–18 (cit. on pp. 20, 94).

50. Perkins, C. & Ott, J. *Real-time Audio-Visual Media Transport over QUIC* in *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (2018), 36–42 (cit. on p. 20).

51. McQuistin, S. & Perkins, C. S. *Is Explicit Congestion Notification usable with UDP?* in *Proceedings of the 2015 Internet Measurement Conference* (2015), 63–69 (cit. on p. 20).

52. Megyesi, P., Krämer, Z. & Molnár, S. *How quick is QUIC?* in *Proceedings of the ICC* (2016), 1–6 (cit. on pp. 20, 22, 60).

53. Carlucci, G., De Cicco, L. & Mascolo, S. *HTTP over UDP: an Experimental Investigation of QUIC* in *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (2015), 609–614 (cit. on pp. 20, 22, 60).

54. Cook, S., Mathieu, B., Truong, P. & Hamchaoui, I. *QUIC: Better for what and for whom?* in *IEEE International Conference on Communications (ICC2017)* (2017) (cit. on pp. 21, 22, 78).

55. Kakhki, A. M., Jero, S., Choffnes, D., Nita-Rotaru, C. & Mislove, A. *Taking a long look at QUIC: an approach for rigorous evaluation of rapidly evolving transport protocols* in *Proceedings of the 2017 Internet Measurement Conference* (2017), 290–303 (cit. on pp. 21, 22, 56, 69, 76, 90).

56. Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R., Sperotto, A. & Pras, A. Flow Monitoring Explained: From Packet Capture to Data Analysis with NetFlow and IPFIX. *Commun. Surveys Tuts.* **16,** 2037–2064 (2014) (cit. on p. 23).

57. Finamore, A., Mellia, M., Meo, M., Munafò, M. M. & Rossi, D. Experiences of Internet Traffic Monitoring with Tstat. *IEEE Network* **25,** 8–14 (2011) (cit. on pp. 24, 41, 42, 45).

58. Langley, A. *Transport Layer Security (TLS) Next Protocol Negotiation Extension* Internet-Draft draft-agl-tls-nextprotoneg-03 (IETF Secretariat, Apr. 2012) (cit. on p. 25).

59. Friedl, S., Popov, A., Langley, A. & Stephan, E. *Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension* RFC 7301 (RFC Editor, July 2014) (cit. on p. 25).

60. *Zero protocol by Facebook* `https://code.facebook.com/posts/608854979307125` (cit. on p. 26).

61. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. & Witten, I. H. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.* **11,** 10–18 (Nov. 2009) (cit. on pp. 30, 32).

62. Mitchell, T. *Machine Learning* 1st ed. (McGraw-Hill, New York, 1997) (cit. on p. 31).

63. Carela-Español, V., Barlet-Ros, P., Mula-Valls, O. & Solé-Pareta, J. An autonomic traffic classification system for network operation and management. *Journal of Network and Systems Management* **23,** 401–419 (2015) (cit. on p. 35).

64. *Chrome DevTools Overview* `https://developer.chrome.com/devtools` (cit. on p. 40).

65. *HttpWatch* `http://www.httpwatch.com/` (cit. on p. 40).

66. *Remote Debugging on Android with Chrome* `https://developer.chrome.com/devtools/docs/remote-debugging` (cit. on p. 40).

67. Bermudez, I., Mellia, M., Munafò, M. M., Keralapura, R. & Nucci, A. *DNS to the Rescue: Discerning Content and Services in a Tangled Web* in *Proc. of the ACM Internet Measurement Conference* (2012), 413–426 (cit. on p. 41).

68. Maier, G., Schneider, F. & Feldmann, A. *NAT Usage in Residential Broadband Networks* in *Proceedings of the 12th International Conference on Passive and Active Measurement* (Springer-Verlag, Atlanta, GA, 2011), 32–41 (cit. on p. 45).

69. *Chrome multiple TCP connection bug* `https://bugs.chromium.org/p/chromium/issues/detail?id=718576` (cit. on p. 52).

70. Cisco. *BGP Best Path Selection Algorithm* `https://www.cisco.com/c/en/us/support/docs/ip/border-gateway-protocol-bgp/13753-25.html` (cit. on p. 54).

71. Juniper. *Understanding BGP Multipath* `https://www.juniper.net/documentation/en_US/junos/topics/concept/bgp-multipath-understanding.html` (cit. on p. 54).

72. Augustin, B., Friedman, T. & Teixeira, R. Measuring Multipath Routing in the Internet. *IEEE/ACM Transactions on Networking* **19,** 830–840 (2011) (cit. on p. 55).

73. Bellardo, J. & Savage, S. *Measuring packet reordering* in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment* (2002), 97–105 (cit. on p. 55).

74. Cataltepe, Z. & Moghe, P. *Characterizing Nature and Location of Congestion on the Public Internet* in *Proceedings of the ISCC Symposium* (2003), 741–746 (cit. on p. 56).

75. Akella, A., Seshan, S. & Shaikh, A. *An Empirical Evaluation of Wide-Area Internet Bottlenecks* in *Proceedings of the IMC* (2003), 101–114 (cit. on p. 56).

76. Tachibana, A., Shigehiro, A., Hasegawa, T., Tsuru, M. & Yuji, O. Locating Congested Segments over the Internet Based on Multiple End-To-End Path Measurements. *IEICE Transactions on Communications* **89,** 1099–1109 (2006) (cit. on p. 56).

77. Zhang, J., Xi, K., Zhang, L. & Chao, H. J. *Optimizing Network Performance using Weighted Multipath Routing* in *Proceedings of the ICCCN Conference* (2012), 1–7 (cit. on p. 56).

78. Lantz, B., Heller, B. & McKeown, N. *A Network in a Laptop: Rapid Prototyping for Software-Defined Networks* in *Proceedings of the Hotnets Workshop* (2010), 19:1–19:6 (cit. on p. 59).

79. Sundaresan, S., De Donato, W., Feamster, N., Teixeira, R., Crawford, S. & Pescapè, A. *Broadband Internet Performance: A View from The Gateway* in *Proceedings of the SIGCOMM* (2011), 134–145 (cit. on p. 59).

80. Sommers, J. & Barford, P. *Cell vs. WiFi: On the Performance of Metro area Mobile Connections* in *Proceedings of the IMC* (2012), 301–314 (cit. on p. 60).

81. *Cisco Visual Networking Index* `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/complete-white-paper-c11-481360.pdf` (cit. on p. 71).

82. Hiertz, G. R., Denteneer, D., Stibor, L., Zang, Y., Costa, X. P. & Walke, B. The IEEE 802.11 universe. *IEEE Communications Magazine* **48** (2010) (cit. on p. 71).

83. Langley, A. *et al. The QUIC transport protocol: Design and Internet-scale deployment* in *Proceedings of the SIGCOMM* (2017), 183–196 (cit. on p. 72).

84. *Sants-UPC Community Newtork* `http://sants.guifi.net` (cit. on p. 73).

85. *Community Networks Testbed for the Future Internet, CONFINE* `http://confine-project.eu/`. FP7 European Project 288535 (cit. on p. 73).

86. *OpenWrt Linux distro. for embedded devices* `https://openwrt.org` (cit. on p. 73).

87. *Quick Mesh Project* `http://qmp.cat` (cit. on p. 73).

88. Cerdà-Alabern, L., Neumann, A. & Maccari, L. *Experimental evaluation of bmx6 routing metrics in a 802.11 an wireless-community mesh network* in *Future Internet of Things and Cloud (FiCloud), 2015 3rd International Conference on* (2015), 770–775 (cit. on p. 73).

89. *Open, Free and Neutral Network Internet for everybody* `http://guifi.net/en` (cit. on p. 73).

90. Cerdà-Alabern, L., Neumann, A. & Escrich, P. *Experimental Evaluation of a Wireless Community Mesh Network* in *The 16th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM'13* (ACM, Barcelona, Spain, Nov. 2013) (cit. on p. 73).

91. *qMp Sants-UPC monitoring page* `http://dsg.ac.upc.edu/qmpsu` (cit. on p. 73).

92. Carlucci, G., De Cicco, L. & Mascolo, S. *HTTP over UDP: an Experimental Investigation of QUIC* in *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (2015), 609–614 (cit. on p. 78).

93. Megyesi, P., Krämer, Z. & Molnár, S. *How quick is QUIC?* in *Communications (ICC), 2016 IEEE International Conference on* (2016), 1–6 (cit. on p. 78).

94. Gusella, R. Characterizing the variability of arrival processes with indexes of dispersion. *IEEE Journal on Selected Areas in Communications* **9,** 203–211 (1991) (cit. on p. 79).

95. Witten, I. H., Frank, E., Hall, M. A. & Pal, C. J. *Data Mining: Practical machine learning tools and techniques* (Morgan Kaufmann, 2016) (cit. on p. 80).

96. Gratzer, F. QUIC-quick UDP internet connections. *Future Internet and Innovative Internet Technologies and Mobile Communications* (2016) (cit. on p. 85).

97. Timmerer, C. & Bertoni, A. Advanced transport options for the dynamic adaptive streaming over HTTP. *arXiv preprint arXiv:1606.00264* (2016) (cit. on p. 94).