End-to-end mechanisms to improve latency in communication networks

Baptiste Jonglez

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Baptiste JONGLEZ

Thèse dirigée par **Martin HEUSSE**
et codirigée par **Bruno GAUJAL**

préparée au sein du **Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Mécanismes de bout en bout pour améliorer la latence dans les réseaux de communication

End-to-end mechanisms to improve latency in communication networks

Thèse soutenue publiquement le **23 octobre 2020**,
devant le jury composé de :

Monsieur Martin HEUSSE
Professeur des Universités, Grenoble INP, Directeur de thèse
Monsieur Bruno GAUJAL
Directeur de Recherche HDR, Inria, Co-Directeur de thèse
Monsieur André-Luc BEYLOT
Professeur des Universités, INP Toulouse - ENSEEIHT, Rapporteur
Monsieur Guillaume URVOY-KELLER
Professeur des Universités, Université Côte d'Azur, Rapporteur
Madame Isabelle GUÉRIN LASSOUS
Professeur des Universités, Université Lyon 1, Présidente
Madame Anna BRUNSTRÖM
Professeur, Université de Karlstad - Suède, Examinatrice

Abstract

The network technologies that underpin the Internet have evolved significantly over the last decades, but one aspect of network performance has remained relatively unchanged: latency. In 25 years, the typical capacity or "bandwidth" of transmission technologies has increased by 5 orders of magnitude, while latency has barely improved by an order of magnitude. Indeed, there are hard limits on latency, such as the propagation delay which remains ultimately bounded by the speed of light.

This diverging evolution between capacity and latency is having a profound impact on protocol design and performance, especially in the area of transport protocols. It indirectly caused the Bufferbloat problem, whereby router buffers are persistently full, increasing latency even more. In addition, the requirements of end-users have changed, and they expect applications to be much more reactive. As a result, new techniques are needed to reduce the latency experienced by end-hosts.

This thesis aims at reducing the experienced latency by using end-to-end mechanisms, as opposed to "infrastructure" mechanisms. Two end-to-end mechanisms are proposed. The first is to multiplex several messages or data flows into a single persistent connection. This allows better measurements of network conditions (latency, packet loss); this, in turn, enables better adaptation such as faster retransmission. I applied this technique to DNS messages, where I show that it significantly improves end-to-end latency in case of packet loss. However, depending on the transport protocol used, messages can suffer from Head-of-Line blocking: this problem can be solved by using QUIC or SCTP instead of TCP.

The second proposed mechanism is to exploit multiple network paths (such as Wi-Fi, wired Ethernet, 4G). The idea is to use low-latency paths for latency-sensitive network traffic, while bulk traffic can still exploit the aggregated capacity of all paths. This idea was partially realized by Multipath TCP, but it lacks support for multiplexing. Adding multiplexing allows data flows to cooperate and ensures that the scheduler has better visibility on the needs of individual data flows. This effectively amounts to a scheduling problem that was identified only very recently in the literature as "stream-aware multipath scheduling". My first contribution is to model this scheduling problem. As a second contribution, I proposed a new stream-aware multipath scheduler, SRPT-ECF, that improves the performance of small flows without impacting larger flows. This scheduler could be implemented as part of a MPQUIC (Multipath QUIC) implementation. More generally, these results open new opportunities for cooperation between flows, with applications such as improving WAN aggregation.

Résumé

Les technologies réseau qui font fonctionner Internet ont beaucoup évolué depuis ses débuts, mais il y a un aspect de la performance des réseaux qui a peu évolué : la latence. En 25 ans, le débit disponible en couche physique a augmenté de 5 ordres de grandeur, tandis que la latence s'est à peine améliorée d'un ordre de grandeur. La latence est en effet limitée par des contraintes physiques fortes comme la vitesse de la lumière.

Cette évolution différenciée du débit et de la latence a un impact important sur la conception des protocoles et leur performance, et notamment sur les protocoles de transport comme TCP. En particulier, cette évolution est indirectement responsable du phénomène de "Bufferbloat" qui remplit les tampons des routeurs et exacerbe encore davantage le problème de la latence. De plus, les utilisateurs sont de plus en plus demandeurs d'applications très réactives. En conséquence, il est nécessaire d'introduire des nouvelles techniques pour réduire la latence ressentie par les utilisateurs.

Le but de cette thèse est de réduire la latence ressentie en utilisant des mécanismes de bout en bout, par opposition aux mécanismes d'infrastructure réseau. Deux mécanismes de bout en bout sont proposés. Le premier consiste à multiplexer plusieurs messages ou flux de données dans une unique connexion persistante. Cela permet de mesurer plus finement les conditions du réseau (latence, pertes de paquet) et de mieux s'y adapter, par exemple avec de meilleures retransmissions. J'ai appliqué cette technique à DNS et je montre que la latence de bout en bout est grandement améliorée en cas de perte de paquet. Cependant, en utilisant un protocole comme TCP, il peut se produire un phénomène de blocage en ligne qui dégrade les performances. Il est possible d'utiliser QUIC ou SCTP pour s'affranchir de ce problème.

Le second mécanisme proposé consiste à exploiter plusieurs chemins, par exemple du Wi-Fi, une connexion filaire, et de la 4G. L'idée est d'utiliser les chemins de faible latence pour transporter le trafic sensible en priorité, tandis que le reste du trafic peut profiter de la capacité combinée des différents chemins. Multipath TCP implémente en partie cette idée, mais ne tient pas compte du multiplexage. Intégrer le multiplexage donne davantage de visibilité au scheduler sur les besoins des flux de données, et permettrait à ceux-ci de coopérer. Au final, on obtient un problème d'ordonnancement qui a été identifié très récemment, "l'ordonnancement multi-chemins sensible aux flux". Ma première contribution est de modéliser ce problème. Ma seconde contribution consiste à proposer un nouvel algorithme d'ordonnancement pour ce problème, SRPT-ECF, qui améliore la performances des petits flux de données sans impacter celle des autres flux. Cet algorithme pourrait être utilisé dans une implémentation de MPQUIC (Multipath QUIC). De façon plus générale,

ces résultats ouvrent des perspectives sur la coopération entre flux de données, avec des applications comme l'agrégation transparente de connexions Internet.

Remerciements

Cette thèse doit beaucoup aux échanges nourris et constructifs que j'ai eu la chance d'avoir avec mes deux directeurs de thèse, Martin Heusse et Bruno Gaujal, et je les en remercie. Ils ont su accorder le temps nécessaire pour se rencontrer tous les trois régulièrement et me permettre ensuite de faire fructifier ces réflexions. Leurs domaines de spécialité bien différents n'a pas toujours facilité les échanges, mais il en est systématiquement ressorti des points de vue complémentaires sur les problèmes abordés.

Merci à Sinan Birbalta que j'ai eu le plaisir d'encadrer comme stagiaire de master et qui a travaillé sur des expériences préliminaires à mes travaux de thèse.

Merci à Maciej Korczynski et Yevheniya Nosyk de m'avoir impliqué dans leur projet « Source Address Validation » qui a donné de beaux résultats : j'ai trouvé du plaisir à participer au projet, et j'ai également apprécié de pouvoir travailler sur un autre sujet quand j'étais bloqué sur mon sujet de thèse principal.

Avec presque 400 heures d'enseignement tout au long de ma thèse, j'ai évidemment beaucoup collaboré avec mes collègues de l'Ensimag : outre Martin Heusse, merci à Roland Groz, Andrzej Duda, Olivier Alphand, Franck Rousseau, Grégory Mounié, Matthieu Moy et Frédéric Wagner de m'avoir donné l'opportunité de travailler ensemble. Sans oublier les services administratifs de l'Ensimag ainsi que le service informatique que j'ai souvent sollicité.

Je tiens également à remercier Juliusz Chroboczek et Matthieu Boutier de l'Université Paris Diderot pour m'avoir permis de mettre un pied dans la recherche en réseau il y a maintenant 7 ans. De même, je suis reconnaissant à l'École Normale Supérieure de Lyon pour sa formation « par et pour la recherche » et pour m'avoir donné l'opportunité de poursuivre cette aventure académique avec un financement de thèse.

Tout au long de ma thèse, les collègues de mes deux équipes de rattachement, Drakkar et Polaris, ont été essentiels à une vie d'équipe riche et à une ouverture sur des sujets de recherche différents des miens. Merci en particulier à Takwa, Ulysse, Henry, Timothy, Pierre et Etienne pour le grand nombre de discussions passionnantes durant ces quelques années passées ensemble. Merci également au

personnel administratif du Laboratoire d'Informatique de Grenoble, toujours très professionnel, et en particulier à Pascale Poulet, Alexandra Guidi et Annie Simon pour leur capacité à surmonter les problèmes de façon particulièrement efficace.

Enfin, merci à ma famille de m'avoir soutenu dans ce projet malgré la distance, et merci à Oriane de m'avoir accompagné toutes ces années, et particulièrement d'avoir réussi à survivre à un confinement en compagnie d'un doctorant en fin de rédaction.

Contents

1 Introduction

1.1 Latency in modern communication networks

The Internet has seen major evolutions over the years: new services have been deployed, mobile and wireless usage has exploded, and the transmission capacity of common technologies — the "bandwidth" of your Wi-Fi network or Internet connection — has increased by 5 orders of magnitude in 25 years [16]. However, **latency** is one aspect of communication networks that has remained relatively unchanged because of strong physical constraints. Improvements have been made, but in a much less spectacular way than the growth in capacity: in 1987, the typical delay through the ARPANET network was around 340 ms of round-trip time (RTT) [63]. Today, the typical round-trip time through the continental US is around 60 ms: for this specific example, latency has seen an improvement of less than an order of magnitude in 33 years. There are hard limits on latency, such as the propagation delay which remains ultimately bounded by the speed of light.

With the stagnation of latency causing such an imbalance between capacity and latency, new problems are arising. The Bandwidth-Delay Product (BDP) has become larger and larger, basically following the increase in network capacity. As a result, traditional congestion control algorithms that were designed in a small-BDP environment have a hard time adapting to a high-BDP environment, and cannot take advantage of high-capacity paths. This has led to the development of more aggressive congestion control algorithms for TCP, such as Cubic [37] or BBR [14]. These new algorithms can indeed deliver the full expected throughput on high-capacity paths; however, they can also cause a significant amount of congestion [39]. This congestion, in turn, increases queuing delays and thus the overall amount of latency. In fact, Cubic may well be responsible for causing, or at least exposing, bufferbloat [2, 48].

This motivates efforts aimed at reducing latency, which can be done in many different ways. This work is focused on **reducing latency through end-to-end mechanisms**.

1.1.1 Experienced latency

In this work, I am mostly interested in the *overall latency* experienced by an application. The application is running on a *terminal*, also called *end-host* (a laptop, a mobile phone, a connected device...). It communicates through a network with a remote application: this remote application typically runs on a server, but it can also run on another terminal when using peer-to-peer protocols.

The overall latency includes delays caused by the network itself, as well as those caused in the end-host (buffers, transport protocol, operating system).

A simple way to approximate the overall experienced latency is to measure the Round-Trip Time (RTT). It can be done by sending a small data packet that goes to the destination host and immediately elicits a small data packet as response. This is what a tool like `ping` measures. However, such a measurement technique ignores delays coming from the transport protocol, from moving data between the operating system and the application, or from the data processing happening on the remote host. To obtain more accurate measurements, the overall latency must be measured between the local application and the remote application.

1.1.2 Impact of latency on applications

Latency is becoming the new performance bottleneck for many applications. This is especially true for new interactive applications such as Virtual Reality (VR). VR needs low end-to-end latency between user motion and the corresponding visual feedback, sometimes called "motion-to-photon" latency. If this latency is too high, the user may experience impaired motor performance or even nausea or headaches. The maximum acceptable end-to-end latency ranges from 50-75 ms [1, 106] to as low as 15-20 ms for Head-Mounted Displays [24, 70]. When graphical rendering is done remotely through a communication network, many steps must fit within such a tight latency budget: motion sensor acquisition, round-trip network communication, graphical rendering, and display update. Even with state-of-the-art equipment for the other steps, this leaves a latency budget of only 14 to 40 ms for network communication.

Traditional applications that involve bulk transfers of data, such as web applications, are also highly affected by latency. When transferring small amounts of data over a high-capacity networks, the transfer time is dominated by two factors: the initialisation phase (e.g. TCP's three-way handshake), and the slow-start mechanism used to probe available capacity and avoid congestion. Both factors are directly

related to the end-to-end latency: when latency gets higher, feedback takes more time to be received from the remote end of the connection. For instance, with a 40 ms RTT and an initial window of 4380 bytes, it takes 10-11 round-trips for TCP's slow-start algorithm to ramp up to 1 Gbit/s in ideal conditions. During these 10 first round-trips, only 4.3 MiB of data is transferred, yielding an average throughput of 90 Mbit/s. This example shows that a standalone transfer of a few megabytes cannot achieve 1 Gbit/s of throughput in the presence of moderate latency.

More generally, the communication pattern of applications can be classified in two categories [85]:

throughput-bound communication or "greedy streams" [85]: in this category, the application almost always has more data to send. Thus, the throughput is typically limited by the TCP congestion window. This category of communication is characterized by large packet size and high packet rate. The main use-case is bulk transfer of data, such as downloading or uploading a file.

thin-stream communication In this category, the application only has sporadic data to send. It is characterized by small packet size, large packet inter-arrival time, and low packet rate. This includes applications such as online gaming, instant messaging, voice-over-IP, DNS, and more generally any application that communicates through a series of small messages or requests/responses.

This distinction has a major implication on choosing the appropriate metric to study performance. For throughput-bound communication, the main metric is the effective throughput, which can be alternatively measured as the *completion time* of each bulk transfer: the average throughput is the total size divided by the completion time. Even though these applications are throughput-bound, the completion time can still depend on latency, as we have seen in the small calculation above. The completion time is typically a large multiple of the RTT. In Chapter 4, I optimize the completion time of individual "streams" that can potentially transfer a large amount of data.

For thin-stream communication, the main metric is the latency of each individual message (which can be a request, response, or a message with any other semantic). Since messages are typically small, the transmission time is usually negligible: in this case, overall latency is dominated by sources of latency coming from the network, such as propagation delay or queuing delay. For instance, I use the latency metric in Chapter 3 to evaluate the performance of DNS. DNS is clearly a thin-stream application because it uses small requests and responses and requires a low packet rate.

Note that, depending on the network conditions — capacity, latency — the same application could belong to one category or the other. For example, sending regular messages of size 50 KB can be considered thin-stream if the network capacity is large enough. With a 100 Mbps link and 40 ms of RTT, assuming perfect congestion control and no loss, transmitting a message and receiving the acknowledgment takes 44 ms[1], which is close to the RTT. However, on a 1 Mbps link with 40 ms of RTT, transmitting the same message would take 440 ms. Since it takes several RTT, this becomes a throughput-bound communication.

Overall, when network capacity increases, applications tend to be more limited by latency. As a result, latency becomes a challenging performance bottleneck for many applications.

1.2 Reducing latency

The goal of this work is to reduce the overall latency experienced by end-hosts. There are many sources of latency, and as a result there are many possible ways to reduce latency. These sources and possible remediation are summarized in a comprehensive survey by Briscoe et al. [12]. For the sake of this introduction, the possible ways to reduce latency can be loosely categorized in two broad families: *infrastructure mechanisms* and *end-to-end mechanisms*. Both families are briefly described below, while Chapter 2 delves more into the details of end-to-end mechanisms.

1.2.1 Infrastructure mechanisms to improve latency

The first family of mechanisms that can improve latency are *infrastructure mechanisms*, i.e. any mechanism that is not directly accessible to the end-hosts. This includes any improvements to the structure of the Internet or its supporting technologies.

For instance, it is possible to improve Internet routing to provide shorter geographical routes. This can be done by careful configuration of BGP routing policies, or by developing local peering points (Internet eXchange Points or IXP). This touches on the structure of the network.

[1]The transmission delay is $4\ ms = \frac{50\ KB*8}{100\ Mbps}$

It is also possible to move content closer to the user thanks to caches or CDNs (Content Delivery Networks). Even going further, there are proposals to move computation capabilities closer to the user (Edge Computing).

Better technologies can improve latency as well: FTTH (Fiber-to-the-home) provides much lower local-loop latency compared to xDSL (Digital Subscriber Line), because it uses less elaborate mechanisms to overcome noise and interference — for instance, xDSL usually implements an interleaving mechanism to mitigate bursts of noise, which increases the transmission latency. As an other example, Low-Earth-Orbit satellite systems (LEO) provide much lower propagation latency than communication satellites in geostationary orbit. One such LEO system, StarLink, has recently demonstrated end-to-end latency of 30 ms [103] in the continental United States. In contrast, typical end-to-end latency through a geostationary satellite is around 600 ms. Given this large difference in end-to-end latency and the general need for lower latency, it is not surprising that LEO satellite systems are being deployed at a fast pace.

Lastly, Active Queue Management (AQM) can be used in ISP-managed routers to better manage their buffers in case of congestion. This can help avoid bufferbloat [42].

All these mechanisms must be deployed by ISPs or infrastructure providers. Once deployed, they can have a significant impact by affecting many customers at once. But they often require large amounts of time and money to be deployed, which may limit their usage.

1.2.2 End-to-end mechanisms to improve latency

The second family consists of *end-to-end mechanisms*, i.e. mechanisms mostly implemented in terminals (computer, smartphone...). These mechanisms can be deployed relatively easily through software updates or configuration changes, although it may take significant time to reach a large-scale roll-out.

Many of these mechanisms that can reduce latency are found in the operating system of end-hosts: drivers for network interface cards, buffer management, transport protocol implementation or parameters... Some mechanisms are also accessible to the applications: for instance, choosing an appropriate transport protocol or appropriate transport protocol options, managing events without blocking... Lastly, exploiting multi-homing to reduce latency is a transversal mechanism that can be done either in the operating system or the application.

Overall, the two families of mechanisms (infrastructure and end-to-end) are complementary. For instance, in the current Internet, better routing can only be done in the infrastructure, while exploiting multi-homing is only possible end-to-end.

Exploiting end-to-mechanisms is the main approach adopted in this work. These mechanisms are easier to deploy and match the end-to-end model of the Internet, in which most of the intelligence is located at the edge of the network.

1.3 Multi-homing and multipath communication

Multipath transport protocols have emerged in the last decades thanks to the widespread availability of connectivity options. It is quite common to have simultaneous access to several network connections: home Wi-Fi, neigbhour's Wi-Fi, mobile connectivity, fiber, DSL... This is called a *multi-homing* situation, illustrated in Figure 1.1. The end-host device is physically attached to several networks, and can exploit these attachment points to use several paths to communicate with a destination. The device can dynamically choose which path it will use to communicate, based on various metrics: monetary cost of mobile data, expected performance, measured real-time performance, physical signal level, and so on. When using a multipath-aware transport protocol such as Multipath TCP [88] or CMT-SCTP [49], it is even possible to use both paths at the same time and load-balance between them to optimize a performance metric.

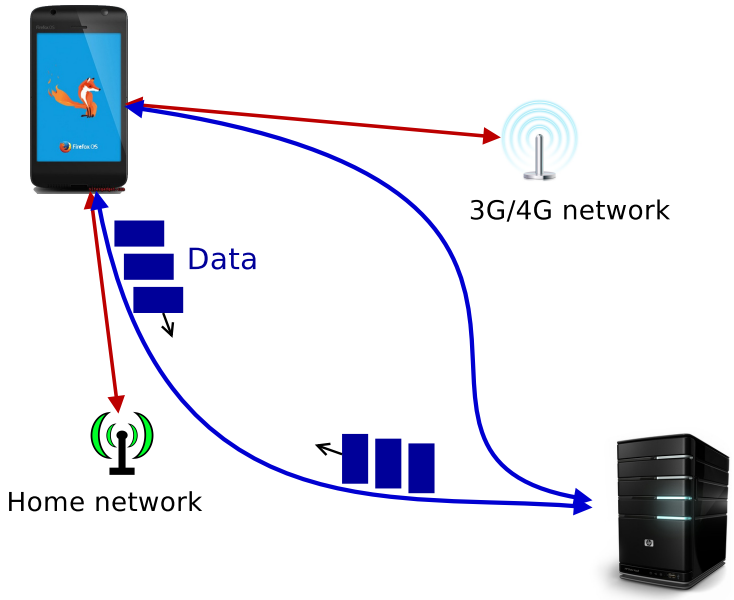

Figure 1.1 An end-host device (smartphone) in a *multi-homing* situation.

Multi-homing connectivity can also be exploited from the home router, as illustrated in Figure 1.2. In this case, the end-host device is connected to a single home router, which is itself attached to several networks. This is the multi-homing model considered by the IETF [4], in particular in the Home Networking working group [17]. In this model, the local network runs several parallel addressing schemes, one for each upstream ISP, using IP address space delegated by the ISPs. Thus, the device can select which network it will use by just setting the appropriate source address in its outgoing packets. Alternatively, all control over which path to use can be delegated to the home router, using Network Address Translation (NAT): this setup is then known as SD-WAN (Software-Defined Wide Area Network).

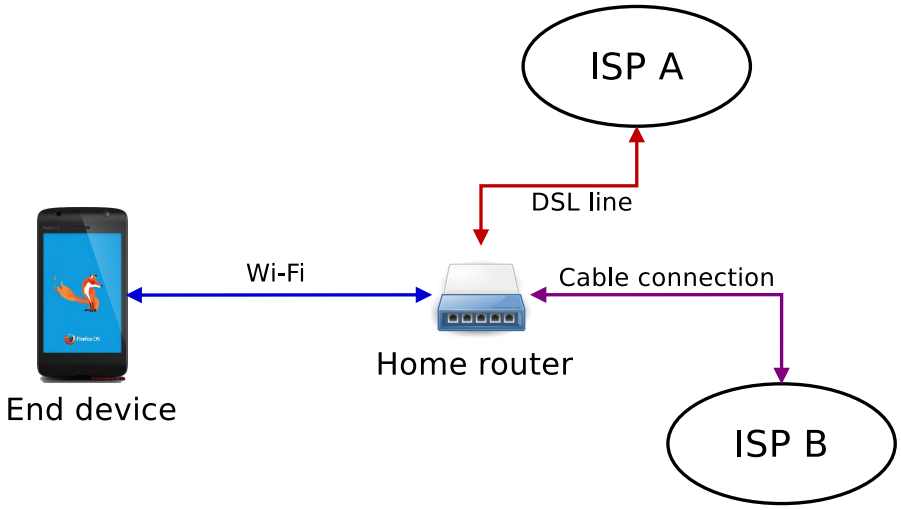

Figure 1.2 An alternative "network-based" multi-homing situation, where the home router has several network connections, potentially with different ISPs (Internet Service Providers).

Multipath networks are a much more general concept. For instance, multipath is an essential component of modern datacenter networks [88] to provide redundant paths between pairs of servers. Similarly, large-scale network operators exploit multiple paths in their core networks to load-balance traffic and cope with failures. However, since this work is mostly focused on end-user devices and not in the core network, I will not consider these situations.

1.3.1 Reducing latency with multi-homing

The diversity of connectivity options offered by multi-homing has been exploited in several ways:

1. to improve throughput by aggregating the capacity of several network paths [81];

2. to increase reliability by quickly re-routing traffic to a different path in case of disruption: for example, Multipath TCP has been massively deployed by Apple to improve the reliability of Siri's communication [18];

3. to improve reliability and latency by duplicating traffic on several network paths [34, 25, 69].

In the context of reducing overall latency from the end-hosts, the second and third items are the most promising. Recovering from disruption, such as temporary loss of connectivity or mobility, is an area where TCP is lacking.

But as Chapter 4 will show, simply having several paths to choose from can be enough to improve latency, even without duplicating traffic.

1.4 Methodology

Two main methods are adopted in this work. The first one is to experiment with real networked systems using testbeds. Indeed, operating systems and network protocols are so complex that simulation may not yield realistic results. On the other hand, experimenting with real systems "in the wild" is notoriously difficult and does not easily allow to isolate interesting behaviors. Using testbeds is a middle-ground that is expected to give more realistic results than simulation, but still allows some control over experimental parameters. Experiments on testbeds still take a considerable amount of time to setup, especially when taking efforts to make them reproducible. Some of these efforts are described in Section 3.4.3 of Chapter 3.

In particular, I worked with several large-scale testbeds such as Grid'5000 [5] or the NLNOG ring [79]. I also setup small testbeds that were more controlled, allowing me to study specific effects in more details.

The second method is to develop theoretical models and algorithms. This is mostly used in Chapter 4 where it is applied to a scheduling problem. In that case, evaluation is done through simulation: I used HTTP/2 traces that I replay in a simple network model. The simulation setup is described in details in Section 4.7.2.

1.5 Overview of contributions

1.5.1 Reducing latency with better routing

My first work in this area was to improve routing by taking into account end-to-end latency. Traditionally, a routing protocol computes paths in a network without taking into account the actual traffic. For example, a sub-optimal routing choice may lead to congestion, but routing protocols such as BGP or OSPF have no way to realize this. It is still possible to take into account the latency of network links, by encoding it in the *link weight* used by the routing protocol to compute shortest paths. However, this requires manual tuning and only provides a *static* view of latency: the routing protocol will still be oblivious to an increase of end-to-end latency caused by congestion.

My proposal is to dynamically compute paths based on end-to-end latency measurements. This allows to always find the path with lowest latency, even in a dynamic network with changing traffic patterns. The most challenging issue of this approach is *stability*: how to prevent the network from oscillating between paths when the traffic load causes congestion? I leverage recent advances in game theory to design an algorithm that provably converges to a near-optimal point, without oscillating indefinitely. There are also other challenges to solve to make this method practical, for instance: how can a router in the middle of the network measure the end-to-end latency experienced by a user?

This work led to a publication at ITC29 [52] (International Teletraffic Congress). The article mixes a theoretical solution based on game theory with an actual implementation and evaluation of the proposed routing scheme.

While related to latency, this work is based on an infrastructure mechanism: routing. As such, it will not be detailed further in this thesis. However, it was still an important milestone to better understand end-to-end latency and led to the development of the other contributions described in this thesis.

Publications

B. Jonglez and B. Gaujal. “Distributed and Adaptive Routing Based on Game Theory”. In: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 1. Sept. 2017, pp. 1–9. DOI: 10.23919/ITC.2017.8064333

Baptiste Jonglez and Bruno Gaujal. “Distributed and Adaptive Routing Based on Game Theory”. In: *ALGOTEL 2017 - 19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*. Quiberon, France, May 2017. URL: https://hal.archives-ouvertes.fr/hal-01517911

Baptiste Jonglez and Bruno Gaujal. *Distributed Adaptive Routing in Communication Networks*. en. report. Inria ; Univ. Grenoble Alpes, Oct. 2016, p. 25. URL: https://hal.inria.fr/hal-01386832

1.5.2 Performance of persistent DNS connections

Chapter 3 provides a first look at protocol performance with DNS (the Domain Name System). The goal is to analyse the performance requirements of DNS applications, whether these requirements are fulfilled adequately, and how to improve DNS performance at the transport layer.

DNS is an excellent use-case because it needs both low-latency and reliability: this is uncommon and not easy to achieve. Current stub resolver implementations mostly use UDP, with a very naive retransmission strategy that does not take into account latency or processing delays. As a result, a single lost DNS query produces an additional delay of 5 seconds (Linux, Android), which can be prohibitive for many applications.

To improve this situation, I leverage an end-to-end mechanism by using a more suitable transport protocol. More specifically, using *persistent* TCP connections allows to adapt the retransmission timer to the measured Round-Trip Time, leading to much lower delay in case of query loss. I validate this intuition on a small testbed, providing detailed explanations of the specific mechanisms that impact the latency of DNS-over-TCP.

The second question addressed in this chapter is related to scalability: switching to DNS-over-TCP brings benefits for the end-user, but can it work at a large scale? Will it overload the DNS infrastructure such as recursive resolvers? Indeed, TCP is a much more complex protocol than UDP and requires more resources such as CPU and memory.

To answer this second question, I setup a large-scale series of experiments on Grid’5000 [5]. The results of these experiments allowed me to analyse the performance impact of large-scale persistent DNS connections on recursive resolvers, with up to millions of simultaneous TCP clients connecting to a single recursive resolver.

Publications

Baptiste Jonglez et al. “Poster: persistent DNS connections for improved performance”. In: *2019 IFIP Networking Conference, Networking 2019, Warsaw, Poland, May 20-22, 2019*. IEEE, 2019, pp. 1–2. DOI: 10.23919/IFIPNetworking46909.2019.8999394

Baptiste Jonglez et al. *Improving end-to-end delay at the application layer*. International Summer School on Latency Control for Internet of Services. Poster. June 2017. URL: https://hal.inria.fr/hal-01632191

1.5.3 Stream-aware multipath scheduling

Lastly, Chapter 4 is dedicated to multipath scheduling in the multi-homing context described in Section 1.3. The main idea is to extend existing multipath scheduling algorithms, as used by Multipath TCP, to the new case of multi-stream transport protocol. It provides increased benefits for applications that multiplex several flows of data in the same connection, such as HTTP/2. This recent approach has been called “stream-aware” multipath scheduling in [87].

More specifically, I identify two key advantages when taking into account application-level data flows in a multipath scheduler:

1. the receiver can avoid Head-of-Line blocking between unrelated data flows, a very common issue with Multipath TCP;

2. the scheduler can take more informed decisions, for instance by prioritizing short flows.

Based on these observations, I introduce a new stream-aware multipath scheduler called SRPT-ECF, and show that it has good properties. Most notably, it minimises the stream completion time in ideal network conditions. I then evaluate it on a HTTP/2 trace.

This work is mainly theoretical and high-level, but the scheduling model is inspired from both SCTP and MPQUIC. As a result, this new scheduling algorithm should be readily implementable in MPQUIC.

Publications

Baptiste Jonglez et al. “SRPT-ECF: challenging Round-Robin for stream-aware multipath scheduling”. In: *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*. IEEE, 2020, pp. 719–724. URL: https://ieeexplore.ieee.org/document/9142713

1.6 Outline

This thesis is organized as follows. Chapter 2 provides more context on end-to-end latency, and focuses on two main mechanisms: *multiplexing* several data flows, and using *multiple paths* to transmit data. Each mechanism is analyzed and its impact on latency — positive as well as negative — is assessed.

Chapter 3 presents the first contribution: multiplexing DNS requests on a persistent connection to improve latency. The performance of this proposal is analyzed from two perspectives: end-user latency on the one hand, and infrastructure load on the other hand.

Chapter 4 combines multiplexing and multi-homing to shed light on a recently identified scheduling problem, *stream-aware multipath scheduling*, by formalizing it. I then propose a new algorithm to solve this scheduling problem: SRPT-ECF. I show that it has interesting theoretical properties and behaves well in a simulation based on HTTP/2 traffic traces.

Finally, Chapter 5 summarizes the contributions of this thesis compared to recent works and draws some perspectives for future work.

2 Dissecting end-to-end latency

This chapter delves into end-to-end latency to better understand its cause and possible ways to improve it. After a brief overview of the different sources of latency related to end-hosts in Section 2.1, I focus on an important aspect of modern applications and transport protocols: *multiplexing*. Section 2.2 defines multiplexing and the possible models for an application to communicate through a transport protocol: message-oriented, stream-oriented, hybrid. Section 2.3 then analyses the impact of multiplexing on latency and illustrate important concepts that will be used throughout all chapters, such as Head-of-Line blocking.

Lastly, Section 2.4 discusses the latency impact of *multi-homing*. Using multiple paths provides several opportunities that can help improve latency, but it also comes with its own set of challenges.

2.1 End-host sources of latency in a network

In a very complete survey [12], Briscoe *et. al* identify and classify the main sources of latency in a network. The authors also explore possible techniques for reducing latency for each source. Among all identified sources of latency, the ones directly related to the end hosts include:

Transport session initialization (III-A [12]) Establishing a transport session to a remote host often needs several Round-Trips (RTT). For instance, the three-way handshake of TCP requires a full RTT before the host is able to send any application-level data, and TLS (Transport Layer Security) adds two more RTT to the initialization procedure. For transfers of small amounts of data, latency is thus dominated by the initialization phase. This is made worse by the fact that TCP SYN are lost more often than other packets in the Internet [11]: as a result, TCP will fall back to multi-seconds retransmission timeouts, leading to catastrophic latency for the end-user. Solutions include reducing the number of RTTs needed to establish a session (TCP Fast Open [15], TLS session resumption), possibly by integrating encryption directly

into the transport layer with protocols such as TCPcrypt [9] or QUIC [66], and better queuing management [11].

End-host congestion control algorithm and its induced queuing delay in the network (IV-F [12]) When an end host uses an aggressive congestion control algorithm such as Cubic [37] or BBR [14], it will always try to fill buffers at the bottleneck link, causing large delays both for itself (self-induced latency) and for other flows. This mostly matters for "bulk" transfers that try to send data as fast as possible through the network: interactive applications will generally never fill the congestion control window. However, applications that have an intermittent transmission pattern will suffer from the lack of *Congestion Window Validation* (Section V-D of the survey), where the congestion window grows without actually being tested for congestion. This creates episodes of self-induced congestion when the application transmits a large burst of data, leading to high delays and packet loss. A new validation mechanism [93] attempts to solve this issue without impacting latency-sensitive applications.

Packet loss recovery (III-C [12]) Reliable transports need to detect and recover losses. Two main mechanisms are used: fast retransmit and retransmission timeout. Triggering a retransmission timeout is undesirable, because it can take several seconds to recover from a loss. In contrast, fast retransmit can detect and repair a loss in just one RTT, as shown in Figure 2.1. However, it only works when the lost packet is immediately followed by several more packets. This is especially problematic for the last packet of a bulk data transfer, or for interactive "thin stream" applications [85] that sporadically send small amount of data: they can only recover from a loss through a costly retransmission timeout. Techniques such as Early Retransmit [3] and Tail-Loss Probes [30] have been developed to mitigate this problem.

Head-of-line blocking at the transport layer (VI-B [12]) This complex subject is related to *application data multiplexing*. This subject is covered in details in Section 2.2.

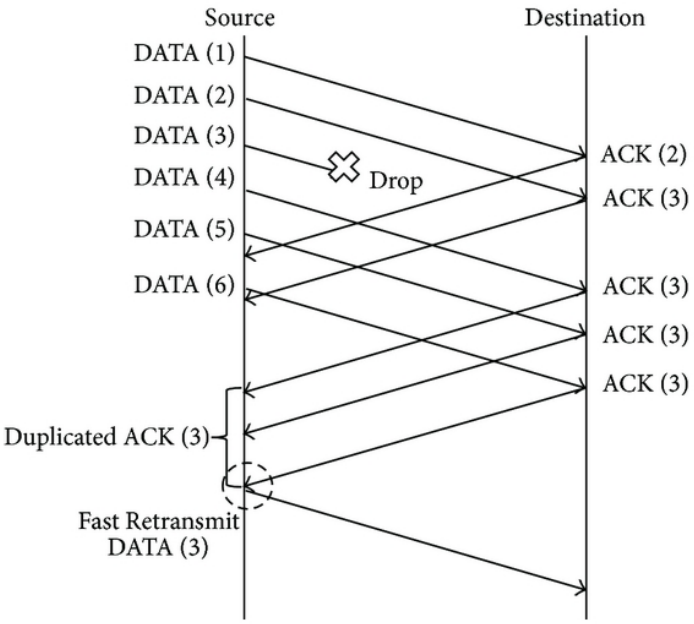

Figure 2.1 Fast retransmit in TCP: when receiving several duplicate ACKs, the missing segment is retransmitted immediately. Figure from [67].

2.2 Multiplexing messages on a connection

This section defines the notion of *multiplexing* used by applications to exchange multiple flows of data within a single connection. It also focuses on the possible communication models between the application and the transport protocol: message-oriented, stream-oriented, and how multiplexing can be introduced in these models. This is an essential pre-requisite to understand performance issues related to multiplexing.

2.2.1 The need for application data multiplexing

Applications commonly need to manipulate, send and receive several kind of *flows of data*, even with the same remote host. Examples include: managing a control flow and one or more data flows (FTP, Bittorrent); loading several HTTP resources from a server; tunnelling several TCP flows within a SSH connection; sending audio, video and text to a video-conferencing server.

Historically, a protocol like FTP opens a TCP connection for its control flow, and then opens a new TCP connection each time it needs to transfer the content of a file. The goal is to simplify the protocol and avoid in-band signalling: a data connection

only transfers the content of a file and nothing else. However, this approach can actually complicate things: there is no *fate sharing* between connections, which means that they can fail independently and the application needs to recover from partial failures. Also, this approach does not interact well with NATs and firewalls.

To some extent, a similar approach was taken with HTTP/1.1, where browsers typically open several TCP connections to the same server and balance HTTP requests on these connections. Here, the goal is to limit head-of-line blocking between requests: even with request pipelining, the server must provide replies in the same order as the requests, which means that a reply that takes time to compute might block a reply that is ready to send out. A downside of opening several connections is that it consumes more resources (both on the server and in stateful parts of the network such as NATs and firewalls) and might lead to increased congestion [27]. Note that, contrary to FTP, HTTP/1.1 implementations take an hybrid approach by not opening a new TCP connection for each request. This hybrid approach is a compromise between increased resource usage and high initial latency (if too many connections are used) and potential request blocking (if too few connections are used).

The basic idea of application data multiplexing is to send and receive several application-level flows in a single transport-layer connection. As we have seen through the examples of FTP and HTTP, it allows to reduce latency and can simplify implementations, while reducing resource usage on the remote host and in the network. However, multiplexing can have an impact on the performance of individual flows. Some applications may require strict performance isolation of flows, may need to ensure fairness between flows, may want to prevent starvation of some flows...

Accordingly, there are many different ways of achieving multiplexing, depending on the semantic required by the application and its performance constraints. The two most common families of transport protocol are message-oriented and stream-oriented, detailed below.

2.2.2 Message-oriented semantic

The simplest piece of data that can be manipulated by an application is a *message*. We define a message as an atomic unit of data with a well-defined length, typically short (less than 64 KiB for UDP). The application gives the message to the transport protocol, and expects that the whole message will be delivered atomically to the

application running on the remote host. That is, the message will either be delivered in its entirety, or not delivered at all.[1].

UDP is the simplest message-oriented transport protocol that can run on top of IP, because it provides no extra feature besides port-based multiplexing. DCCP [64] additionally provides congestion control. SCTP [100] provides more features and organises messages into *streams* but is still message-oriented according to our definition. QUIC, although not message-oriented, has been extended to provide an unreliable datagram service [83]. Note that this QUIC extension has not yet been standardized.

In general, there exists several variants of the message-oriented semantic, depending on additional services offered by the transport protocol:

ordering Both UDP and DCCP provide no ordering guarantee between successive messages sent by the application. With SCTP, all messages must belong to a *stream*: any two messages belonging to the same stream are delivered in the same order as they were provided by the sending application. However, there is no ordering guarantee between messages that belong to different streams. In addition, SCTP supports *unordered messages* for which it provides no ordering guarantee, even with respect to other messages from the same stream. Note that the concept of *stream* in SCTP is substantially different from TCP streams.

reliability UDP and DCCP provide unreliable message delivery, while SCTP ensures reliable message delivery. A Partial Reliability [101] extension has been designed for SCTP, allowing the sender to "abandon" some messages even though the receiver was expecting them to guarantee in-order delivery.

Examples of application protocols that use the message-oriented semantic include: DNS, SS7, RTP, SNMP, SIP. With the exception of SS7 that leverages SCTP, most of them simply run on top of UDP. In the context of WebRTC and real-time media, RTP can run on top of SCTP, even though it is encapsulated in UDP in the end to ease deployment.

Multiplexing in the message-oriented semantic

When messages are small, multiplexing is naturally done at the level of messages: send one complete message, then move to the next message, and so on.

[1] As an exception, note that SCTP allows partial delivery to the application when the SCTP buffer runs out of space, see Section 6.9 of [100]. But since SCTP provides reliable delivery, it does not contradict this statement.

However, when messages can be large, it makes sense to split messages into smaller pieces. For instance, SCTP splits messages into DATA chunks, typically sized to fit in a single IP packet [100]. Multiplexing is then performed at the level of DATA chunks. That is, DATA chunks from several messages can be interleaved, allowing concurrent progress of several messages. This provides a much finer granularity and avoids the situation where a big message blocks smaller unrelated messages.

2.2.3 Stream-oriented semantic

When an application uses the stream-oriented semantic, it sends data as one or more *continuous streams of bytes*. The total size of a stream does not need to be known in advance, and it is even possible to send infinite streams of data. The expectation of the sending application is that the exact same stream of data will eventually be delivered to the receiving application: in particular, the order of individual bytes must be preserved.

The transport protocol is responsible for ensuring this semantic. It performs segmentation of application data, ensures reliability through a retransmission algorithm or error-correcting codes, and reorders data bytes on the receiver side. It also provides congestion control and flow control.

The stream-oriented semantic gives no guarantee on data boundaries. For instance, if the sending application gives a single block of 4 KiB of data to the transport protocol, it is possible that the data will be delivered to the receiving application in 2 blocks of 2 KiB. Similarly, several small data blocks might be aggregated and delivered as a single block to the receiving application. Transport-application APIs may allow some control over aggregation and splitting, but they provide no absolute guarantee. Examples include the `TCP_CORK` socket option on Linux, and Nagle's algorithm [76] (`TCP_NODELAY` option). TCP also includes a PUSH flag: when set on a segment, it informs the remote peer that all data in its TCP buffer should be immediately delivered to the receiving application.

Note that stream-oriented protocols are always reliable: the stream abstraction does not make sense if it is not reliable. To work without reliability, the application would need some control over the structure and framing of data, and a guarantee that atomic sets of data are either fully received or fully lost. In other words, it would require a message-oriented semantic, not a stream-oriented semantic.

Multiplexing in the stream-oriented semantic

While TCP only offers a single communication stream to the application, QUIC [66] provides multiplexing with a stream-oriented semantic, effectively providing several streams in a single connection. A stream represents "an elastic message abstraction: a single STREAM frame may create, carry data for, and terminate a stream, or a stream may last the entire duration of a connection" [50].

In practice, data from a stream is transmitted within frames. Just like the DATA chunks of SCTP, QUIC frames fit in a single IP packet and represent the granularity of the multiplexing process.

2.2.4 Hybrid semantic: HTTP

While originally an application protocol, HTTP gained features over time and can be used as a somewhat generic transport protocol. However, it has a "hybrid" semantic that is not clearly either stream-oriented or message-oriented.

The target element manipulated through HTTP is called a *resource*. The actual definition of a resource is intentionally left vague by the HTTP specification [28] for the sake of generality. Resources are communicated or manipulated through a *representation* carried in *HTTP messages*. Such a representation could be, for instance, some text encoded in a given charset or a JSON document, and can optionally be "content-coded" with a transformation such as compression. More formally [28]:

> *A "representation" is information that (...) consists of a set of representation metadata and a* ***potentially unbounded stream*** *of representation data.*

Syntactically, a HTTP message contains *header fields* with metadata, and an optional *message body* with a stream of bytes that encodes a payload body [27]. Thus, within a message, the payload is stream-oriented in nature.

When actually transmitting HTTP messages, several protocol variants are possible, with an impact on application semantic and protocol performance:

- **explicit message body length, allowing persistent connections and pipelined messages**: the regular way for HTTP/1.0 and HTTP/1.1 to send requests and responses with a message body is to include a `Content-Length` header. This header explicitly specifies the total length of the message body. It means that

the sender must know the size of the message in advance: this is suitable for message-oriented communication. Arbitrarily large messages are supported: data can be partially delivered to the "application" (for instance a web rendering process or a storage system) before the entire message has been fully received by HTTP.

Performance-wise, the main advantage of an explicit length header is that the sender can pipeline messages one after the other, although the order of pipelined responses must be the same as the corresponding requests. This pipeline feature was introduced in HTTP/1.1.

- **implicit message length by closing the connection**: when the sender wants to transmit "stream-oriented content" for which it does not know the total length in advance, it can omit the `Content-Length` header. In that case, the message body is terminated by closing the connection. This is only possible for responses, not requests, and is particularly inefficient: it does not allow to reuse connections or pipeline messages. As such, it is recommended to use the *chunked transfer-coding* introduced in HTTP/1.1 instead.

- **chunked transfer-coding**: the recommended alternative for stream-oriented content is to send the message body as a series of self-terminated chunks. This method is applicable to both requests and responses, and is compatible with pipelining and connection reuse. This works by first sending a `Transfer-Encoding: chunked` header. Each chunk can then be sent in sequence, preceded with its length. The last chunk has a length of zero: upon reception of this empty chunk, the receiver knows that the message body is complete. Chunked transfer-coding was introduced in HTTP/1.1 [27].

Multiplexing in HTTP/1

In HTTP/1.0 and HTTP/1.1, the granularity of multiplexing is the HTTP message. That is, a new message can only start when the previous one has been fully transmitted. This is a form of *sequential multiplexing*, which has severe limitations.

In particular, even with pipelining, a web server must send HTTP responses in the same order as the HTTP requests. This produces a form of *Head-of-Line blocking* on the server side, because a fast response maybe be blocked by a slow response. HTTP/2 was designed to overcome this issue.

Multiplexing in HTTP/2

Among other changes, HTTP/2 [6] generalises the notion of chunked transfer and allows full multiplexing between messages.

All messages must be sent in a *stream* as a series of *frames*, which have roughly the same role as chunks had in HTTP/1.1. The difference is that frames from different streams can be freely interleaved, enabling the server to send responses in any order and allowing several messages to progress simultaneously. Frames should be small enough to allow efficient application data multiplexing, because they represent the granularity of the multiplexing process: a sender cannot switch to a different stream in the middle of transmitting a frame. Frame length does not typically exceed 16 KB, although larger frames up to 16 MB are possible [6].

Overall, this mechanism provides full multiplexing and avoids the server-side Head-of-Line blocking issue found in HTTP/1.1 pipelining.

2.3 Performance impact of multiplexing

Now that we have a better idea of what multiplexing entails, let's consider its impact on performance. In this context, *performance* should be understood as in Section 1.1, *i.e.* either end-to-end latency between applications, or the completion time of a bulk transfer.

This section details a set of challenges faced by multiplexing protocols and how these challenges can positively or negatively impact performance.

2.3.1 Sharing costs

Multiplexing effectively means that several streams of data share the same connection. As a result, there are a number of costs that can be shared across the streams, lowering the burden on each individual stream.

Initialization cost sharing The most obvious advantage is that the connection only needs to be initialized once. Any subsequent communication on the connection will not need to wait for the connection to be established, which can be especially costly for TCP and TLS: the three-way handshake followed by the TLS key exchange is costly both in latency and processing time.

RTT measurements Most reliable transport protocols take Round-Trip-Time (RTT) measurements, so that they can adapt their retransmission timeout to the actual latency of the network. However, different connections typically do not share their RTT measurements: for thin streams, this results in very few and inaccurate RTT measurements. Multiplexing several streams allows to obtain more RTT measurements for the same connection, and re-use them for new streams, resulting in faster retransmission.

Fast Retransmit The most efficient retransmission mechanism in TCP, Fast Retransmit, requires a steady flow of data on the connection. Multiplexing several thin streams allows to push more data on the same connection, helping to trigger Fast Retransmit. In effect, with multiplexing, packets from a stream can help another stream to recover quickly from an earlier loss.

Overall, sharing costs is positive, and streams are able to "help" each other when they belong to the same connection. However, this has the drawback of creating dependencies between streams, which can increase latency. Three kind of dependencies are detailed below.

2.3.2 Reactivity and buffer management

When several data flows are multiplexed, they often have different requirements. Some data flows may be bulk transfers with no particular deadlines, while other flows may be highly latency-sensitive. An application will typically prioritize latency-sensitive flows so that they always take precedence.

However, prioritizing sensitive flows doesn't solve the *reactivity problem*: when a latency-sensitive flow suddenly has new data to send, how long does it take before this new data can be actually transmitted? With TCP, unsent data from bulk transfers may still be waiting in the send buffer because of congestion control. When latency-sensitive data is passed from the application to TCP, this new data will be enqueued *after* unsent bulk-transfer data, creating an artificial delay on the sending side.

Since the TCP send buffer is managed by the kernel, it is difficult for applications to effectively control this send buffer. The application can choose the size of this buffer, but it cannot know its optimal size: it should be equal to the Bandwidth-Delay Product (BDP), a value that is not known in advance and that is subject to change over time. If the buffer is smaller than the BDP, bulk transfer performance will suffer; if it is larger, the reactivity problem appears.

An easy way to overcome this issue is to use the `TCP_NOTSENT_LOWAT` socket option on Linux [22]. This allows the kernel to limit the amount of unsent data in its send buffer: it will only mark the socket as writable if the amount of unsent data is below `TCP_NOTSENT_LOWAT`. This helps to solve the reactivity problem while keeping the send buffer large enough to obtain good performance on high-BDP links. This technique has been used with success for HTTP/2 [74, 96].

But even when the reactivity problem is solved at the kernel level, it may still remain at the application level. Software projects such as web servers are very complex and also have internal buffering that can impair reactivity. For instance, the Nginx web server needed extensive internal changes to be able to support HTTP/2 traffic with good reactivity [51].

This problem highlights an important property of multiplexing: the application must be able to choose *which piece of data to send as late as late possible*. This new requirement is specific to multiplexing: in contrast, with TCP, the application does not need to choose which piece of data to send, because there is a single stream of in-order data. As a result, with TCP, the application could just push all its data to a large buffer in the kernel and wait for the work to be done. This explains why having more control on buffer size and receiving more feedback from the transport protocol are relatively recent concerns.

2.3.3 Head-of-line blocking at the transport layer

Head-of-line blocking is a well-known problem with transport protocols that provide a reliable and in-order service, such as TCP. It occurs when several application-level streams are multiplexed over a single connection and there is packet loss, as shown in Figure 2.2. In this case, segment A1 was lost in the network between the sender and the receiver. Because of the in-order guarantee of TCP, all subsequent segments are buffered at the receiver, waiting for A1 to be retransmitted. As a result, the application does not receive any data from the transport layer, while it could already start processing messages of type B: this is what we call head-of-line blocking.

HTTP/2 is a typical case of head-of-line blocking causing performance degradation. Even though HTTP/2 supports multiplexing, it still uses a single TCP connection underneath, and this TCP connection has no visibility over the multiplexed streams. As a result, any lost segment will block all multiplexed streams.

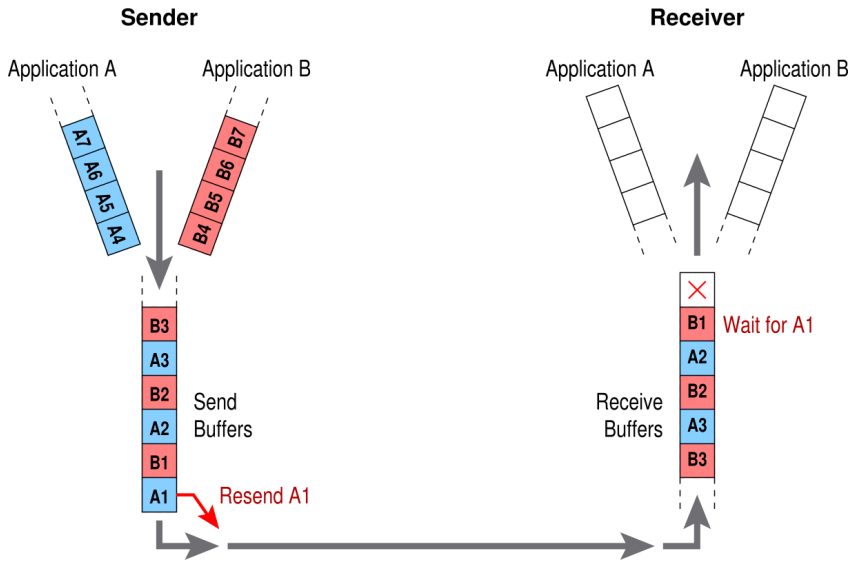

Figure 2.2 Head-of-line blocking situation when two streams are multiplexed on a single TCP connection: any segment loss will block subsequent segments at the receiver, even if they belong to a different stream. Figure from [12].

The solution to avoid head-of-line blocking is to change the semantic of the transport protocol, so that it becomes aware of application data multiplexing. There are two main approaches:

1. SCTP [100] can provide *unordered* delivery of messages to the application, while keeping the service reliable. This means that the application will be delivered messages in any order, and can start processing them as soon as possible [35]. The downside is that additional complexity is pushed to the application: it has to enforce any partial ordering constraint by itself. This method is suitable for applications where messages are completely independent from one another, such as DNS.

2. Both QUIC [66] and SCTP support application data multiplexing through the notion of *stream*, where data belonging to different streams can be delivered in any order to the application. However, within a stream, data is delivered in-order.

2.3.4 Message bundling

Message bundling refers to the practice of aggregating several pieces of data in a single packet. This is typically done by the transport protocol implementation on the sender side.

For stream-oriented traffic, message bundling can be applied whenever the application is giving data to the transport protocol in small chunks (i.e. `write` with a size that is smaller than a single segment). The transport protocol can either transmit each data chunk immediately, or it can wait for a small amount of time to try to bundle several `write` into a single segment. This is used in Nagle's algorithm [76] and can be controlled by the application using TCP's "corking" feature. A similar feature is also supported by SCTP implementations. This kind of message bundle improves transfer efficiency by transmitting larger packets, but it introduces an additional delay before data is actually transmitted.

When a transport protocol is aware of multiplexed data flows, such as SCTP or QUIC, an additional decision needs to be made. If two small messages belonging to different data flows are available for transmission, should they be bundled in the same packet?

The trade-off is slightly different in this case, because no additional delay is introduced on the sender side. However, if the packet containing the two messages is lost, this creates Head-of-Line blocking for both data flows. In effect, this creates a temporary coupling between the two flows. Thus, it is often beneficial to disable bundling for messages that belong to different data flows. SCTP-based simulations showed that it decreases the average latency when running many streams with small amount of data, a typical usage pattern for SS7 running on top of SCTP [95].

2.4 Latency impact of multi-homing

When using multipath networks, there are a few more considerations to have about latency. The main issue arises when the available paths have very different latency characteristics: if the transport protocol is not careful enough, the latency perceived by the application may well be the latency of the worst path! This section starts with effects that negatively impact the end-to-end latency experienced by applications, and then reviews multipath scheduling algorithms as an opportunity to take advantage of path diversity to provide latency.

2.4.1 Multi-homing challenges that impair latency

Head-of-line blocking

It is possible to suffer from head-of-line blocking even without packet loss, simply because of the different RTT on each path. Packets sent on different paths will take a different amount of time to arrive at the receiver, leading to a large amount of out-of-order packets. Thus, to ensure the in-order property of a data flow, data sent on the path with lowest RTT may be blocked at the receiver, waiting for data sent on higher-RTT paths. In the end, the latency experienced by the application can be worse than using a single path.

Multipath scheduler such as DAPS [92, 65] and BLEST [26] try to compensate this problem, either by deliberately sending packets out-of-order or by avoiding a path when it has a risk of producing head-of-line blocking.

Receive window blocking

Receive window blocking is a related problem. Because of head-of-line blocking, the receive buffer may contain a lot of blocked data that cannot be delivered to the application. As a result, the receive buffer may become full and block the sender through the flow-control mechanism. It means that the sender will be blocked from sending on the lowest-RTT path because of the in-flight data on the slower paths.

To overcome this problem, a simple solution is to use very large receive buffers. Solutions that avoid head-of-line blocking will also avoid receive window blocking.

Short flows

Practical measurements with Multipath TCP have shown that short flows cannot take advantage of multiple paths [78]. This is because it takes time for MPTCP to setup an additional subflow on a secondary path: if the flow is short, it will already have been fully transferred on the primary path. This increases overall latency if the primary path has higher latency than the secondary one.

2.4.2 Opportunities offered by multipath scheduling

Of course, excessive latency may also come from the network itself. A well-studied cause is bufferbloat [42], which happens when routers on the path have oversized buffers that can accumulate up to several seconds of queuing delay.

When latency comes from the network, multi-homing can offer end hosts a second chance. That is, if a multi-homed host measures high latency on a path and cannot do anything about it, it can switch to a different path in the hope of obtaining a lower latency. This is the main expected benefit of multi-homing.

To take advantage of this opportunity, a *scheduling strategy* is needed. Such a strategy defines how data received from the application is split across the different paths: it has therefore a huge impact on the end-to-end latency. A scheduler can be designed to fulfill various goals: maximizing aggregated throughput, minimizing latency, minimizing reordering at the receiver… Multipath TCP implements several schedulers [81]:

Round-robin This is the simplest scheduler: it simply spreads data over all available paths, provided their congestion window is not yet filled up (ACK pacing). When paths have different RTT, this strategy exhibits poor performance, large amounts of reordering and high latency.

Lowest-RTT-first The default Multipath TCP scheduler works as follows: it always sends data on the path with lowest RTT, unless its congestion window is filled up (in which case it sends data to the second lowest-RTT path, and so on). This method is suitable both for interactive traffic and bulk transfer: an application that sends small amounts of data will always use the lowest-RTT path, thereby minimizing perceived latency, while a bulk transfer will utilize all paths in parallel to aggregate the available capacity.

In practice, when the paths are very dissimilar (for instance Wi-Fi and 3G), this scheduler still exhibits high latency [112]. This is because it tries to use the high-RTT path from time to time, causing severe head-of-line blocking at the receiver.

Other scheduling strategies have been proposed to overcome these issues:

DAPS [92, 65], short for "Delay-Aware Packet Scheduling", tries to ensure that packets arrive in-order at the receiver. It works by computing a schedule that deliberately sends packets out-of-order, based on one-way-delay estimations, to compensate the RTT difference between the paths.

BLEST [26], short for "BLocking ESTimation" scheduler, tries to estimate when selecting a path would cause head-of-line blocking, and adapts its scheduling strategy accordingly.

ECF [68] uses an Earliest Completion First strategy. Whenever the lowest-RTT path has filled its congestion window, it decides whether to wait for the path to become available again, or to transmit immediately on a higher-RTT path, depending on which solution is expected to complete sooner. For this, it takes into account the RTT and congestion window of each path, but also the amount of data in the sending buffer: it only makes sense to use a higher-RTT path if there is a large amount of data waiting to be transmitted. ECF exhibits much better latency than the lowest-RTT-first scheduler when faced with highly dissimilar paths.

STTF [46] is short for "Shortest Transfer Time First" scheduler. For each segment, it estimates its transfer time on each possible path, taking into account path characteristics but also the congestion state. It then selects the path with lowest transfer time.

2.5 Conclusion

While end-to-end latency involves many differences sources, few of them are in control of the end-host. The main end-host factors affecting latency are the relation between the application and the transport protocol, and the way the transport protocol itself works.

Two techniques in particular can have a large impact of latency, positive or negative:

- multiplexing several flows into a single connection
- using several paths to transmit data

2.5.1 Multiplexing

The main challenge with multiplexing is to exploit its beneficial aspects to improve latency, while avoiding the detrimental dependencies between unrelated data flows. The following table sums up various mechanisms related to multiplexing, and whether they can improve or impair latency.

Table 2.1 Multiplexing mechanisms and their impact on latency

Multiplexing mechanism	Impact on latency
Sharing initialization cost	☺
More RTT measurements	☺
Faster retransmission	☺
Message bundling	☹
Oversized send buffer	☹
Head-of-Line blocking	☹

Thus, any multiplexing transport protocol can expect to enjoy improved latency, but needs to be very careful to avoid pitfalls that can have detrimental effects on latency: message bundling, oversized send buffer, and Head-of-Line blocking.

For instance, working to avoid Head-of-Line blocking requires significant complexity in the transport protocol, and protocol designers may not want to introduce such complexity if they can avoid it. The following table summarizes how different protocols are affected by Head-of-Line blocking:

Table 2.2 Protocols affected by head-of-line blocking

	Avoids HoL blocking in application? (Section 2.2.4)	Avoids HoL blocking at transport layer? (Section 2.3.3)	Details
HTTP 1.1 with pipelining	☹	☹	Section 2.2.4
DNS over persistent TCP [114]	😐	☹	Chapter 3
HTTP/2 over TCP [6]	☺	☹	Section 2.2.4
SCTP [100]	☺	☺	Section 2.3.3
QUIC [66]	☺	☺	Section 2.3.3

2.5.2 Multi-homing

The next challenge is to exploit several paths to improve latency. While the diversity of available paths should yield lower latency, it can be difficult to take advantage of. Serious efforts have been made to tackle the RTT difference, with multipath schedulers focused on reducing Head-of-Line blocking or Receive Buffer Blocking. The table below summarizes several aspects of multi-homing and their impact on latency:

Table 2.3 Multi-homing and its impact on latency

Multi-homing aspect	Impact on latency
More diversity	
RTT difference	
HoL blocking	
Short flows	
Combination with multiplexing	

Practical deployments have pointed out that short flows do not benefit much from multi-homing, because the transport protocol does not have time to setup secondary paths [78]. I argue that this is a perfect opportunity to bring multiplexing into the equation: if short flows are part of a larger, long-lived connection, then all paths can be fully exploited.

Overall, both multiplexing and multi-homing are challenging but full of potential to reduce latency, and this is even more the case when combining the two techniques. This tradeoff is the main focus of this work. Chapter 3 explores in details the performance impact of multiplexing DNS queries within a single TCP or TLS connection. Chapter 4 tackles the more general problem of scheduling multiplexed messages unto several paths, effectively combining both techniques to provide the best possible latency even for short flows.

3 Performance of persistent DNS connections

This chapter provides a first look at protocol performance with DNS (the Domain Name System). The goal is to analyse the performance requirements of DNS applications, whether these requirements are fulfilled adequately, and how to improve DNS performance at the transport layer. DNS is quite particular because it needs both low-latency and reliability: this is uncommon and not easy to achieve.

First, Section 3.1 details the particular performance requirements of DNS. In Section 3.2, I then show that the current approach, using UDP to transport DNS messages, does not provide the required level of performance: it can exhibit very large latency when packets are lost. In Section 3.3, I finally explore how other transport protocols such as TCP could provide better client performance thanks to *persistent connections*.

However, performance requirements of DNS applications is only one side of the question: the DNS infrastructure also needs to withstand the load. In Section 3.4, I develop an experimental methodology to perform large-scale measurements against a DNS server. I then use this methodology to determine the performance impact of DNS-over-TCP and DNS-over-TLS (DoT) on recursive resolvers.

Overall, this chapter focuses on two core aspects of the performance of DNS: the latency experienced by clients, and the server-side resources needed to handle client queries such as CPU time or memory usage.

3.1 DNS: performance requirements and transport protocol

The DNS (Domain Name System) is a fundamental protocol of the Internet and it is expected to remain a critical component in the foreseeable future. Indeed, most programs and protocols in the Internet rely heavily on DNS, for instance web browsers or email servers, and need timely answers to function correctly. A web

browser cannot carry out any work before it knows the IP address of the web server: loading a web page can only start after a successful DNS exchange. If this initial DNS exchange is delayed or fails, then the resulting latency cascades down to the page load time and may be noticeable by the user [12].

Thus, a DNS client needs both *reliability* because it really needs to obtain an answer to its queries, and *low latency* because it may be blocked while waiting for the answer.

Having both requirements is quite atypical in the Internet: most applications either need only low-latency (real-time communication such as Voice-over-IP, where applications can cope with packet loss) or only reliability (file or "resource" transfer using protocols such as FTP or HTTP). This dichotomy is reflected in the limited choice of transport protocols, with UDP and TCP being the most widespread: UDP does not provide any built-in reliability, allowing applications to take advantage of the "native" latency of the network; TCP provides strict reliability and in-order delivery, but does not make any guarantee on the resulting application-level latency.

As a result, it may come as a surprise that DNS uses UDP, because it does not provide the required reliability. This choice was likely made for two main reasons:

1. providing low latency was deemed more important than reliability;

2. UDP is lightweight, in that it requires less processing and less memory than TCP: this allows a server to handle a large volume of DNS queries from many different clients.

A less obvious reason relates to the interface provided by the transport protocol: UDP is message-oriented, which fits the needs of DNS, while TCP is stream-oriented. I will show in Section 3.3.1 that it can have an impact on performance and that neither UDP nor TCP may be the right choice for DNS.

My first contribution in Section 3.3 is to show that, although UDP indeed provides better latency than TCP in the general case of a well-working network, it degrades to provide very poor DNS latency when the network exhibits packet loss and TCP becomes desirable in this case. My second contribution in Section 3.4 is to show that the processing cost of DNS-over-TCP is not as high as one would expect given the complexity of TCP. As a result, it is entirely feasible to run a large-scale DNS recursive resolver over TCP or even TLS.

3.2 Message loss dramatically impacts DNS-over-UDP latency

DNS resolution conceptually involves three types of actors: 1) a *stub resolver* which initiates queries on behalf of an application; 2) a *recursive resolver* which uses various sources to build a response for the query, and maintains a cache of responses for efficiency; 3) *authoritative servers* which are the ultimate source of DNS data, where each server is responsible for a subset of the DNS tree. A full example of DNS resolution without cache is shown in Figure 3.1.

Message loss – losing a DNS query or response – can happen for several reasons during resolution: packet loss due to congestion, overloaded recursive resolver, rate-limiting at the recursive resolver to avoid reflection attacks, or network rate-limiting to mitigate DDoS attacks.

UDP has no built-in reliability mechanism such as acknowledgements or retransmission, so it is up to the application layer to detect and recover from these losses. This is difficult for DNS: a stub resolver has no way to distinguish a message loss from a slow recursive resolver.

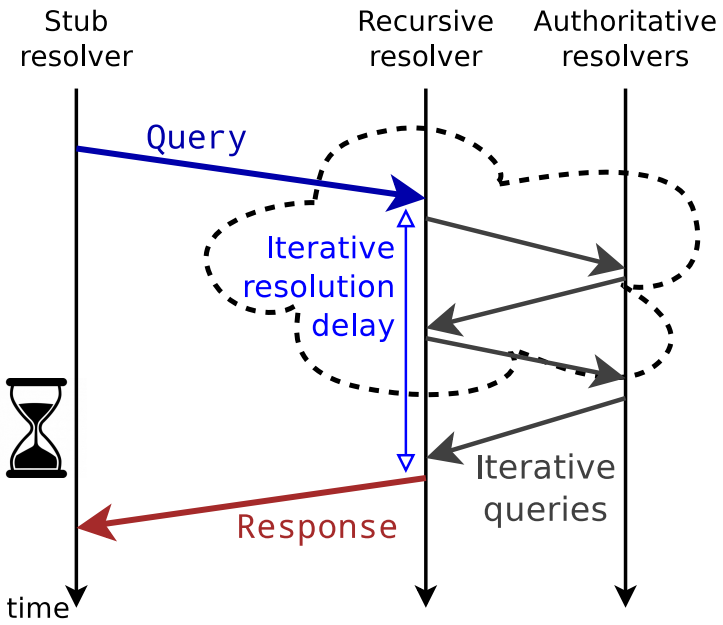

Figure 3.1 Iterative DNS resolution performed by a recursive resolver. When the answer is not in the cache of the recursive resolver, the iterative resolution process can take up to several seconds, during which the stub resolver has to wait.

More precisely, when the stub resolver sends a query, it has no feedback on the iterative resolution process being performed by the recursive resolver. This process

generally completes quickly but can sometimes take up to several seconds, as shown in Figure 3.1. If no response comes back after some time, the stub resolver has to guess whether the iterative resolution process is taking longer than expected, or if a message was lost and a retransmission is required.

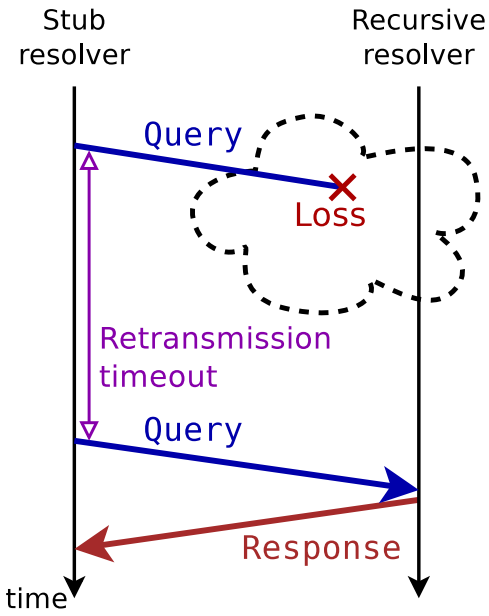

Figure 3.2 The stub resolver must implement a retransmission timer. This timer needs to be larger than the worst-case response time to avoid spurious retransmission. It it thus much larger than typical RTTs.

How should this retransmission timer be chosen? To avoid spurious retransmission, it should be at least as long as the iterative resolution delay, as illustrated in Figure 3.2. The following table lists the retransmission timers used in practice by typical stub resolvers. It also lists their retransmission strategy, along with the total time it takes for the stub resolver to give up retransmission altogether and signal a failure to the calling application.

Table 3.1 Retransmission behaviour of widely used stub resolvers, obtained through experiments. Each stub is configured with two recursive resolvers. The results have been partially confirmed with source code analysis (glibc, bionic) and documentation (Windows).

Stub resolver	First retrans. timeout	Retransmission strategy	Time before failure
Glibc 2.24 (Linux)	5 seconds	Constant interval	40 seconds
Bionic (android 7.1.2)	5 seconds	Constant interval	30 seconds
Windows 10	1 second	Exponential backoff	12 seconds
OS X 10.13.6	1 second	Exponential backoff	30 seconds
IOS 11.4	1 second	Exponential backoff	30 seconds

These timeout values directly impact the end-user experience: on Android, the most widely used operating system on smartphones and tablets, any application needs 5 seconds to recover from a single query loss.

3.3 Improving DNS latency with persistent connections

Using a *persistent DNS connection* means that a connection is opened and reused for several successive DNS transactions. In this setup, the stub resolver can use acknowledgements to measure the RTT towards the recursive resolver, independently from the iterative resolution delay shown in Figure 3.1. It can then adapt its retransmission timer to recover more quickly from a packet loss, based only on the measured RTT. This mechanism is illustrated in Figure 3.3, where it is assumed that the persistent TCP connection has already been opened before.

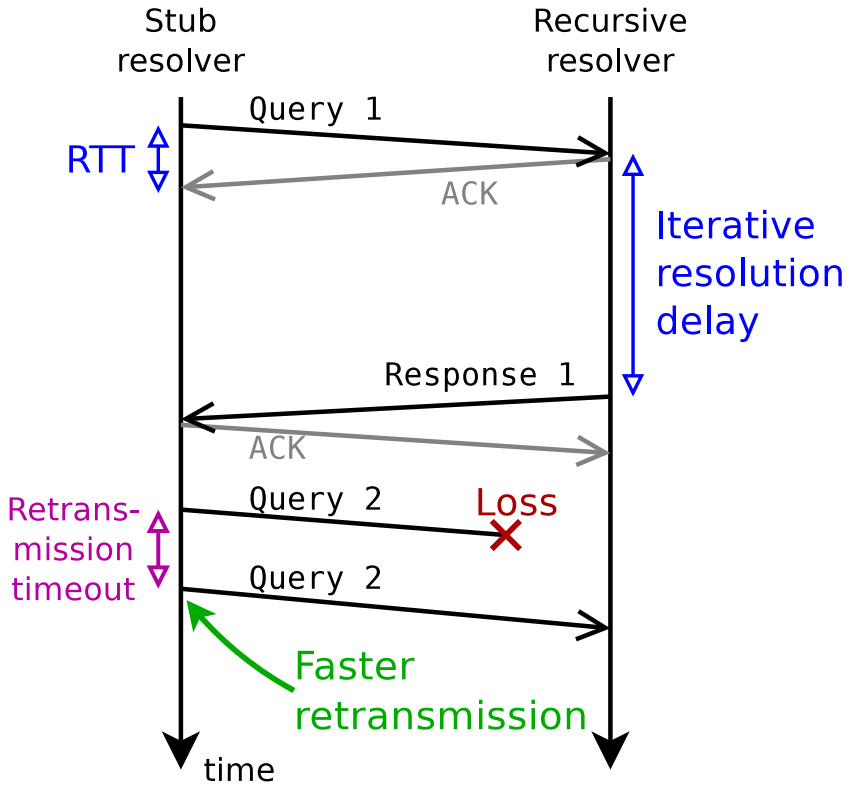

Figure 3.3 Persistent connections improve the response time: the retransmission timer can be adapted to the RTT and does not depend on the worst-case response time anymore.

This approach basically decouples transport-layer and application-layer concerns: the transport protocol simply ensures that messages are delivered reliably and in a

timely manner, without being affected by large delays in higher-level mechanisms such as the iterative resolution process. This decoupling offers several advantages:

1. it avoids wasting resources on repeated queries: since retransmission happens at the transport layer, only a single copy of the query will be delivered to the application even if it was received multiple times due to retransmission;
2. it simplifies applications by moving complexity from the application into the transport protocol.
3. it provides faster retransmission, thanks to the ability to measure the RTT without being affected by delays in the application;

Overall, running DNS on top of a reliable transport protocol such as TCP, coupled with persistent connections, should significantly improve query latency when recovering from packet loss.

3.3.1 Experimental validation

To better understand the impact of packet loss on latency and verify the effectiveness of persistent DNS connections, I designed experiments in the controlled testbed illustrated in Figure 3.4.

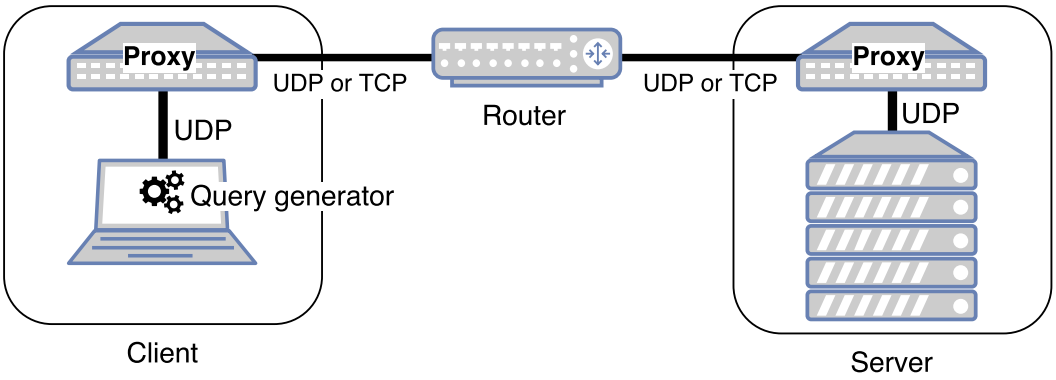

Figure 3.4 Setup for the testbed experiment comparing the latency of DNS-over-UDP and DNS-over-TCP. The router (middle) can apply packet loss and delay to all packets flowing through it.

A client sends DNS queries using either UDP or a persistent TCP connection. A router is inserted between the client and the server to emulate various network conditions such as high RTT or packet loss. The testbed uses three APU2 boards from PC Engines cabled directly to each other, without any intermediate switch or network equipment that could affect the results of the experiment. All boards run Debian Stretch with version 4.9 of the Linux kernel.

The client[1] and proxy[2] are custom software, while the server runs `unbound` with a pre-filled cache. Packet loss and delay are emulated on the router using `netem`. Loss is induced in both directions: a loss of X% means that the router independently applies X% of loss from the client to the server, as well as X% of loss from the server to the client. For the following experiments, the testbed is configured with 2% of packet loss in each direction and 20 ms of RTT.

Queries are generated by the client with a fixed inter-query interval. In the figures below, values of either 50 ms and 100 ms are used, depending on the experiment. The client is configured to retransmit after a fixed timeout of 3 seconds, which is the average retransmission timeout of stub resolvers in Table 3.1. In each experiment, 1000 queries are sent. For a given set of parameters, the experiment is repeated 21 times, and data points from all experiments are aggregated in a single CDF. As a result, each CDF contains 21000 queries.[3]

These experiments allow comparing the latency of UDP and TCP under packet loss. Figure 3.5 shows the complementary CDF (CCDF, also known as tail distribution) of query latency for a 100 ms inter-query interval. Figure 3.6 shows the same data with a logarithmic scale to better see the tail of the distribution.

UDP has a very characteristic step-like behavior: 96% of UDP queries are immediately successful and experience a latency close to the RTT (22 ms), while most of the remaining UDP queries need a single retransmission and end up with a latency as high as 3 seconds. A handful of queries need two retransmission and experience a latency of 6 seconds.

Table 3.2 gives the theoretical probability distribution of the number of retransmissions needed to achieve a successful exchange, based on a 2% failure probability for queries and 2% failure probability for responses. The experimental data in Figure 3.6 gives values that are very close to these theoretical probabilities.

Table 3.2 Theoretical probability of success depending on the number of retransmission.

Success after...	Latency	Probability
No retransmission	20 ms	96.040%
One retransmission	3020 ms	3.803%
Two retransmissions	6020 ms	0.151%
Three retransmissions	9020 ms	0.006%

[1] `https://github.com/SinBirb/dnsquerier`
[2] `https://github.com/SinBirb/dnstransformer`
[3] Performing 21 repetitions might seem strange. The initial goal was to additionally vary TCP settings (see Section 3.3.2) with 7 different values, and repeat each experiment 3 times. However, the TCP settings were not applied due to a mistake in the experiment script. As a result, for each set of parameters, data points are available from $7 * 3 = 21$ experiments.

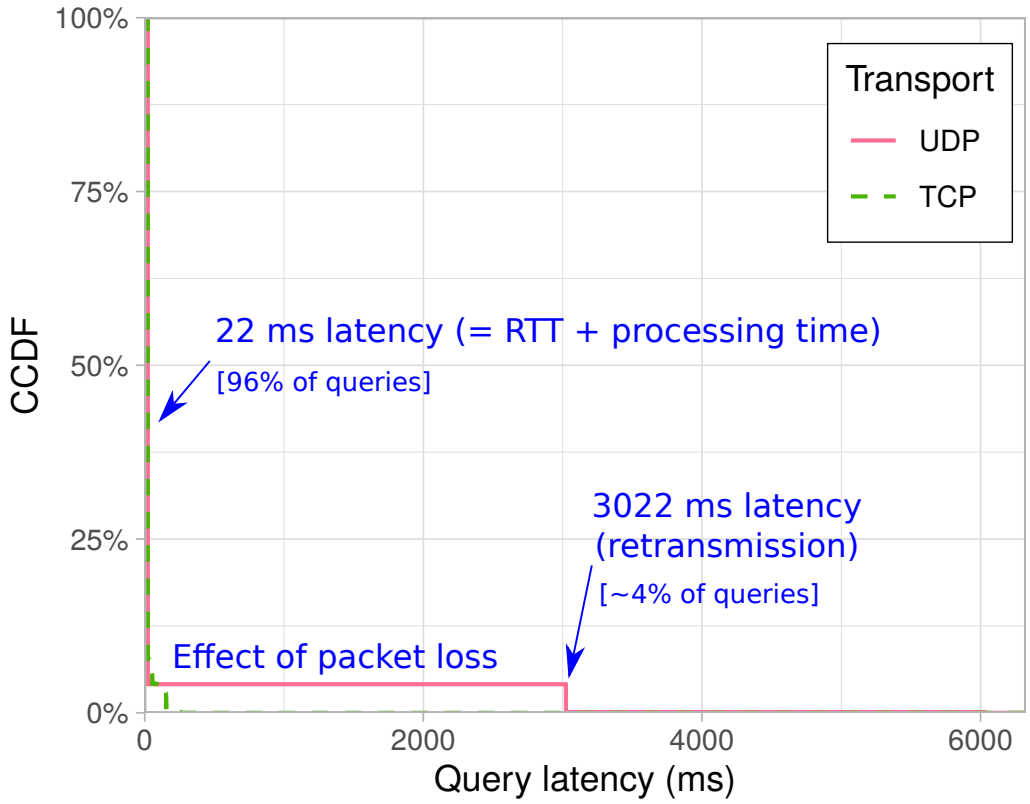

Figure 3.5 Query latency for UDP and TCP shown as a Complementary CDF. The testbed is configured with a 20 ms RTT and 2% of packet loss in each direction. The inter-query interval is 100 ms. With UDP, a retransmission is needed for roughly 4% of queries: this happens when either the query or the response is lost.

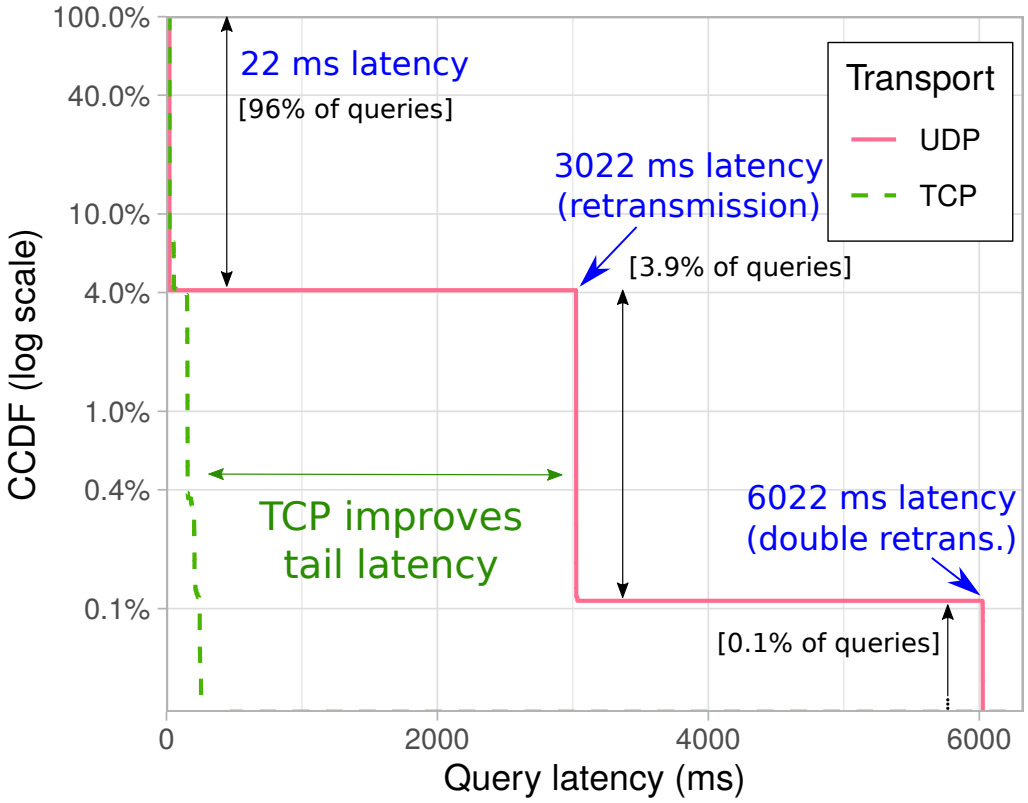

Figure 3.6 Query latency for UDP and TCP: same data as in Figure 3.5 but with a logarithmic scale. A few queries actually need two retransmissions. Using TCP significantly reduces the retransmission latency in case of loss.

Figure 3.6 shows that TCP exhibits a much better tail latency compared to UDP: the 99th percentile is reduced from 3022 ms to 109 ms, and the 99.9th percentile is reduced from 6022 ms to just 197 ms. This is because the TCP retransmission scheme adapts itself based on the RTT, while UDP retransmissions are blind and use a fixed timeout.

Next, Figure 3.7 takes a close-up look at small latencies to better understand the retransmission behaviour of TCP. It shows the same data as Figure 3.5 but focuses on small latencies and only shows part of the CCDF. In addition, it introduces "TCP bursts", where 3 queries are sent back-to-back on the TCP connection. The inter-query interval is still 100 ms. This new curve (dotted blue) uses 3 times as much queries compared to the simple TCP case: in each experiment, 1000 bursts of 3 queries are sent.

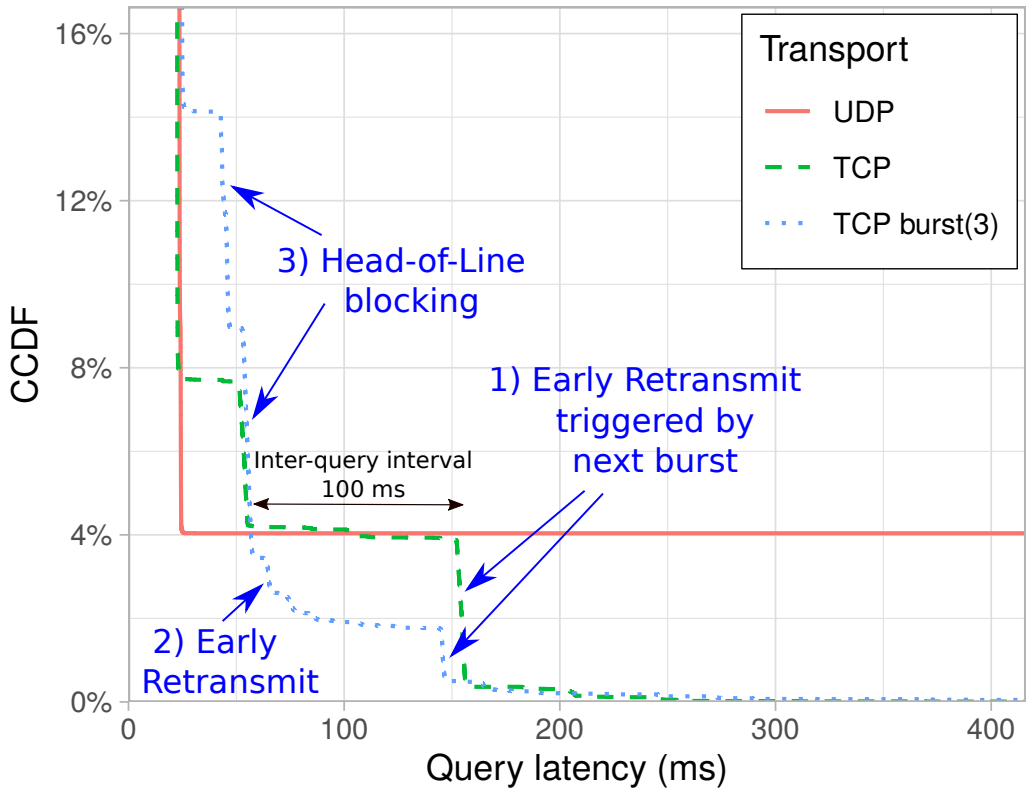

Figure 3.7 Query latency for UDP and TCP: zoom on small latencies, and introduction of query bursts for TCP (groups of 3 queries sent back-to-back) The testbed is configured with a 20 ms RTT and 2% of packet loss in each direction.

There are several interesting behaviours to notice in this figure:

1. **Early Retransmit triggered by next burst:** for TCP, approximately 4% of queries exhibit a latency close to 155 ms. This corresponds to queries or

responses that were lost and were able to recover thanks to the next query. 100 ms after sending the lost message, the client sends a new query. When this new query arrives at the server, it triggers a duplicate ACK. When this duplicate ACK reaches the client after one RTT (20-25 ms), it triggers the *Early Retransmit* [3] mechanism that is enabled by default on Linux 4.9. That is, whenever there is only a small amount of data in flight (which is the case here), the sender is allowed to use Fast Retransmit even if the usual threshold of 3 duplicate ACKs is not met. As a result, the client is able to retransmit the lost packet almost immediately, and it takes a further RTT to complete the exchange. The theoretical overall latency would be 140 ms, computed as: 100 ms (inter-query interval) + 20 ms (loss detection, one RTT) + 20 ms (retransmission, one RTT). The actual latency is a bit higher at 155 ms: the difference can be explained by processing times and by the fact that Linux uses Delayed Early Retransmit, according to which it delays Early Retransmit by 1/4 of the measured RTT.

TCP bursts exhibit a similar pattern but only for about 1.5% of queries. This is because Early Retransmit is triggered by the next burst only if the third and last query of a burst is lost. That is, this mechanism only applies to one-third of the queries. In addition, the latency is slightly lower at 145 ms: this is because the next burst triggers 3 duplicate ACKs, which avoids the 1/4 RTT delay of Delayed Early Retransmit.

2. **Early Retransmit for TCP bursts:** this second mechanism only applies to the first and second query of a burst. Since they are immediately followed by a third query, a duplicate ACK is immediately triggered and Delayed Early Retransmit can be applied earlier, without having to wait 100 ms for the next burst of queries. This yields a query latency around 65-75 ms: 20 ms for loss detection, 20 ms for retransmission, and 25-35 ms to account for processing delays and Delayed Early Retransmit.

3. **Head-of-Line blocking:** this applies to queries that were not lost, but were blocked by a previous lost query. This blocking happens because TCP delivers data in-order. For TCP, this accounts for 4% of queries: one query is blocked for each lost query. For TCP bursts, Head-of-Line blocking is more severe and the mechanism applies to around 10% of queries, because several queries in a burst can be blocked. The latency of blocked queries is 45-55 ms: it can be interpreted as 20 ms of loss detection, 20 ms of retransmission, along with processing time and the delay of Delayed Early Retransmit. It is interesting to see that there is exactly 100 ms between the two parts of the CCDF cor-

responding to “Early Retransmit triggered by next burst” and “Head-of-Line blocking, especially for TCP in dashed green. This is because two successive queries – sent with a 100 ms interval – are delivered at the same time if the first one is lost.

To validate these findings, the same series of experiments was run with 50 ms of inter-query interval instead of 100 ms. The result is shown in Figure 3.8 and supports the analysis done with 100 ms.

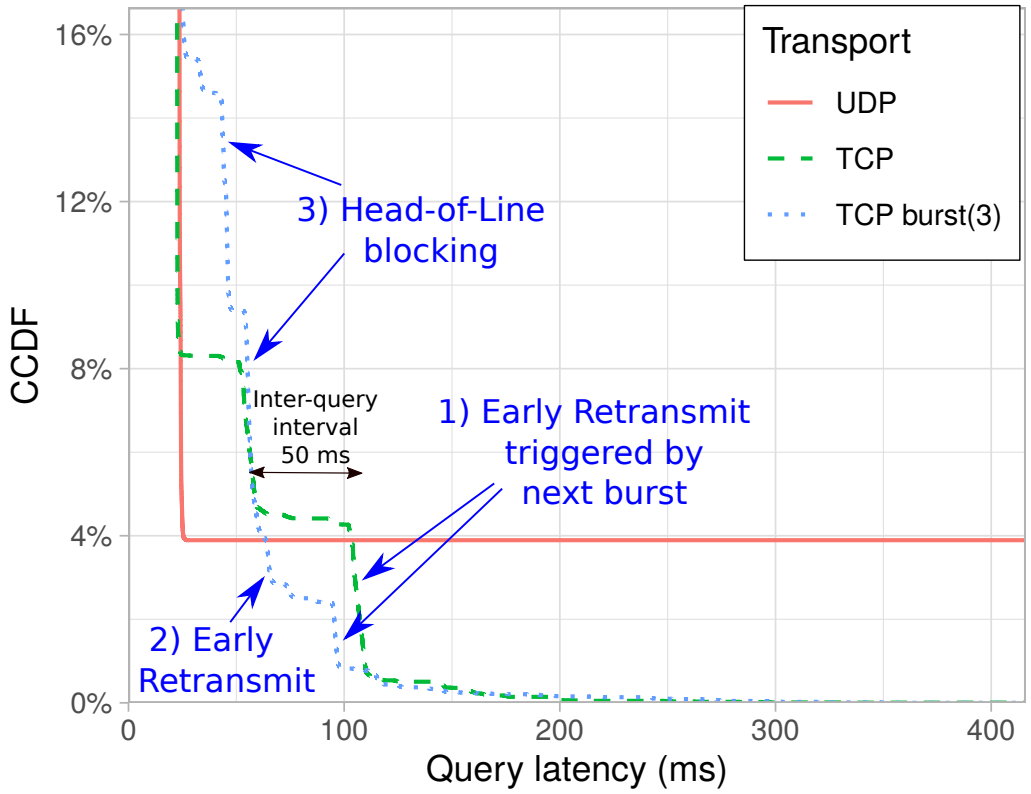

Figure 3.8 Latency comparison of UDP, TCP and TCP bursts, with the same methodology and parameters as Figure 3.7 except for the inter-query interval that is set at 50 ms.

Overall, using persistent TCP connections significantly improves the worst-case latency compared to UDP when DNS messages are lost. Sending frequent queries or bursts of queries helps TCP to recover even more quickly, supporting the use-case of aggregating queries from different devices or applications in a single persistent connection.

In addition, Early Retransmit [3] has a very positive effect on latency because it allows any new query to help recover the loss of previous queries. Without Early Retransmit, TCP would need either 3 queries to trigger 3 duplicate ACKs and thus

Fast Retransmit, or it would need to wait until triggering the RTO (Retransmission Timeout) whose typical timeout values range between 200 ms and 1 second [84].

However, the use of TCP introduces Head-of-Line blocking, which needlessly degrades latency for some queries. This problem could be solved by using DNS-over-QUIC [45] or DNS-over-SCTP, because both protocols provide independent "streams": sending each query in a different stream would eliminate Head-of-Line blocking between queries. This is a motivation for Chapter 4 in which I provide a more general discussion on stream-aware scheduling in the context of QUIC and SCTP.

3.3.2 Related work

Thin streams Running DNS over a persistent TCP connection is an instance of a "thin-stream" application [85]. That is, the connection stays open for a potentially long time, but very little data is sent on the connection. In addition, traffic is typically very sporadic, with long periods of silence intertwined with short bursts of data.

TCP is known to behave badly in this use-case, especially regarding latency [85]. The most serious issue is "tail loss" [89]: if one of the last TCP segments in a communication is lost, then there are not enough duplicate ACKs to trigger a fast retransmit, so the sender has to wait for a full RTO before retransmitting. This can take several seconds and produce head-of-line blocking. Tail losses are normally only a problem at the very end of a TCP-based transfer, hence the name *tail* loss. But in a thin stream, any segment preceding an idle period can suffer from a tail loss.

There have been several proposals to improve latency for thin streams [89], including Early Retransmit [3] and Tail Loss Probes [21]. Early Retransmit ensures that only the very last packet can be negatively affected by a tail loss because it lowers the threshold to trigger Fast Retransmit. Tail Loss Probes try to trigger duplicate ACKs by resending the last packet with a lower timeout than the RTO. These two mechanisms are complementary.

The Linux kernel implements many of these algorithms. Table 3.3 below lists available settings related to TCP thin streams as well as their default value.

For the experiments in Section 3.3.1, all these Linux settings were kept to their default values and this yielded good DNS-over-TCP performance. Still, experimenting with various combinations of these settings would be an interesting future development.

Table 3.3 Kernel sysctl settings related to TCP thin streams (Linux 4.9)

Setting	Default value
tcp_thin_linear_timeouts	0
tcp_thin_dupack	0
tcp_early_retrans	3
tcp_recovery	0x1
tcp_slow_start_after_idle	1
tcp_low_latency	0

Comparing performance of DNS transport protocols Early work [114] highlighted the advantage of using persistent connections: it greatly improves latency by amortising the cost of establishing a connection.

Recent work [43, 44] compared the latency of DNS-over-UDP, DNS-over-TLS (DoT) and DNS-over-HTTPS (DoH) against a set of open resolvers, using emulated loss for some measurements. They instrument a web browser to obtain DNS latency data when loading actual web pages, which gives a real-world view of DNS performance. However, these measurements have drawbacks: the measured latency includes several factors that are hard to pull apart, such as network latency, query retransmission, communication with authoritative DNS servers if the answer is not in the cache, variable connectivity with these authoritative servers over time, client connection persistence. Another factor is that a given open resolver provider might use different servers for each protocol, with different load, cache policy and cache hit rate. As such, the resulting latency data is very variable and hard to interpret. By contrast, the latency measurements I performed were done in a controlled environment: it is less realistic, but it provides the ability to observe specific behaviors in isolation and explain them.

In the end, the authors of [44] reach a similar conclusion: under moderate amount of packet loss, DoT and DoH generally exhibit larger latency, but they still deliver equivalent or even slightly better Page Load Time compared to DNS-over-UDP. This is likely caused by the large timeout in DNS-over-UDP implementations that I identified in Section 3.2. Because of this timeout, an unlucky web page resolved with DNS-over-UDP may be significantly slowed down if one of its important DNS queries is lost. However, the performance of DoT and DoH becomes poor under very adverse network conditions (high latency, low throughput, high loss), likely because TCP has trouble recovering from heavy losses with a thin-stream kind of traffic.

Another work [10] again highlights the importance of persistent connections to amortise the high cost of TLS connection establishment. The authors also study the impact of Head-of-Line blocking, but they only consider blocking at the level of the

recursive resolver, instead of the more general Head-of-Line blocking problem caused by packet loss (see Section 2.5). This makes a difference for DNS-over-HTTP/2, which is not affected by blocking in the recursive resolver, but which would still suffer from Head-of-Line blocking caused by packet loss, like any other TCP-based protocol. Lastly, they use a methodology similar to [44] and obtain consistent results: the latency of DoT and DoH is generally higher than with UDP, but it does not significantly affect web Page Load Time.

3.3.3 Going beyond latency

These controlled experiments illustrate that persistent DNS connections can significantly improve the worst-case latency of DNS queries when messages are lost, although head-of-line blocking can negatively affect the overall latency; this last effect could be mitigated using QUIC or SCTP.

But to be widely deployed, persistent DNS connections need to be feasible on a large scale for all actors of the DNS ecosystem. Thus, the next section looks at the performance impact of persistent DNS connections on recursive resolvers.

3.4 Evaluation of recursive DNS resolver performance

My second contribution is to assess the server-side cost of using DNS-over-TCP or DNS-over-TLS. Both TCP and TLS introduce protocol overhead, such as managing session state or timers for retransmission. Perhaps more importantly, the cryptographic operations performed by TLS can significantly increase the CPU cost of handling queries.

To determine the server-side cost of TCP and TLS, I experiment with recursive resolvers under high load and determine how many clients and queries they can handle at a maximum, all while maintaining acceptable latency for clients. I also compare how this "peak performance" varies when clients use UDP, persistent TCP connections, or persistent TLS connections.

3.4.1 The need for persistent connections

We have already seen in Section 3.3 that persistent connections can be helpful to improve the latency experienced by DNS clients. For recursive DNS resolvers, the main performance metric is the cost of handling queries. In particular, when using TLS, cryptography plays a role in performance. TLS uses two kind of cryptographic primitives:

1. **symmetric cryptography** used to encrypt and decrypt user data with a session key. This is typically very fast and efficient, either because it is implemented in hardware (AES) or because efficient software implementation are possible (ChaCha20)

2. **asymmetric cryptography** used during key exchange to derive a session key. It is much more costly CPU-wise, but is only needed at the start of a TLS connection.

This gives additional weight to the use of *persistent DNS connections*: the high CPU cost of asymmetric cryptography will only need to be paid once for several queries. However, it comes at the cost of more memory usage: the server will have to maintain state for a large number of persistent connections, even if they remain idle. In practice, servers can close connections when the client has been idle for too long.

In the rest of this section, I assume that persistent connections are used.

3.4.2 Deployment model: large-scale persistent DNS connections

The guiding principle for the experiments is: if we were to switch all end hosts in the Internet to use persistent DNS connections, could the current DNS infrastructure withstand the additional load?

The specific deployment model I consider is shown in Figure 3.9. Home routers run a simple DNS forwarder and maintain a persistent TCP or TLS connection towards a single recursive resolver. Devices in the home network send their DNS queries to the home router, which forwards them over the persistent connection to the recursive resolver.

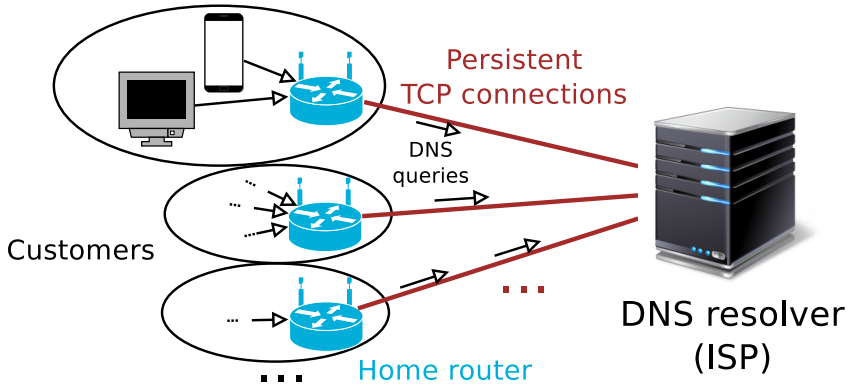

Figure 3.9 The deployment model for persistent DNS connections.

This is a simplified model, since a real ISP would typically use several DNS resolvers – possibly organized in a multi-tier topology – and each customer would be assigned to predetermined resolvers in the pool so that the load can be spread on a large number of machines. This model is thus a "worst-case" situation where the number of DNS resolvers is reduced to just one.

I focus exclusively on the "frontend" communication between stub resolvers and recursive resolvers: I ensure a 100% cache hit ratio so that the results don't measure anything related to iterative resolution. I also make sure that all clients open their persistent connections before starting to send queries, so that I obtain results for an ideal steady state.

3.4.3 Experimental setup and methodology

To implement the model shown in Figure 3.9, I setup a large-scale testbed setup involving tens of thousands of stub resolvers connected to a recursive resolver. Stub

resolvers run an custom instrumented DNS client[4], while the recursive resolver either runs `unbound`, `bind9` or Knot Resolver.

I use a large number of physical servers from the Grid'5000 [5] research platform. A single physical server runs the recursive resolver software, while each other server spawns several Virtual Machines (VM) that run the custom DNS client. Each server has 20 CPU cores (2x Xeon E5-2630 v4) and 128 GB of RAM, and all servers are interconnected through a local 10 Gbit/s Ethernet network. For the largest experiment, I used as many as 18 physical machines to host 216 VMs.

Each VM opens several persistent connections to the recursive resolver to send queries: for instance, it is possible to simulate 10,000 TCP customers by running only 100 VMs, with each VM sending queries on 100 TCP connections in parallel. The custom DNS client generates queries according to a Poisson process, and then sends each query on one of the persistent connections chosen uniformly at random. This naturally results in a Poisson process on each persistent connection. I discuss the relevance of this query generation model as well as other models in Section 3.4.6.

Reproducibility To manage the experiments and make them reproducible, I developed an automated deployment script thanks to the Execo library [47]. Given a set of parameters as input, the script reserves the appropriate server resources on Grid'5000, deploys the necessary software including virtual machines, starts the experiment in a synchronised manner on the various machines, and then collects traces for later analysis. Overall, this script encapsulates all the complexity of setting up the experiment and allows to easily repeat experiments.

The source code for this tool is available at `https://github.com/jonglezb/dns-server-experiment`

3.4.4 Methodology: performance metrics

The first goal is to estimate the peak performance of the DNS resolver, *i.e.* the maximum rate of queries it can process. To this end, I start a large number of DNS clients that send queries according to a Poisson process, and then slowly increases their query rate. When the answer rate from the resolver drops permanently below the query rate, it means that the resolver has reached its capacity. I define the *peak performance* as the highest rate for which the answer rate matches the query rate. Figure 3.10 shows the query rate and answer rate evolving over time during a typical experiment using `bind9`. Using `unbound` and/or TLS clients yields similar

[4] `https://github.com/jonglezb/tcpscaler`

results. The horizontal blue line in the figure shows the peak performance (in Q/s *i.e.* queries per second) the method computed for this experiment.

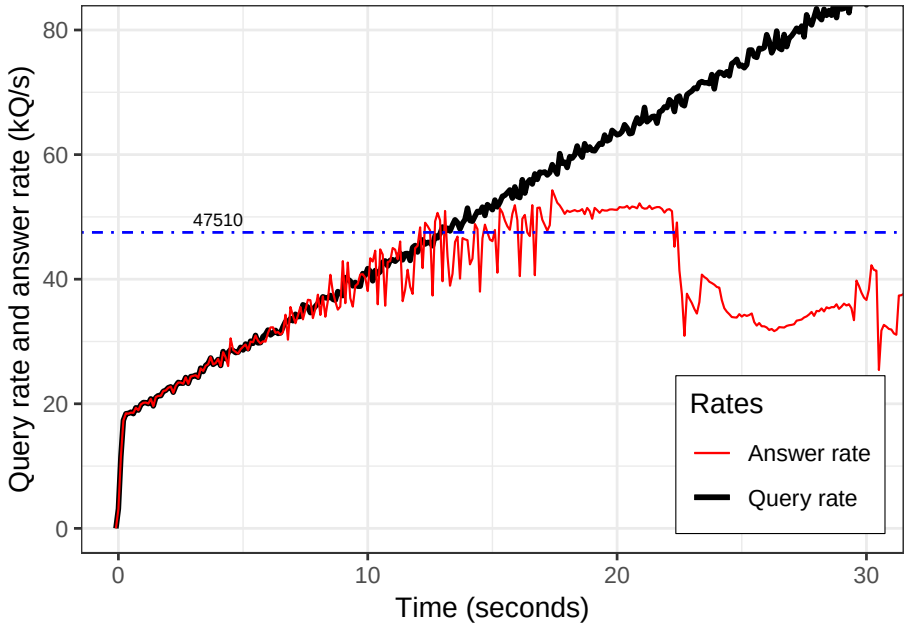

Figure 3.10 Total query and answer rates seen by clients during an experiment. The horizontal line indicates the peak performance computed with the method: 47.5 kQ/s. Parameters: `bind` 9.13.3 with 1 thread, 24 VMs, 125 TCP connections / VM.

The second main performance metric is latency, which I collect through client-side logging: all clients measure the end-to-end latency of each query they have sent. Figure 3.11 shows aggregated client latency as a function of the query load for the same experiment as Figure 3.10. As expected from queuing theory, latency roughly follows an exponential evolution as the query load approaches the saturation point. Still, it is safe to use the resolver at 80% of its peak performance, yielding a 95th latency percentile below 20 ms.

3.4.5 Results

Using the methodology described above, I ran a campaign of around 600 experiments. The obvious parameters I varied are the recursive resolver software, the number of clients, and the transport protocol (UDP, TCP, TLS). Other parameters include the slope with which clients increase their query rate, or the number of threads configured on the recursive resolver.

Figure 3.12 shows the peak performance of `unbound` running on a single CPU core, as a function of the number of client connections. With few clients, performance of

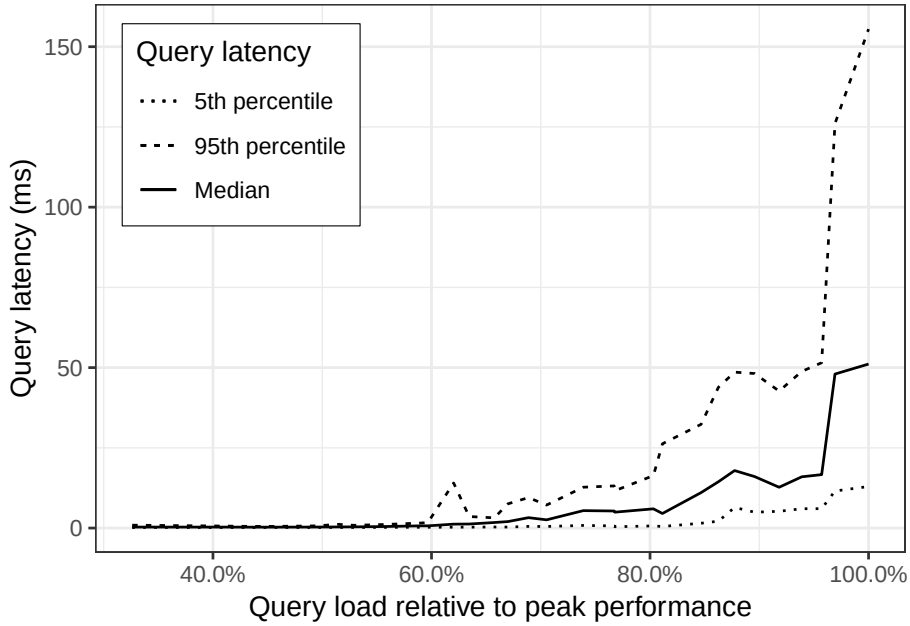

Figure 3.11 Query latency as a function of the query load, up to the peak performance rate that was measured (47.5 kQ/s in this case). This is from the same experiment as Figure 3.10.

DNS-over-TCP is close to that of DNS-over-UDP, with only a 30% slowdown. When the number of clients increases, performance of DNS-over-TCP drops, stabilizing around a slowdown of roughly 75%. For DNS-over-TLS, the performance profile is similar to TCP, but with a 30% to 45% performance hit. It means that with a large number of TLS clients, the performance slowdown compared to UDP is around 83%.

Figure 3.13 shows a comparison of the peak performance of `unbound` and `bind9`. Only the plots for TCP clients are shown, since `bind9` does not natively support TLS. The performance profiles are similar, although `bind9` is generally slower. Interestingly, running `unbound` with DNS-over-TLS or `bind9` with DNS-over-TCP yields roughly the same performance. This suggests that with modern hardware, encryption is not a serious bottleneck.

For both TCP and TLS, performance drops significantly when the number of clients increases. Since it similarly affects `bind9` and `unbound`, I believe this is caused by the high number of concurrent TCP connections that needs to be managed by the kernel. First, the kernel data structures associated to TCP connections may no longer fit in the CPU cache, slowing down any access to them. Second, when there are more clients, the query rate per client is lower for the same overall query rate. It means that there are less opportunities for TCP to process several segments from the

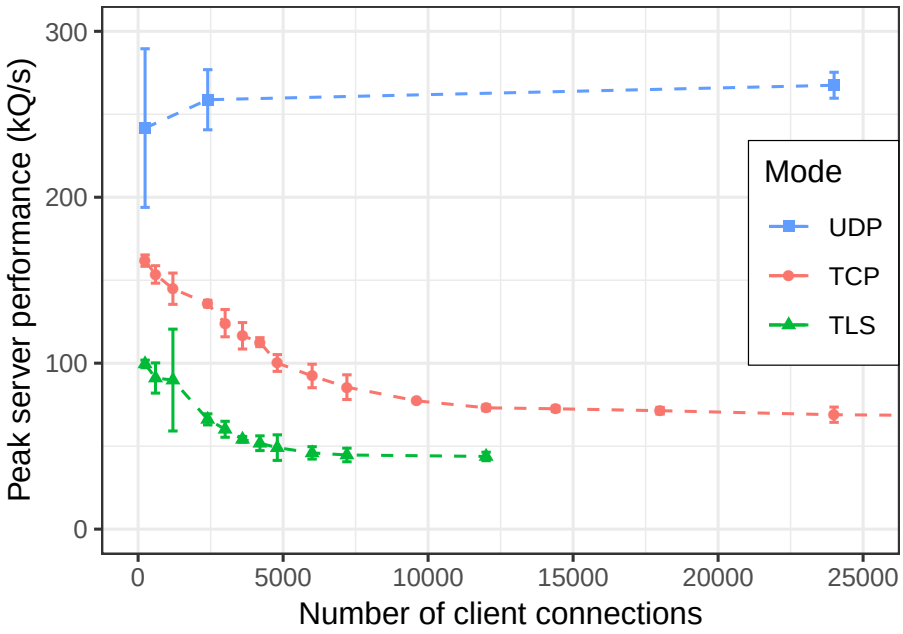

Figure 3.12 Performance of `unbound` when the number of clients increases. Each point shows the average peak performance for the given number of clients over several experiments, with 95% confidence intervals.

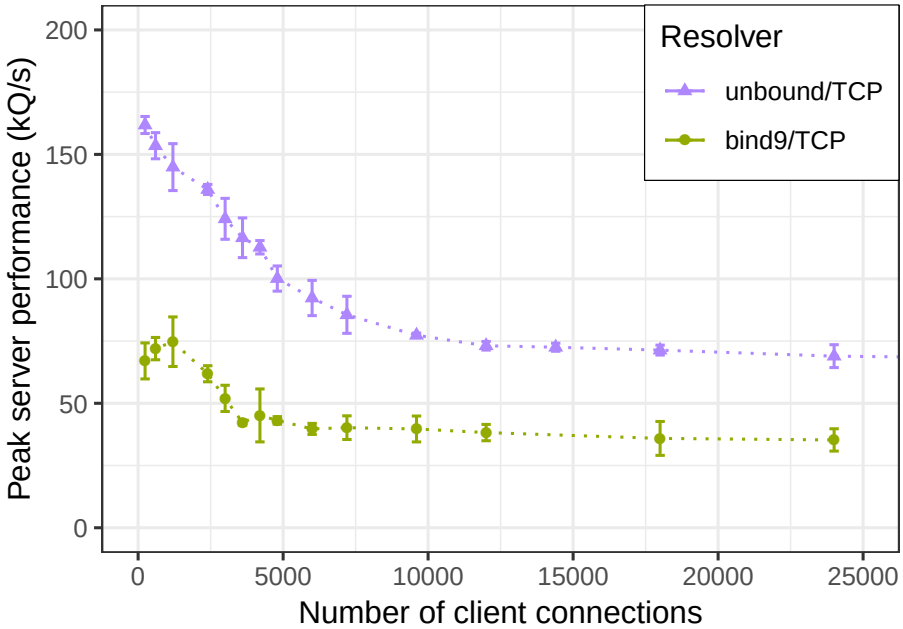

Figure 3.13 Performance comparison of `unbound` and `bind9` with TCP clients. Each point shows the average peak performance for the given number of clients over several experiments, with 95% confidence intervals. The plot for `unbound/TCP` is the same as `TCP` in Figure 3.12.

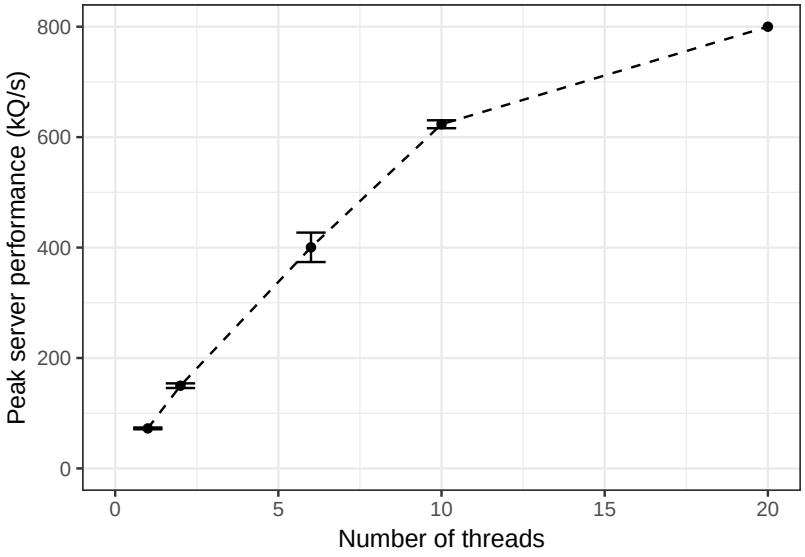

Figure 3.14 Performance speedup when using several threads with `unbound` on a server with two 10-cores CPUs. For each experiment, there are 48 VMs running clients, with 300 TCP connections per VM.

same TCP socket in a single pass. These two effects are likely to combine to explain the drop in performance.

Next, I look at multi-core performance. Figure 3.14 shows that performance scales linearly with the number of cores up to 800 kQ/s. At such a high query rate, the bottleneck seems to move to the hardware rather than the software: even with UDP, the server could not go much higher than 800 kQ/s, hinting at a limit of the physical network interface (NIC) or the NIC – host communication. These limits impact UDP, TCP and TLS similarly. Thus, using DNS-over-TLS on a single CPU core is a worst case situation.

The most extreme experiment (not shown in the graphs) involved 6.5 million TCP clients, all connected to a single `unbound` instance. We obtained an honorable peak performance of 50k queries per second on a single CPU core, which shows that DNS-over-TCP can scale to extreme number of clients. However, the memory consumption is high enough to make this scenario undesirable in practical setups: as much as 51.4 GB of RAM was used on the resolver machine, including both kernel and userspace usage. Since `unbound` allocates 4 KB of buffer space for each TCP connection, I estimate that a further 3.7 KB of memory is needed for each TCP connection: in the kernel, in `libevent`, and in sections of `unbound` that are unrelated to query buffers. TLS memory usage would be even higher.

Overall, the results show that with up to 1000 clients, the CPU processing cost is 2.5 times higher with DoT than with UDP. Interestingly, for a larger number of clients, performance drops and DoT becomes 5 times more expensive than DNS-over-UDP.

3.4.6 Limitations of the methodology

Query generation model

To generate queries, I use a Poisson process. This has the advantage of simplifying the client side of the experiment, as described in Section 3.4.3.

More realistic query distributions could be used for each client, such as Pareto with a point mass [58], but they would need to be fitted on real data. Unfortunately, I could not obtain any DNS client traces from a production network, as this type of data is very privacy-sensitive. In any case, with a large number of independent clients that send a small amount of queries, the overall traffic can be approximated with a Poisson process.

Differences between DoT and DoH

I only experiment with TCP and TLS, as opposed to the more recent trend of using DNS-over-HTTPS (DoH).

On the protocol side, DoH and DoT are very similar and only differ in the presentation format. Thus, I expect them to exhibit similar server-side performance in terms of CPU cost: in this specific setting, most of the CPU cost is related to TCP (state and timer management) and TLS (cryptography).

However, there may still be significant differences in performance due to implementation differences.

Churn and cost of new TLS connections

The performance evaluation shows that TLS is not that much slower than TCP (Figure 3.12). This is because I chose to focus on the steady-state by opening all persistent connections before starting to send queries: this hides the cost of establishing a new TLS session. In a real DNS setup, clients will come and go, so

the recursive resolver will need to spend CPU time on cryptographic key exchange, potentially slowing down resolver performance.

Initial performance measurements with `unbound`, configured with a 2048 bits RSA key and default ciphers, shows that it can accept roughly 900 new TLS sessions per second on a single CPU core. Furthermore, when spreading the incoming connections on a few CPU cores, the number of new TLS sessions per second scales linearly. This is consistent with common performance figures found in the HTTPS industry: Nginx reports 620 RSA signatures per second in a virtual machine on unspecified hardware from 2014 [77], while I use powerful hardware from 2016 (Xeon E5-2630 v4 at 3.10 GHz turbo frequency) and more recent software.

Overall, a CPU core can either handle 50,000 to 100,000 queries per second over TLS (Figure 3.12), or accept around 900 new TLS sessions per second. To obtain a balanced CPU usage, clients would need to send between 55 and 110 queries per session on average, which seems rather high. Thus, I expect in practice that most of the CPU resources will be spent on TLS connection establishment.

TLS session resumption via session tickets, as specified for TLS [90] and QUIC [104], is a promising solution to this performance bottleneck. It would allow sporadic clients to terminate their TLS connections after a short idle period, freeing up memory resources on the resolver, while subsequent TLS sessions would benefit from fast resumption and be much less costly for the resolver CPU.

3.5 Conclusion

In this chapter, I have evaluated the performance of new transport protocols for DNS. I have adopted two complementary points of view: on the one hand, the key objectives of the DNS client are low latency and reliability; on the other hand, the DNS resolver needs to handle a large number of clients while making reasonable use of resources such as CPU and memory.

From the DNS client perspective, I have shown that persistent TCP connections are very efficient at reducing worst-case latency when faced with message loss. Lost queries can be recovered faster when the client sends frequent queries on the connection, although sending queries too frequently can produce Head-of-Line blocking. By extension, DNS-over-TLS would provide similar latency properties because TLS is based on TCP. In comparison, UDP has no way to reliably detect lost messages and disambiguate them from a slow resolver. It also cannot take

advantage of subsequent queries to recover the loss of previous queries. As a result, the application either retransmits conservatively and introduces several seconds of latency, or retransmits more aggressively, leading to spurious retransmission and duplicate queries. UDP is currently the default transport protocol for DNS.

From the DNS resolver perspective, I have shown that the CPU and memory resources needed by TCP and TLS are higher than for UDP but are still reasonable. For TLS, most of the CPU resources are expected to be spent in TLS session establishment. This makes it even more important to maintain long-lived persistent connections, trading reduced CPU cost for more memory usage. TLS session resumption is a promising future development that could overcome this trade-off.

In both cases, my approach is to explore these research questions through controlled experiments – that is, experiments that involve real systems in a controlled environment. The conditions of the experiments are different from real-life deployments: for instance, in the experiments, the resolver runs entirely from its cache and the network latency is fixed. As a result, real deployments will likely experience a different level of performance than the results of my experiments. However, controlled experiments provide a key advantage: specific effects and mechanisms can be isolated and understood in depth. For instance, I was able to identify that Early Retransmit plays a key role in achieving good latency with DNS-over-TCP under packet loss. Controlled experiments also allow to make meaningful comparisons, such as comparing the performance impact of UDP, TCP and TLS on recursive resolvers, because the conditions are the same for all experiments.

Overall, the results of this chapter support the move to new transport protocols for DNS. DNS-over-TLS (DoT) and DNS-over-HTTPS (DoH) are becoming increasingly popular in the Internet, mainly because of the privacy guarantees they provide. My results show that, in addition to this privacy advantage, the mere fact that they rely on a persistent TCP connection has latency advantages when the network experiences losses.

There is still room for future protocol improvements. DNS-over-QUIC [45] is expected to solve the head-of-line blocking issue between DNS requests, while TLS session resumption [90, 104] has the potential to dramatically lower the CPU cost of re-establishing connections for returning clients. More experiments would be needed to better understand this cost and how much session resumption would help to reduce it.

Experiments on the client side were very helpful to understand the relation between the needs of an application and the features provided by a transport protocol. It led

me to more general considerations about multiplexing, how to avoid Head-of-Line blocking, and scheduling. The next chapter explores these questions in depth, with a particular focus on scheduling.

4 Stream-aware multipath scheduling

Application requirements have changed significantly since the early days of the Internet, when `telnet` and FTP where the dominant applications. Applications now manipulate many concurrent flows of data and need lower end-to-end delay; transport protocols have evolved accordingly to provide more elaborate services to these applications. In particular, the concept of *multi-stream transport*, once reserved to niche applications such as telephony signalling, is now becoming mainstream and is being applied to web applications and real-time communication. Simultaneously, the emergence of *multipath communication* provides even more possibilities to application and transport protocols, but it also brings its fair share of challenges. In this chapter, I will mostly focus on challenges related to *scheduling*.

In Section 4.1, I start by reviewing the emergence of multi-stream applications and transport protocols. In Section 4.2, I introduce a model that can be used to reason about multi-stream scheduling. Then, in Section 4.3, I extend this multi-stream model to account for multipath communication. I analyse the associated impact on transport protocol schedulers, highlighting the need for *stream-aware multipath schedulers*. The main contribution, in Section 4.5, is the proposal of a new *stream-aware multipath scheduling algorithm*, called SRPT-ECF, that addresses these challenges. In Section 4.6, I extend the algorithm to the online case; in Section 4.7, I evaluate it on HTTP/2 traces and show it exhibits good properties. Finally, in 4.8, I discuss how it could be implemented as part of a MPQUIC implementation.

4.1 Background on stream multiplexing and scheduling

To understand why stream scheduling is necessary, we first need to look at how the needs of applications have changed over time, and how applications and transport protocols have adapted to these new requirements by multiplexing data. I then briefly review existing protocols that perform stream multiplexing (SCTP, QUIC, HTTP) and their possible scheduling strategies. Finally, I introduce a multi-stream

scheduling model that abstracts away protocol-specific details. I will use this model in the rest of this chapter.

4.1.1 From single-stream to multi-stream transport: an historical perspective

In the early Internet, the requirements of applications were basic. If you needed to send lightweight messages, you would use UDP; if you wanted a reliable connection or needed to transfer a file, you would use TCP. Applications such as `telnet` typically only manipulated a single *data flow*[1], while other applications such as FTP clients or early HTTP servers manipulated just a few concurrent flows at most.

This is reflected in the historical design of the Apache web server and its "prefork" model that spawns several Unix processes, dedicating an entire process to each client. In this model, an Apache process handles a single client connection from beginning to end before it can move on to the next client[2]. The downside of this approach is that it wastes server resources whenever a connection is left idle by the client or is blocked by the congestion window. The newer and more efficient "event" model [31] only became a viable alternative with the release of Apache 2.4.0 in January 2012 [32], and has been the default model since then.

In these early applications, each application data flow was typically mapped to a separate connection. For instance, FTP opens a TCP connection for its control flow, and then opens a new TCP connection each time it needs to upload or download the content of a file. As another illustration, HTTP/1.0 [7] required the use of a separate TCP connection for each request, even when several requests are made to the same web server.

Trend 1: more data flows Over time, applications started to manipulate more and more data flows concurrently. For the web in particular, HTTP can easily need to transfer up to hundreds of resources from a server to a client just to display a web page. A reverse proxy or load balancer may handle simultaneous data flows for tens of thousands of clients. Other examples include tunnelling several TCP flows within a SSH connection, or receiving several audio, video and text streams from a video-conferencing server.

[1] I am intentionally vague on what a "flow" of data actually is, because it depends on the application (stream, message, file...). It can be loosely defined as a set of data that has a consistent meaning for the application, i.e. data from different flows semantically belongs to different objects.

[2] The "worker" model is similar but additionally uses threads instead of relying only on processes.

Trend 2: more costly connections Despite a very significant improvement in hardware computing resources over time, opening new connections has actually become more costly.

First, many applications are now encrypting their communications by default, using protocols such as Secure Shell [113] (SSH) or Transport Layer Security [90] (TLS). Secure connection establishment uses asymmetric cryptography algorithms, which consume a large amount of processing resources. Thus, even a high-end server may have trouble processing a large amount of simultaneous connection establishment.

In addition, "middle-boxes" [82] are now widely used throughout the Internet, and they typically keep a state for each connection they observe or process: this includes for instance stateful firewalls, Network Address and Port Translation (NAPT or more commonly called "NAT") devices [99], or Intrusion Detection Systems (IDS). Each connection consumes a potentially scarce resource on these middle-boxes, such as memory or port number.

Lastly, and perhaps most significantly, *latency* has become a bottleneck for most applications, as described in Chapter 1. As such, the latency of opening a connection is becoming more costly relatively to other network operations, due to both transport session initialization and congestion control. This is what motivates CDNs (Content Delivery Networks) to deploy edge servers as close to the user as possible, but even CDNs cannot completely offset the latency cost of opening a new connection.

The need for multiplexing Combining these two trends, it becomes too costly to create a new transport-layer connection for each data flow. This has led to a push to *multiplex* several flows of data within a single connection whenever possible. Note that this kind of multiplexing is conceptually distinct from transport-layer multiplexing — that is, delivering data to the correct application socket depending on transport-layer port information.

A simple way to multiplex application data has been introduced as early as 1997 in HTTP/1.1 using persistent connections [29]. Since the length of a HTTP message can generally be determined by the receiver, it is possible to "pipeline" several requests or responses in the same connection, thus amortising the cost of connection establishment. However, this is not very flexible, because a HTTP message needs to be entirely transmitted before starting transmission of the next one. In addition, the order of pipelined responses must be the same as the corresponding requests, further limiting the potential for taking advantage of this feature.

Meanwhile, the Stream Control Transmission Protocol [100] (SCTP) was specified in 2000 [102] as a generic transport protocol with built-in message multiplexing. The application gives messages to SCTP for sending, and associates each message to a *stream* thanks to a stream identifier. Messages are then transmitted reliably by SCTP as a series of interleaved "DATA chunks", where the length of a DATA chunk is designed to fit in a single IP packet to avoid IP fragmentation. On the receiver side, messages are reassembled from their individual DATA chunks. Then, messages belonging to the same stream are delivered in-order to the receiving application, while messages from independent streams have no ordering constraints and can thus be delivered to the application without blocking each other.

15 years later, HTTP/2 [6] introduced a similar mechanism for HTTP messages on top of TCP. Each *HTTP message* belongs to a *stream*, and messages from concurrent streams can be freely interleaved by the sender. To achieve this, a HTTP message is broken down as a series of *frames*, typically up to 16 KB in length, that are transmitted atomically on the TCP connection. Interleaving is achieved by transmitting frames from different messages in succession, which means that frames should be small enough to match the desired multiplexing granularity. This concept of "frame" plays a similar role as "chunks" in HTTP/1.1 and "DATA chunks" in SCTP, although they differ in their details.

Finally, QUIC [66] offers a multi-stream service inspired from HTTP/2, but is intended as a more generic transport protocol that could be used by other application protocols. In addition, instead of relying on TCP like HTTP/2 did, QUIC implements a new reliable stream-oriented transport service directly over UDP. As a consequence, while QUIC also uses "frames" to interleave data from different streams, they are actually very different from HTTP/2 frames: QUIC frames must fit into a single UDP datagram. Therefore, QUIC frames are much more closely related to SCTP DATA chunks than they are to HTTP/2 frames.

Note that, unlike SCTP, QUIC provides no built-in way to frame several messages within a stream: each QUIC stream is simply exposed as a continuous byte stream to the application. The application either needs to send a single message per stream and close the stream to signal the end of message, or it needs to add the necessary framing as part of the application protocol itself. This second approach is taken in the current draft HTTP/3 specification [8] so that it can continue using a similar framing as HTTP/2.

4.1.2 Scheduling multiplexed streams

As we have seen, applications and transport protocols now multiplex data from several streams on a single connection. This raises a *scheduling* question: when the sender has several active streams with new data to transmit, which one should it transmit data from? Should it use Round-Robin? If the application assigns priorities to streams, should they be interpreted as strict priorities or as input to a Weighted Round-Robin algorithm? What is the metric to optimise?

I review recent literature on multi-stream scheduling. Note that the HTTP literature tends to use the term *prioritization* while the SCTP literature uses the term *scheduling*, but there is no significant difference between the two concepts.

Conceptually, *stream scheduling* is necessarily performed by the sender. However, some protocols allow the receiver to provide *scheduling hints* to the sender: the sender can then schedule data according to these hints, or it can choose to partially or completely ignore the hints. Scheduling hints are mainly used for request-response protocols such as HTTP, because the receiver needs to be aware of which streams are going to be sent by the remote peer.

Client-side HTTP prioritisation via scheduling hints HTTP/2 [6] defines a complex scheme of scheduling hints. It allows HTTP clients to express many different scheduling strategies that they would like a server to implement. It works by specifying a weighted dependency tree between HTTP resources: a parent node has strict priority over its children, and siblings should be served with Weighted Round-Robin once their parent has completed. The dependency tree is maintained on the server side and is updated dynamically according to client messages.

A recent work [110] describes how web browsers use this dependency tree system to provide a variety of scheduling hints. It uncovers very different approaches: simple Round-Robin (Internet Explorer, Edge), Weighted Round-Robin (Safari), strict priority classes combined with FCFS [First-Come First-Serve] within each class (Chrome), complex tree-based strategy (Firefox). When looking at the impact of the prioritization scheme on the Page Load Time, an interesting pattern emerges: the advanced strategies from Chrome and Firefox perform very well, especially in low-packet-loss environments, but the naive FCFS strategy is almost as good. On the other side of the scheduling spectrum, Round-Robin consistently performs poorly.

This dependency tree system has been found to be complex to implement and difficult to use correctly: as a result, when designing HTTP/3, there was a debate

as to whether this system should be simplified, adapted or simply carried over unchanged [71]. This debate was partially settled in September 2019 by completely dropping support for scheduling hints in HTTP/3 until a better scheme could be found [8].

At around the same time, a simpler system of scheduling hints has been proposed [80], but it is still work-in-progress and has yet to be adopted. With this scheme, a request carries two parameters: the "urgency" parameter which defines one of 8 strict priority classes, and the "incremental" parameter which specifies whether resources within the same priority class should be delivered sequentially or in parallel. The idea is that resources that can be processed "incrementally", such as HTML documents or progressive images, benefit from loading in parallel; on the contrary, if the "incremental" parameter is set to false, the resources should be delivered sequentially in the same order as the requests.

Server-side HTTP prioritisation Literature on server-side HTTP prioritisation is surprisingly poor, considering that web servers are ultimately responsible for the scheduling of stream data.

The HTTP/2 specification [6] defaults to serving streams with Round-Robin if the client provides no scheduling hints. However, Round-Robin has been found to be the worst strategy for loading typical web pages [110].

CloudFlare developed an alternate priority scheme that runs on the server side [72, 73]. This scheme is called the "bucket" priority scheme in [71]. This scheme implements strict priority classes (the "buckets"). Within each priority class, resources can again be given strict priority against each other, or are served with a two-tier Round-Robin strategy. This system combining "urgency" and "concurrency" inspired the most recent HTTP priority proposal described above [80].

Hybrid HTTP prioritisation Vroom [91] takes an interesting approach that splits the scheduling effort between client and server. This takes into account that a client often needs to load HTTP resources from several independent servers: as a result, a single server does not have complete control over the scheduling problem, and the client should be partially left in charge of prioritizing its requests across several servers. On the other hand, the server has initially access to more information about the content, so it is in a better place to prioritize important resources.

In Vroom's proposed solution, the server directly "pushes" important resources that are critical for the client to start loading the web page, such as HTML and

Javascript files. This uses the server push functionality of HTTP/2. For less important resources or resources that are provided by third-party servers, the server simply sends "dependency hints" that let the client know about the resources it will likely need. This is especially helpful for CPU-constrained clients, because the client can already start fetching these resources before it has finished parsing HTML files or executing Javascript code.

More precisely, from a scheduling perspective, the server classifies resources in one of several classes:

1. important local resources (HTML, JS): they are pushed to the client using HTTP/2 server push;
2. important third-party resources (HTML, JS): they are sent as high-priority hints to the client. The client fetches these resources as soon as possible;
3. other resources, both local and third-party: they are sent as low-priority hints to the client. The client fetches them once it has finished downloading the important resources.

Within each class, the server estimates which resources will be needed first by the client, and it orders resources accordingly (either by pushing them in order or by sending ordered dependency hints). When receiving dependency hints for a given priority class, the client simply fetches resources in the order provided by the server.

Finally, when actually serving HTTP resources and there are several concurrent requests from the same client, the server uses FIFO (First-In First-Out) instead of Round-Robin. This is because requests are already ordered "optimally" thanks to the dependency hints; using Round-Robin would slow down the completion of the most important requests.

Overall, Vroom tries to discover most of the required HTTP resources as soon as possible to minimize round-trips. Then, client and servers coordinate to transfer resources sequentially in the order they are needed by the client.

SCTP stream scheduling While SCTP has native multi-stream support, there are no stream scheduling guidelines in the SCTP specification itself [100]. In practice, implementations take different approaches: according to [95], FreeBSD uses Round-Robin while Linux uses First-Come First-Server (FCFS).

Early work [95] explored two scheduling aspects of SCTP. First, it is beneficial to only bundle messages from a single stream within an IP packet: this is called "Per Packet Scheduling" by the authors. This reduces the impact of packet loss because only one stream will be affected by the resulting Head-of-Line blocking. Second, it is useful for a receiver to be able to control which stream is prioritised by the sender. Conceptually, this can be achieved through per-stream flow control; however, since SCTP does not provide per-stream flow control, the authors achieve the same goal by applying stream priorities on the sender side.

Another work [109] proposed a "pluggable" stream scheduling system with an implementation in the Linux kernel. This allows the application to select a scheduling strategy at run-time, for instance FCFS, Round-Robin, Weighted Fair Queuing or Strict Priority. The choice is left to the application because it depends on its specific requirements.

QUIC stream scheduling QUIC, just like SCTP, is a generic transport protocol. As such, deciding which scheduling strategy should be used mostly depends on the application requirements.

The only generic piece of advice in the QUIC specification is to avoid bundling data from several streams in the same packet, for exactly the same reason as in SCTP: it would cause Head-of-Line blocking for several streams in case of packet loss [50]

An example of application-specific stream scheduling is the Deadline-Aware Transport Protocol [97], based on QUIC. For each "block" of data, a priority and a deadline are provided by the application, and the transport protocol must schedule data to respect both the deadline constraints and the priorities.

4.2 The multi-stream scheduling model

As we have seen, there are many approaches to stream scheduling, and they are quite dependent on the application requirements. In addition, multi-stream transport protocols such as QUIC and SCTP provide similar but slightly-different services. Reasoning about them would be made easier with a model that encompasses features from both protocols.

Throughout this chapter, I will use a multi-stream scheduling model that can be applied to a broad class of applications that use either QUIC or SCTP. This includes

HTTP/3 in particular. Inspired mostly by SCTP, the model defines the following components that are visible to a scheduling algorithm:

message a piece of data with a well-identified beginning and end. A message is the network representation of a single abstract "object" or "resource" in the application. The size of the message needs to be known in advance: when the application creates a new message, it should provide the scheduler with the total size of this message.

message size The total data length of a message, expressed in bytes.

message completion time In the model, the receiving application is only interested in the whole message: it can start receiving parts of a message, but I assume that it will only be able to process the data once the complete message has been received. Thus, the performance metric I will consider is the **application-level completion time of individual messages**. It is defined as the time elapsed between two instants: 1) the instant at which the sending application provides a message to the transport protocol; 2) the instant at which the message has been fully delivered to the receiving application.

message priority The sending application can attach a *priority* to a message, which is interpreted as a strict priority: if messages m_1 and m_2 have priorities P_1 and P_2 such that $P_1 > P_2$, then the whole message m_1 should be scheduled before message m_2. However, priorities do not constrain delivery of messages to the receiving application: in special circumstances, it is possible that message m_2 might be fully received before message m_1, despite the priorities. In that case, message m_2 is delivered to the application before message m_1.

stream Each message is sent on a *stream*. In the model, only a single message can be sent on a given stream: as such, messages and streams are effectively equivalent.

data-chunk the transport protocol splits messages into *data-chunks* so that they can be multiplexed on the sending side. The transport protocol then reassembles messages from their data-chunks on the receiving side. In addition, data-chunks need to fit constraints from lower layers (UDP, IP). Data-chunks are typically not seen by the application since they are only used internally by the transport protocol. However, the scheduler needs to manipulate data-chunks: it decides how to split messages and combine their data-chunks, and may even decide to bundle several data-chunks in a single lower-layer datagram.
This is called a *frame* in QUIC. However, the word "frame" may be confusing because it has three different meanings in HTTP/2, QUIC and HTTP/3. As a

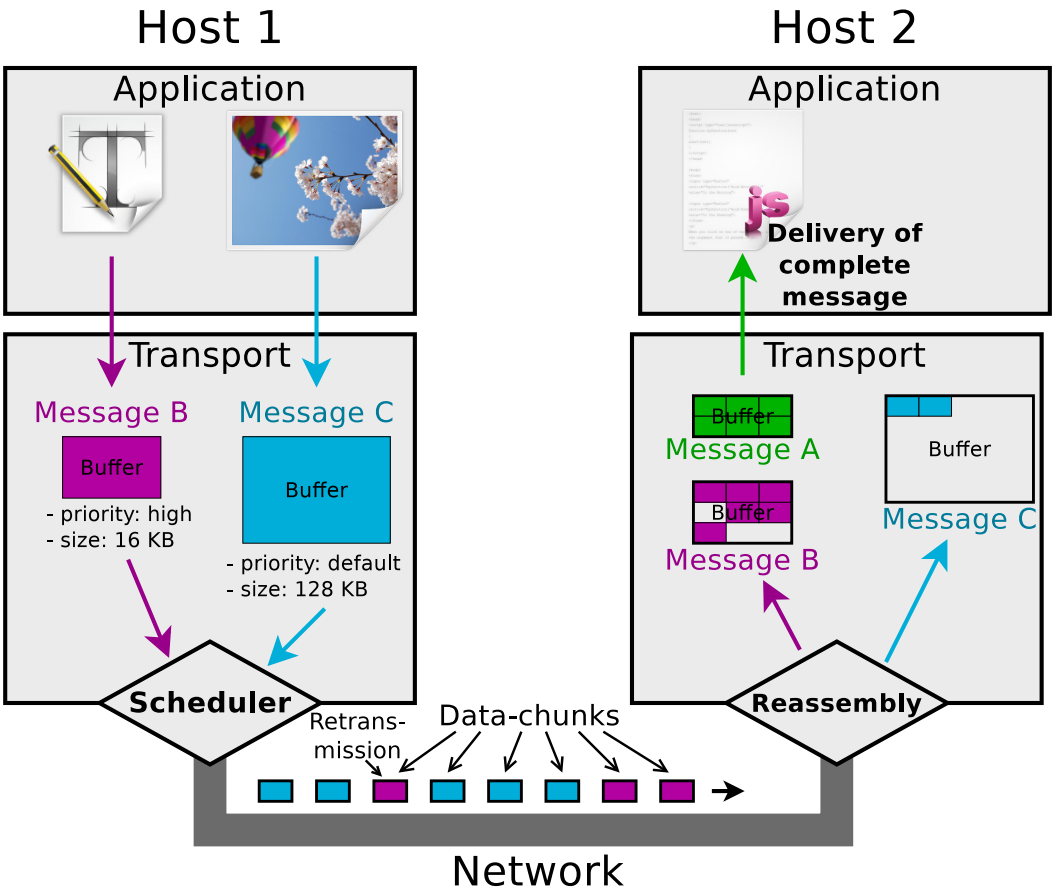

Figure 4.1 Summary of the multi-stream scheduling model: applications create *messages* to transport resources, and each message has a *size* and *priority*. The scheduler then decides which message should be served, and sends the content of messages as a sequence of data-chunks. Data-chunks from different messages can be freely interleaved: this can be useful for instance when retransmitting lost data-chunks. Messages are reassembled on the receiver side; conceptually, they are delivered to the application only once they are complete.

consequence, I use the unambiguous expression "data-chunk" with a definition that matches the "DATA chunks" concept from SCTP.

Figure 4.1 summarises the main aspects of the multi-stream scheduling model. For simplicity, the model merges the concept of *message* and *stream*: only a single message can be sent on a stream. In addition, the message size needs to be known in advance. I discuss the relevance of these two simplifications and other aspects of the model below.

Applicability to SCTP

The message-oriented semantic of the model is directly inspired from SCTP.

The main difference between the model and SCTP is that SCTP allows several messages to be sent on a given stream. SCTP provides an ordering guarantee between these messages: they will be delivered to the receiving application in the same order as they were provided by the sending application.

From a scheduling perspective, this is equivalent to introducing strict priorities between messages belonging to the same stream, which is possible in the model.

However, SCTP would need an additional serialisation step at the receiver to ensure that messages are indeed delivered in the correct order. This slight difference between the model and SCTP is not critical because it happens in very specific conditions: packet loss would need to happen in such a way as to sufficiently delay one message, while a lower-priority message in the same stream is not delayed: this lower-priority would complete sooner although it started being transmitted later. The impact of this slight difference is that the model will predict a slightly optimistic completion time compared to SCTP in this specific case.

Applicability to QUIC

QUIC does not natively support message framing within a stream: as such, it is limited to a single "implicit message" per stream. That is, opening a new stream is the same as starting a new message, and closing a stream signals the end of the message. This exactly matches the model.

The main difference between QUIC and the model is that QUIC is more stream-oriented. It allows an application to consume any amount of data on a stream and start processing it, even before the stream ends. Some applications might find this feature useful, while others might only be able to process the data when they have fully received the content of a stream. Thus, the model only applies to this second category of applications, i.e. those using QUIC for message-oriented communication.

Finally, QUIC allows the application to assign priorities to streams as part of the application-transport API (e.g. a socket-like API) [50]. While QUIC implementations are encouraged to take these priorities into account, no specific scheduling mechanism is specified. Thus, even though other priority models are possible, my choice of strict priorities in the scheduling model is fully compatible with QUIC.

Applicability to HTTP

The model matches the use-case of transferring web resources between a server and a client with HTTP. Each web resource is represented as a message. The message size is typically known in advance for static resources (Javascript files, images...).

In some cases, the total size of a HTTP message is not known in advance, for instance when generating responses on the fly. In that case, the sender transmits the message payload as a sequence of HTTP DATA frames until it reaches the end of the message. In the model, this can be translated as a message with an initial size equal to infinity. Whenever the sender eventually determines the remaining amount of data, it can inform the transport protocol about the actual message size.

Finally, HTTP clients and servers may start processing partial messages. For instance, it is usually beneficial to start parsing a large HTML document even if it has not been fully received, so that links to resources can be discovered and requested as soon as possible. Progressive images are another example of partial message processing. These use-cases can be modelled with several successive messages with appropriate priorities. For instance, a progressive image can be modelled with three messages:

1. a high-priority message containing the image metadata (type, width, height)
2. a medium-priority message containing a low-resolution version of the image
3. a low-priority message containing the full image data

Specific notes about HTTP/2 While the model can be applied to HTTP/2, it is not a very convenient fit. Since HTTP/2 relies on TCP, all scheduling decisions need to be made before passing data to TCP. This means that any data waiting in the TCP send buffer cannot be rescheduled, preventing late scheduling decisions or preemption of high-priority messages over low-priority messages. The TCP send buffer needs to be at least as large as the Bandwidth-Delay Product (BDP) for good performance, which can mean as much as several megabytes. See Section 2.3.2 in Chapter 2 for more details.

Overall, HTTP/2 matches the model if all resources are known at the same time. It otherwise provides very limited support for online scheduling – the case where new messages can be created at any time, potentially preempting existing messages.

Specific notes about HTTP/3 In HTTP/3, at most a single message with a payload body is allowed on a QUIC stream [8]. This matches the model.

In addition, with good cooperation between the QUIC layer and the HTTP/3 layer, it is possible to schedule data as late as possible, even if it is already in the QUIC send buffer. This is useful when a new high-priority message is created and suddenly needs to preempt existing messages.

Overall, notwithstanding the general applicability of the model to HTTP, both HTTP/3 and SCTP provide the best match with the model.

4.3 Stream-aware multipath scheduling

In the previous section, I introduced a model to study stream scheduling algorithms. I now extend this model to the *multipath* case and discuss the challenges that arise from this extension.

4.3.1 Multipath scheduling with several streams

The previous scheduling model in Figure 4.1 was tackling the problem of scheduling data from several streams. I now assume that there are several paths to choose from when sending data: this adds a new dimension to the scheduling problem. I call this problem *stream-aware multipath scheduling* in line with recent literature [87]. I show the basic principle of how a stream-aware multipath scheduler works in Figure 4.2.

A stream-aware multipath scheduler needs to solve two separate problems: *stream scheduling* and *path allocation*. I detail these two problems below:

Stream scheduling

The scheduler needs to choose which stream(s) it will service at any given time. It may use priorities provided by the application. This is the same problem that I described in Section 4.1.

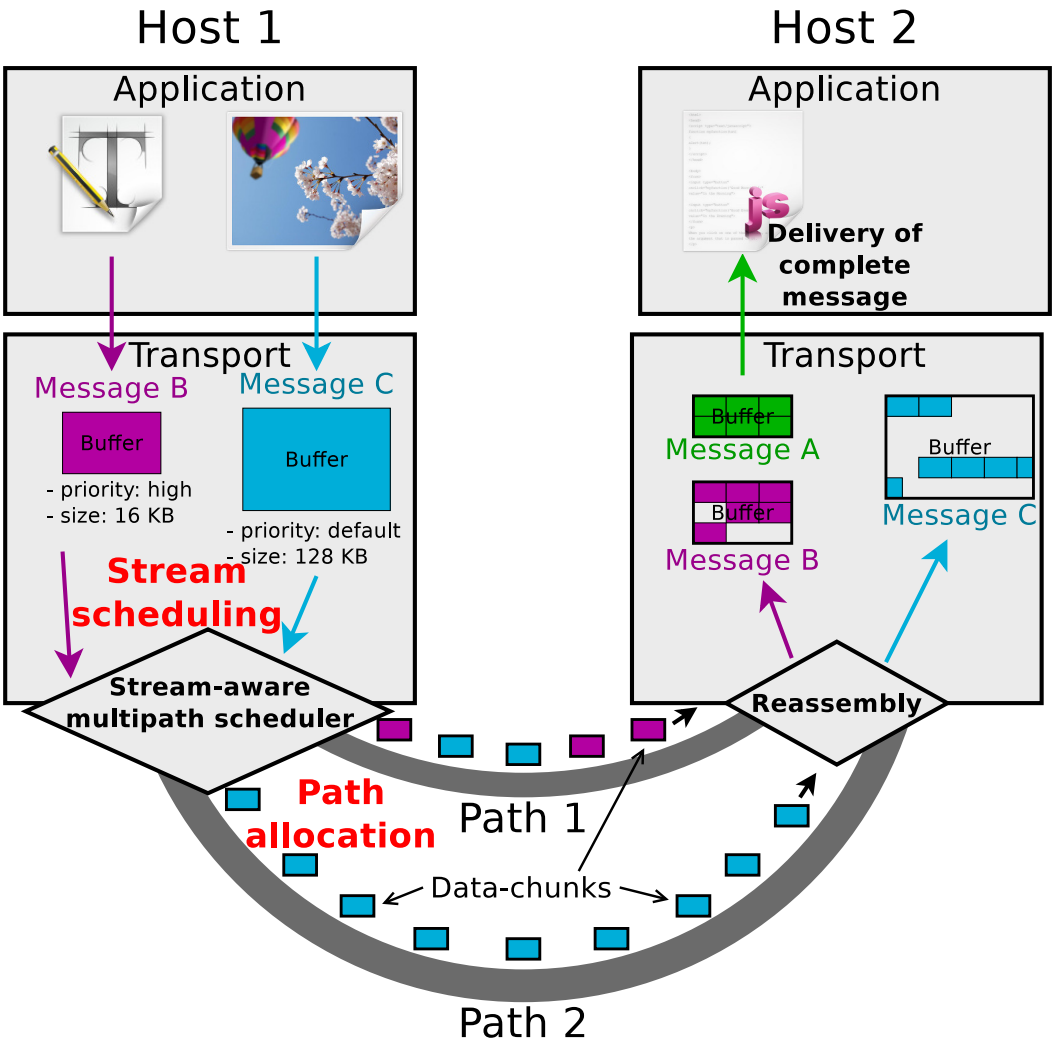

Figure 4.2 The stream-aware multipath scheduling model. Compared to Figure 4.1, there is an additional dimension: several paths can be used to send data. The scheduler needs to solve two problems: *stream scheduling* and *path allocation*.

Path allocation

Once the scheduler has data to send, it needs to decide which path(s) it will use. This is similar to the classical multipath scheduling problem that I described in Section 2.4.2 of Chapter 2.

4.3.2 Shortcomings of MPTCP schedulers

Multipath TCP (MPTCP) has been the de-facto standard for multipath transport research for many years. This has led to a number of proposed multipath scheduling algorithms as described in Section 2.4.2. As such, it is tempting to reuse schedulers developed for MPTCP in this new multi-stream case. This is the approach taken by early works on MPQUIC [19, 105].

However, MPTCP was specifically developed to match the semantic of TCP: it handles a single stream of data with no visible framing. Whenever the application multiplexes several messages on a connection, as discussed in Section 4.1, Multipath TCP is not aware of this fact and this may negatively impact performance. More specifically, I identify two main shortcomings of MPTCP when handling multiplexed messages: the first is related to scheduling, while the second is inherent to the way the protocol delivers data.

Shortcomings of MPTCP scheduling: serialisation at the sender

When using MPTCP, a *serialisation* step is needed before data can be handed out to MPTCP: data from several stream needs to be ordered and organised in a single buffer. The application is responsible for this serialisation step and is thus forced to perform stream scheduling itself. This behaviour is illustrated in Figure 4.3.

The main problem of this serialisation step is that it completely decouples stream scheduling (performed by the application, on the left of Figure 4.3) from path allocation (performed by the MPTCP implementation, on the right of Figure 4.3). It causes a lack of visibility: the path allocation algorithm does not see the boundaries between streams. As a result, it cannot optimise the scheduling process for a given stream, and it may unknowingly take poor scheduling decision such as sending part of a stream on a high-latency path: this may cause delayed stream completion.

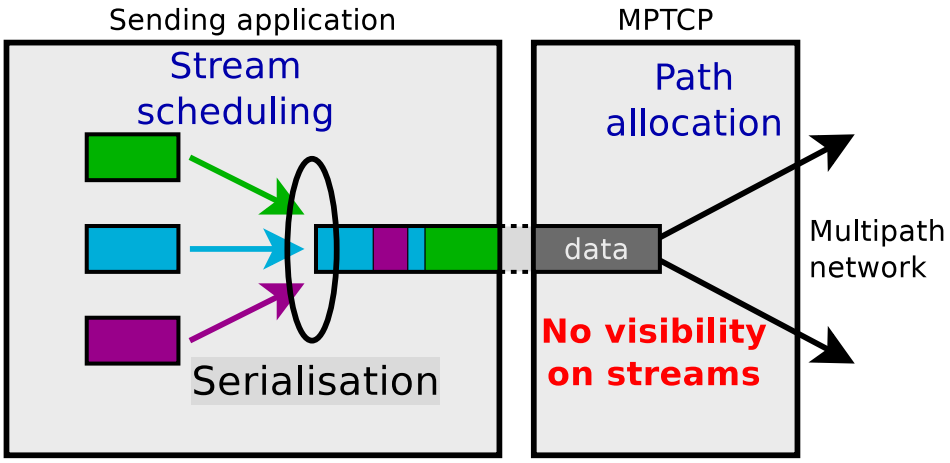

Figure 4.3 Illustration of the *serialisation* step incurred by MPTCP when scheduling multiplexed data. Stream scheduling and path allocation are performed completely independently (by the application and by the MPTCP scheduler respectively).

For instance, the Earliest Completion First [68] scheduler aims to minimise the completion time of the whole TCP data flow; however, it cannot consider the completion time of individual messages.

Shortcomings of MPTCP delivery: serialisation at the receiver

The second issue happens on the receiver side: the data flow is again serialised into the receive buffer. Since paths have different latencies, it is common to obtain gaps in the receive buffer. When such a gap happens, it blocks delivery to the application because the semantic of TCP implies strict in-order delivery. However, the blocked data may contain data that belongs to a different stream than the missing data in the gap, and this blocked data could be used straight away by the application: this is called Head-of-Line blocking. This behaviour is illustrated in Figure 4.4.

This head-of-line blocking issue is much more frequent when using multiple paths. On a single path, it mainly happens when a packet is lost and needs to be retransmitted: all subsequent received packets are blocked until the retransmission reaches the receiver. With multiple paths, there is an additional significant cause of head-of-line blocking: different latency on each path. Many MPTCP schedulers such as DAPS [92, 65] or BLEST [26] try to avoid Head-of-Line blocking by estimating network characteristics before scheduling data, but there will always remain some cases of head-of-line blocking because the latency of a network is variable over time.

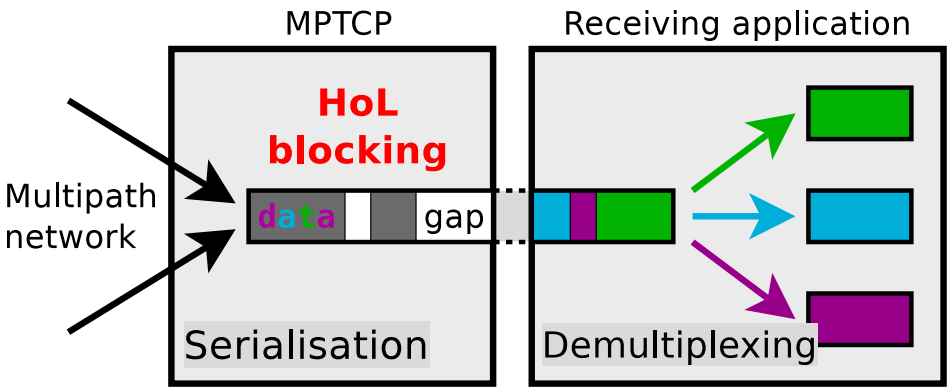

Figure 4.4 Illustration of the serialisation step at the receiver, before stream demultiplexing can be done. Any gap in the received data will cause Head-of-Line blocking.

In the case of the message-based model, head-of-line blocking is mainly an issue when it delays completion of a stream. That is, if the last data-chunks of a message M_1 are received but are blocked by missing data-chunks from another unrelated message M_2, the completion of message M_1 is delayed.

To sum it up, MPTCP schedulers provide a good basis to solve the *path allocation* problem individually. However, to tackle the more general *stream-aware multipath scheduling* problem, it is necessary to solve both stream scheduling and path allocation simultaneously, a double task that MPTCP schedulers cannot solve on their own.

4.3.3 Stream-aware multipath schedulers

I now present a few illustrative stream-aware multipath scheduling strategies. I start by reviewing existing algorithms for *stream scheduling* and *path allocation*. I then show how they can be combined: while stream scheduling and path allocation are distinct problems, a scheduling algorithm needs to integrate both aspects to obtain good performance.

I present scheduling algorithms in a normalised form `<stream scheduler>-<path allocation>`. For instance, `<Round-Robin>-<MinRTT>` means that Round-Robin is used to choose which streams are served, while the MinRTT strategy is used for path allocation.

Table 4.1 Stream scheduling algorithms

Stream scheduling algorithm	Description
Round-Robin (RR)	Serve all active streams in parallel. It is possible to take priorities into account using Weighted Round-Robin (WRR).
Sequential	Serve streams exclusively: finish a stream before starting the next one. The order can be based on message priorities, message lengths, or other criteria.
First-Come First-Serve (FCFS)	Special case of Sequential where the order is simply the order in which messages were handed out to the scheduler.
Strict priority classes	Messages with higher priorities are served first. A secondary stream scheduling strategy (RR, Sequential) is needed to schedule streams with the same priority.

Table 4.2 Path allocation algorithms

Path allocation algorithm	Description
Single Path	Only use a single path to send data for a message.
MinRTT	Use the path with lowest RTT if congestion control allows sending on it, otherwise fallback to another higher-RTT path.
Earliest Completion First (ECF)	Try to ensure simultaneous completion of messages on multiple paths.

<Round-Robin>-<MinRTT>

This is the most straightforward algorithm: it combines the default stream scheduling strategy of HTTP/2 (Round-Robin) with the default path allocation strategy of MPTCP (MinRTT). There is no interaction between the two strategies: MinRTT only takes into account path characteristics when allocating a data-chunk to a path and it performs this allocation without any consideration of which message this data-chunk belongs to. As a result, this combined algorithm suffers from the first shortcomings described in Section 4.3.2 because it has an implicit serialisation step at the sender, even when used with MPQUIC.

<Round-Robin>-<ECF>

Rabitsch et al. [87] proposed the SA-ECF scheduler that combines Weighted Round-Robin and ECF. It provides a good example of interaction between stream scheduling

and path allocation: to perform path allocation, ECF needs to estimate the completion time of a message, and this is only possible by knowing the stream scheduling strategy (WRR in this case). I will describe in much more details how ECF interacts with the stream scheduling strategy in Section 4.4.

<Round-Robin>-<Single Path>

This single-path design has been proposed for CMT-SCTP and was named the "Fixed" scheduler [20]. It sends each stream on a single path. When multiple streams are mapped to the same path, they share its capacity with Round-Robin.

Sending a message on a single path has the advantage of avoiding out-of-order reception of packets that would be caused by heterogeneous path latency. However, large messages will not be able to take advantage of the combined throughput of all paths. As such, it is mostly beneficial for small messages or for a high number of concurrent messages.

FStream [98] proposed a similar design for MPQUIC: messages are sorted according to their HTTP/2 priority and each message is mapped to a single path, chosen to minimize the estimated completion time of the message. Once mapped to a path, messages share the capacity of the path using Weighted Round-Robin based on their priority.

<Sticky Round-Robin>-<Single Path>

The DS scheduler, short for "Dynamic path- and Stream-based Scheduling algorithm" [23], is designed to schedule messages from SCTP streams. It does not exactly fit the scheduling model, because the algorithm schedules messages that belong to the same stream differently than messages that belong to different streams. Nevertheless, its scheduling strategy warrants a detailed explanation.

First, it provides strict priority classes: streams in a higher priority class get absolute priority over path resources.

Then, within a priority class, it uses a "sticky" version of round-robin to select which stream can send data. That is, if a stream has just sent a message on a path and has more messages to send, it will keep getting access to the same path: the stream is "sticked" to this path. Note that several streams can be "sticked" at the same time, each on its own path.

Once the "sticked" stream has no more data to send, Round-Robin is used to select another stream. This is done in two steps: first, the algorithm looks for a stream that had used this path to send its previous message. This provides another level of stickiness. As a second step, the algorithm will fall back to select any stream that has data to send.

Finally, path allocation simply sends each message on a single path. The choice of path is done by estimating which path would yield the lowest transfer time.

Overall, this strategy cleverly exploits the message structure of SCTP to avoid head-of-line blocking while still exploiting all the capacity offered by the available paths. It bears some similarity with the single-path allocation strategy from [20], but with a much finer-grained and dynamic approach.

<Sequential>-<ECF>

A recent MPQUIC scheduler for HTTP/2 has proposed a sequential stream scheduling approach [107].

To determine the sequential order in which streams are served, the authors simulate a WRR algorithm based on the HTTP/2 priority tree. They determine the order in which messages would complete in this WRR simulation, and they then apply the same order with a sequential strategy. Overall, the sequential order is determined using both message size and message priority.

Path allocation is similar to ECF in that it ensures simultaneous completion on multiple paths, although the exact method is slightly different.

<FCFS>-<ECF>

Although it was developed for Multipath TCP, the DEMS scheduler [36] splits the data flow into "chunks" and schedules each chunk independently. This is similar to the notion of message in the model, although it is less accurate because (MP)TCP does not have real visibility into application messages: a chunk may encompass several application-level messages, or a message may be split into several chunks.

In DEMS, chunks are scheduled in the order they are provided by the application (First-Come First-Serve). Path allocation tries to ensure "balanced subflow completion" for each chunk, using a technique similar to ECF.

4.4 ECF: multipath scheduling for a single message

Before describing the general SRPT-ECF algorithm, I start with the simpler problem of scheduling a single message on multiple paths. It means that we only need to solve path allocation: there is no stream scheduling.

I focus on the *Earliest Completion First* [68] (ECF) path allocation strategy and show that it is minimizes the completion time for a single message in an idealized network model. I will then use this result in Section 4.5 to prove the optimality of my proposed algorithm SRPT-ECF in the same model.

4.4.1 Network model

The network model is the following: each path p has a fixed round-trip latency D_p and a fixed bottleneck capacity or "bandwidth" B_p, expressed in bytes per second. This model is very simple but captures the two main parameters of interest for a transport protocol. Even though using fixed values for these parameters is widely unrealistic, it will allow us to more clearly highlight the important properties and requirements of stream-aware scheduling algorithms. I then discuss how to handle variable latency and capacity in Section 4.8.

4.4.2 Completion time of ECF

The metric I consider is *message completion time*, which is the difference between the time at which the message becomes available and the time at which it has been fully transferred and acknowledged.

I compute the minimum completion time as a function of (D_p), (B_p) and the message size S, assuming that paths are ordered by increasing latency. The key finding is that, for two paths, there is a size threshold S_{lim}: below this threshold, the completion time is minimized by taking only the first path, because the latency of the second path is larger than the total completion time. Above this threshold, both paths are useful, until the remaining size crosses the threshold: at this point, all remaining data should be sent on the first path. This allows both paths to complete their transfer at the same time from the point of view of the receiver. The completion time as a function of the message size S is given by:

$$C^*_{S \leq S_{lim}} = D_1 + \frac{S}{B_1} \tag{4.1}$$

$$C^*_{S \geq S_{lim}} = D_2 + \frac{S - S_{lim}}{B_1 + B_2} \tag{4.2}$$

where

$$S_{lim} = B_1 \cdot (D_2 - D_1) \tag{4.3}$$

These equations are actually valid at any point in time, giving the *remaining completion time* as a function of the *remaining size*. This *stability* property yields a natural scheduling algorithm that, at any time, simply compares the remaining size with the threshold S_{lim}. Fig. 4.5 shows such an ECF schedule with two paths. At first, both paths are used. Starting from time $t = 12.86$, the remaining size drops below the S_{lim} threshold, so only the path with lowest latency is used.

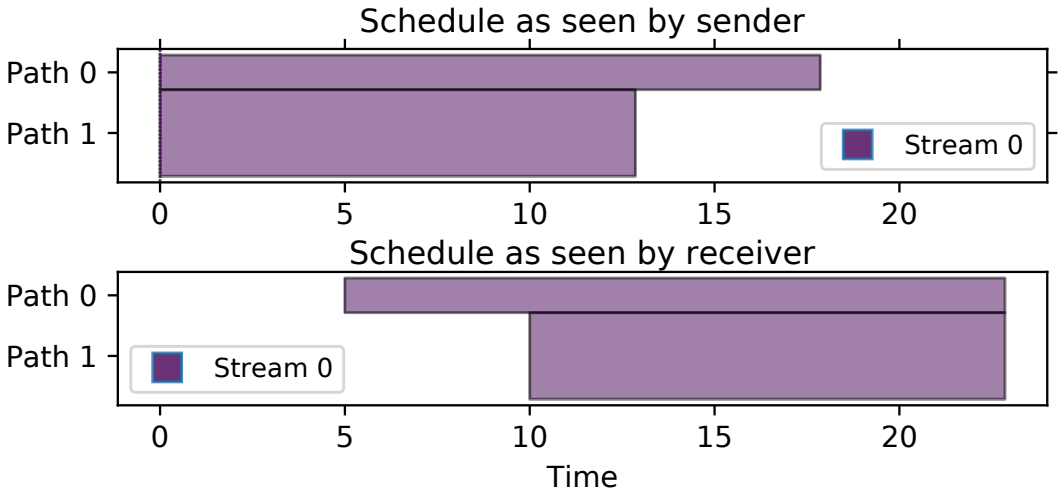

Figure 4.5 Example of an optimal schedule computed with ECF for a single message, as seen from the sender (top) and the receiver (bottom). Notice how data on the two paths completes simultaneously on the receiver side. Path 0 has capacity 200 KB/s and latency 5 ms, while path 1 has capacity 500 KB/s and latency 10 ms. The message has a size of 10 KB.

For n paths, we can show recursively that there exists $n - 1$ similar size thresholds: as the message size increases, it becomes more and more useful to use paths with larger latency. An example curve with 3 paths is shown in Fig. 4.6.

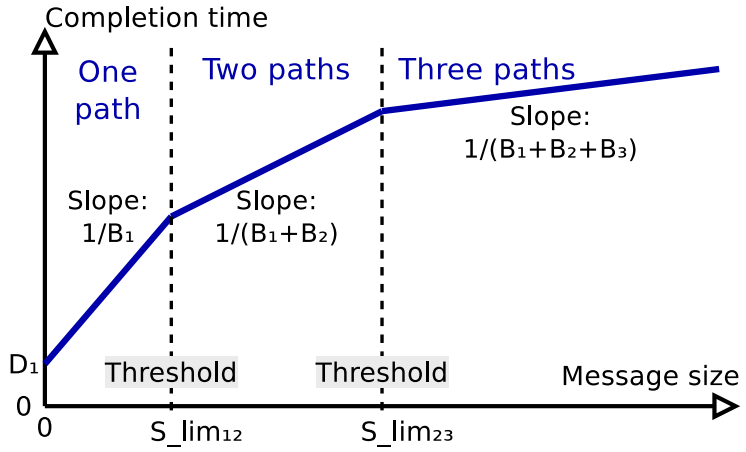

Figure 4.6 Completion time as a function of message size using ECF. S_{lim12} is the size threshold at which point the optimal solution changes from only using one path to using two paths. Similarly, S_{lim23} is the threshold between two and three paths. Paths are ordered by increasing delay. Here, D_1 is the delay of the first path, while B_i is the capacity (in byte/s) of path i.

4.5 SRPT-ECF: optimal stream-aware multipath scheduling

I now describe SRPT-ECF, my novel stream-aware multipath scheduling algorithm. I compare it with other schedulers and show that my algorithm is optimal in a simple network model.

I consider multiple messages (m_k) of size S_k. I assume that all messages are created at the same time and are available at the sender. I use the same network model as in Section 4.4 and the goal is to minimize the completion time of individual messages.

The algorithm works in two steps:

Stream scheduling: SRPT The algorithm first takes into account application-provided priorities to partition messages into priority classes: messages with higher priorities are scheduled first.

Within a given priority class, it orders messages according to the *Shortest Isolated Remaining Completion Time* principle. It computes the remaining completion time for each message *if it was alone* using ECF, and prioritise messages that have the smallest "isolated" completion time. This principle is inspired from the SRPT algorithm (Shortest Remaining Processing Time) from classical scheduling theory, hence the name of the algorithm. In the simple case where all messages are created

at the same time, it boils down to scheduling *smaller messages first* because the function from Fig. 4.6 is non-decreasing. In a real implementation, we would need to keep track of in-flight data-chunks to estimate the isolated completion time of each message. This is described in more details in Section 4.6.

Overall, messages are ordered according to a lexicographical ordering on the couples (application-provided priority, isolated completion time).

Path allocation: ECF For each message in the order determined by stream scheduling, the message is scheduled on the available paths using Earliest Completion First (ECF). For a given message, the size that is considered to determine which paths to use (see Fig. 4.6) accounts for all the messages that are scheduled before. Thus, the effective size of message m_k for ECF is: $S_k^{ECF} = \sum_{i \leq k} S_i$. This is because the data of all messages $(m_i)_{i<k}$ is scheduled before message m_k.

4.5.1 Examples

Fig. 4.7 shows a simple example of how SRPT-ECF optimally schedule two messages.

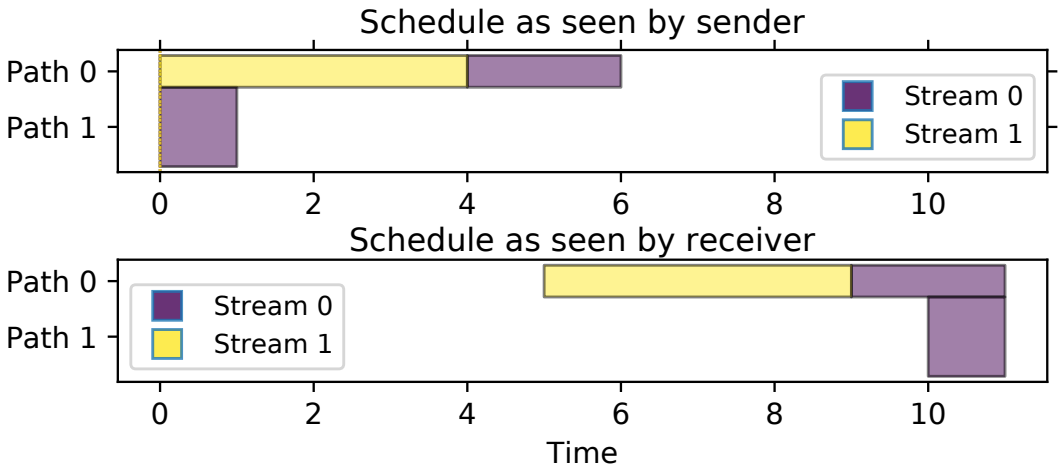

Figure 4.7 Optimal schedule computed with SRPT-ECF for two messageges with the same paths as in Fig. 4.5. Message 0 has size 900 B while Message 1 has size 800 B. Notice how the smaller message gets priority but only uses the shortest-latency path thanks to ECF, while the bigger message exploits the unused resources on Path 1. However, thanks again to ECF, it stops using Path 1 early to ensure simultaneous completion on both paths. The sequence of completion times is $(9, 11)$ and is optimal.

I also compare my algorithm with a Round-Robin strategy, more specifically SA-ECF [87]. The result is shown in Fig. 4.8: Message 0 has roughly the same com-

pletion time as with SRPT-ECF, but the completion of time of Message 1 notably increases. Overall, Round-Robin is a net loss: by introducing more fairness, it actually made the situation worse for a message while not improving the situation for the other message.

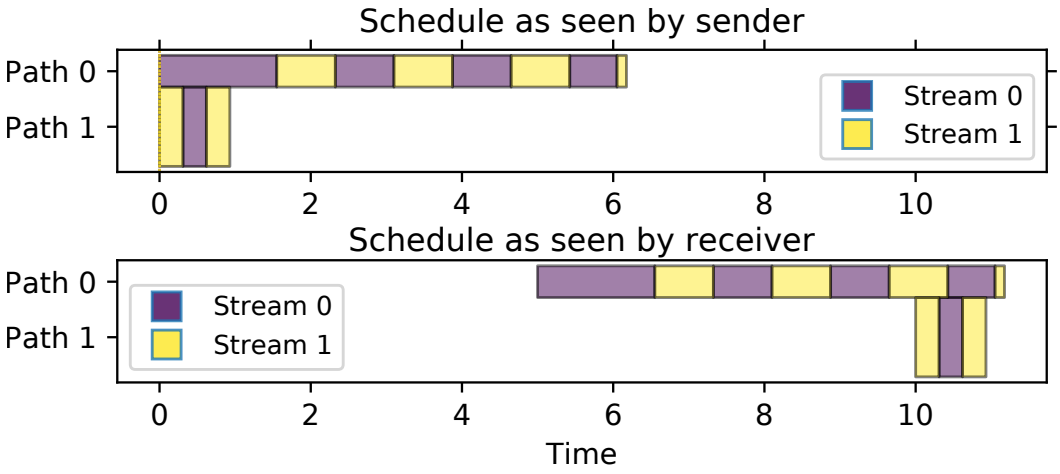

Figure 4.8 Same situation as Fig. 4.7, but with the SA-ECF scheduler [87] (Weighted Round-Robin + ECF) with 150 bytes of quantum and equal weights. The overall completion time is roughly the same as SRPT-ECF, but all messages have the worst possible completion time among ECF schedulers: the sequence of completion times is $(11, 11.2)$.

4.5.2 Properties of SRPT-ECF

My SRPT-ECF algorithm has desirable properties: it prioritizes smaller messages, lowering their completion time without having a significant impact on larger messages. In addition, since ECF may not use all paths to transmit a small message, unused resources can be used by larger and thus lower-priority messages. This behavior is illustrated in Fig. 4.7 where Message 0 is scheduled with ECF with an effective size of $S_0^{ECF} = S_1 + S_0$, causing it to use the high-latency path while it waits for access to the low-latency path.

SRPT-ECF achieves an optimal completion sequence Let (C_i) be the ordered sequence of completion times achieved by SRPT-ECF, and (C'_i) the ordered sequence of completion times achieved by any other scheduling algorithm. The order in which messages complete may not be the same for both cases: C_1 might be the completion time of message m_1 while C'_1 might be the completion time of message m_3.

The sequence (C_i) is optimal in the following sense:

$$\forall i, \quad C_i \leq C'_i \tag{4.4}$$

This result can be proved by showing that in any optimal schedule, the overall schedule behaves like ECF with a single message of size $\sum S_k$, and that the SRPT ordering is the best ordering. This is because the total completion time $C^*(\sum_{k\in\mathcal{S}} \mathrm{S}_k)$ of any subset of messages $\mathcal{S}$ is a non-decreasing function of their total size.

Optimal average completion time In particular, the optimality result implies that SRPT-ECF is also optimal for the average completion time of messages.

Stability Similarly to Section 4.4, the SRPT-ECF algorithm is stable: the order in which messages are considered will stay the same over time. This is because the Isolated Remaining Completion Time of each message will reduce faster for higher-priority messages: therefore, their order is not modified. This stability property means that messages have *no regrets* about their past actions, and avoids *priority inversions* that would be harmful for message completion times.

4.6 Running SRPT-ECF online

In Section 4.5, we assumed that all messages were available at the same time. In practice, however, the application may decide to create and transmit a new message at any time: this becomes an *online scheduling problem* where each message m is *released* at a given date r_m. The scheduler needs to take decision without knowing future release dates, potentially leading to poor decisions in hindsight. In this context, the *completion time* of a message is defined relatively to its release date, i.e. the completion time equals $C_m = f_m - r_m$ where f_m is the finishing date at which the message is completely received.

The SRPT-ECF algorithm can still be used, but it is no longer optimal: an offline algorithm would know about future messages and could anticipate and produce a slightly better schedule. Still, I show that SRPT-ECF remains desirable.

4.6.1 Online SRPT-ECF algorithm

In an online setting, the algorithm still works as described in Section 4.5: 1) it orders messages according to their *Isolated Remaining Completion Time*, that is: assuming we dedicate all paths resources to a message, how much time would it need to complete? 2) it allocates paths to messages with ECF. The main difference is that the *Isolated Remaining Completion Time* is more complicated to compute.

Assume that we already have a set of messages $(m_k)_{1 \leq k \leq n-1}$ already being scheduled and transmitted with SRPT-ECF. Then, at time r_n, a new message m_n is released and wants to be scheduled: the current schedule may need to be changed to satisfy the requirements of SRPT. We need to recompute and compare the *Isolated Remaining Completion Time* for each message. A key difference with Section 4.5 is that part of a message may already be in-flight, possibly impacting its remaining completion time. This is especially the case when a data-chunk is in-flight on a high-latency path: even if all future data-chunks use a low-latency path, we still need to wait for this in-flight data-chunk to arrive. In practice, the Isolated Remaining Completion Time of a message m_k with size S_k is the maximum of two terms: the optimal completion time given by ECF (which only depends on the size S_k) and the time $T_{\text{in-flight}}$ needed for existing in-flight data-chunks to reach the destination:

$$C_{\text{isolated}}(m_k) = \max\left[C^*_{\text{ECF}}(S_k), T_{\text{in-flight}}(m_k)\right] \tag{4.5}$$

This second term can be computed by predicting the date at which in-flight data-chunks will finish on all paths. A simple way to do this is to record the last time at which we sent a data-chunk on a path p and add the path delay D_p, assuming it does not change over time, and take the maximum over all paths. Once the Isolated Remaining Completion Time is computed for all messages, we order messages from smaller to bigger and use ECF to allocate paths to messages, as before.

Thanks to the stability property, we don't need to recompute C_{isolated} for all messages, since the relative order of existing messages is unchanged: the new message can simply inserted in the ordered list of messages, which can be done with only $O(log(n))$ operations.

Fig. 4.9 shows an online example of SRPT-ECF where a big message is pre-empted by two small messages in succession.

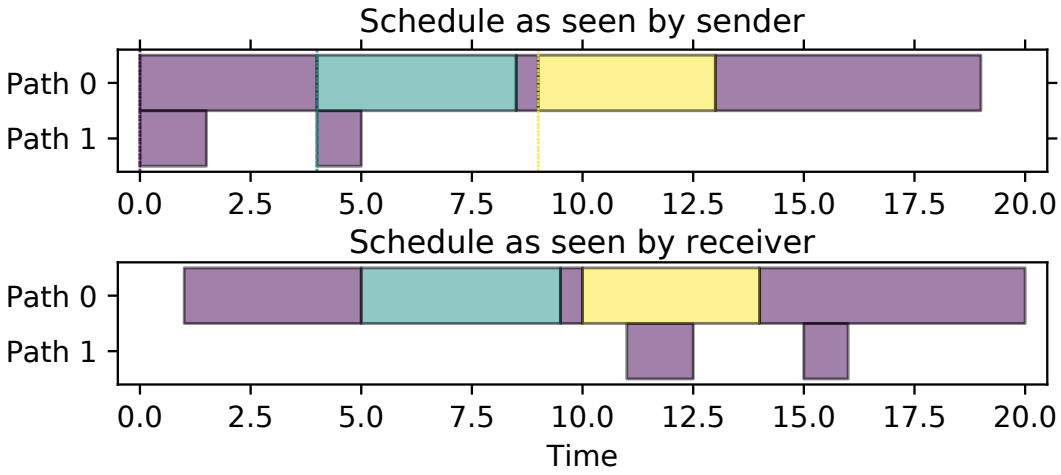

Figure 4.9 Example of SRPT-ECF running online. Message 0 is initially alone and uses ECF as in Fig. 4.5. When Message 1 arrives ($t = 4$), Message 0 gets preempted immediately: as a result, it starts using Path 1 again so that data on both paths finishes simultaneously, accounting for Message 1. When Message 2 arrives ($t = 9$), Message 0 is preempted again but this time it is too late to use Path 1, so it just waits for Message 2 to finish.

4.6.2 Comparison with offline algorithms

To understand how much SRPT-ECF can lose by not anticipating future events, I compare it to *offline scheduling algorithms* that know about future message releases and are able to anticipate. The goal is to illustrate the behavior of SRPT-ECF, rather than conducting a thorough analysis of offline-to-online performance ratio.

I implement a simple offline algorithm that tests all possible priority orderings between messages. For each priority ordering, it allocates resources to messages in this order while ensuring simultaneous completion on all paths, giving resources to a lower-priority message if a higher-priority message has not yet been released. The algorithm then selects the ordering that yields the best value of the chosen metric, which could be for instance the average completion time or the maximum completion time.

In Fig. 4.10 I show the result of this offline algorithm in the same setting as Fig. 4.9, taking the average completion time as metric to minimize. For Message 0, the offline algorithm does 18% better than SRPT-ECF. This example illustrates that SRPT-ECF works best when messages are available in batches; in addition, the application should inform the scheduler as soon as possible about data it plans to send, even if such data is not yet available.

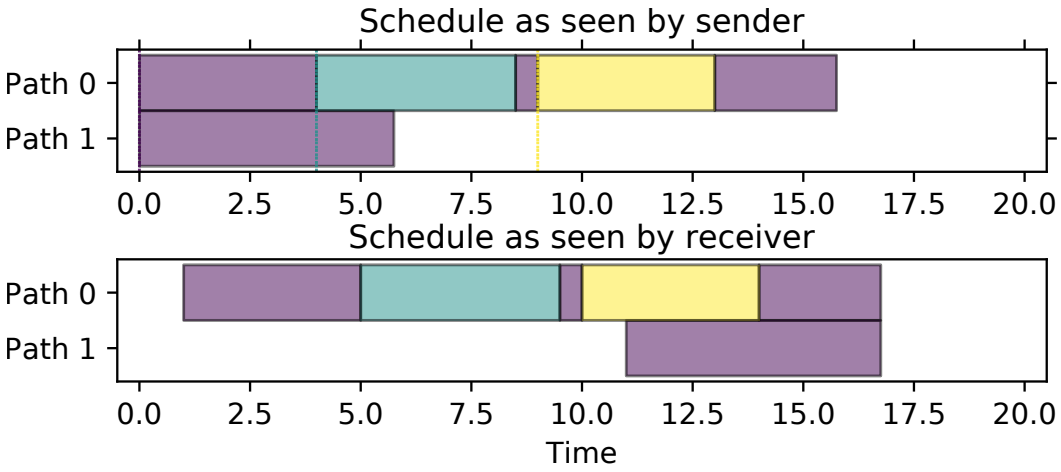

Figure 4.10 Optimal schedule obtained by an offline algorithm that minimizes the average completion time. Message 0 can anticipate the arrivals of Messages 1 and 2, so it uses Path 1 for a longer time to ensure both paths finish at the same time.

4.7 Trace-based evaluation of SRPT-ECF

To evaluate the SRPT-ECF algorithm against other scheduling algorithms, I use a trace-based approach.

4.7.1 Methodology

I use `webpagetest.org` to run an actual web browser and instrument it when loading a web page with HTTP/2 on a single path. The tool emulates a low-end residential connection (DSL with 1.5 Mbps download capacity and 50 ms of additional RTT). For each web resource, I record the date of the request and the size of the response. I then select a single connection from this trace and replay it in the multipath model described in Section 4.4: for each web resource, I create a message of the same size in a multipath connection from the server to the client; the release date r_m is the date of the original request. I use two paths with comparable combined capacity and end-to-end latency compared to the original setup: a first path with 80% of the capacity and 80% of the end-to-end latency (DSL), and a second path with 40% of the capacity and 180% of the end-to-end latency (mobile). Note that I don't compare the results with the original trace because it would be meaningless: I simply compare different schedulers on the same trace replay.

I take several steps to ensure the trace is meaningful:

1. I ensure that data is cached on the server by repeating the experiment and keeping the last run: we can thus assume that the server can send the content as soon as it receives the client request;

2. when recording the trace, I emulate a slow DSL connection to make sure the page load is network-bound and not CPU-bound, and also to have many resources loading simultaneously;

3. when replaying the trace, I simulate a slightly faster overall network (120% of the original capacity) to respect dependencies without having to enforce them explicitly. Indeed, each web page has an implicit dependency graph [108], but instead of trying to determine this graph, my method makes sure that any prerequisite requests are finished by the time a new request is made. Said differently, any dependency in the original web page produces two independent groups of requests in the replayed trace.

4.7.2 Simulation code

The simulation code is based on discrete event simulation with the help of the Salabim Python library [38]. I implemented the simple network model from Section 4.4.1 with a continuous transmission model. Instead of simulating packets, the scheduler can transmit data continuously during a time interval, and each byte of data is subject to the RTT of the path before being acknowledged.

The code of this simulator is available[3][4]. I describe a few aspects of this simulator below:

discrete event core This is located in `discrete_event.py`. It contains the main loop of the simulation in the `process()` method.

schedulers implementation All scheduling algorithms or "strategies" are located in `scheduling.py`. A scheduling strategy must define a function that will be called whenever a path is free and at least a stream has data to send, or on special occasions (such as when a new stream is created). This function is given the list of streams with data to send, the list of available paths, the list of all paths (including those that are already busy sending data), and the current

[3] `https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/multipathsim/-/tree/master/model`

[4] An earlier packet-based simulator with a graphical visualisation can also be found in the same repository: `https://gricad-gitlab.univ-grenoble-alpes.fr/jonglezb/multipathsim/-/tree/master/simulation`. However, this earlier simulator was only used to obtain initial intuition: no result in this work is based on it, and I will not describe it further.

time. The scheduling strategy must return a triple with: the stream that should be served, the path on which data from this stream should be sent, and the duration for which this action is valid. Specifying a duration is necessary because of the continuous network model, and because the scheduling strategy may want to stop using a path after some time (this is the case for ECF). The scheduling strategy can also choose to return nothing, which means that it is not interested in using the available paths: again, this is needed for ECF.

input instances The input is specified in a simple text format, documented in the `README`. It describes the available paths, each with a fixed "speed" and "delay". It also describes the available streams, each with a release date (allowing online experiments) and a total size. It is possible to specify dependencies between streams but it was not used for this thesis.

HTTP/2 trace conversion For the trace-based evaluation, I needed to convert the HTTP/2 trace from `webpagetest.org` in the input format expected by my tools. This is done in `trace_to_model.py` and works from the JSON trace produced by `webpagetest.org`. It produces several files, one for each connection recorded in the trace.

offline solver for the needs of Section 4.6.2, I implemented offline solvers in `solve_omniscient.py`. These solvers are not based on discrete event simulation, but they read the same input format and the result can be plugged into the same graphical visualisation.

4.7.3 Results

Figure 4.11 shows the CDF of message completion times with SRPT-ECF and SA-ECF [87] when loading the 132 images of a wikipedia page. The Weighted Round-Robin component of SA-ECF is implemented with the same weight for all messages. I also include two simpler schedulers based on MinRTT: First-Come-First-Serve (FCFS-minRTT) that schedules messages in their order of arrival, and SRPT-minRTT that orders messages by size. This experiment yields two main results:

1. ECF exploits the low-latency path to reduce completion time for small messages, while MinRTT does not; the poor performance of MinRTT can be partially explained by the network model, see Section 4.8.2;

2. SRPT brings a very significant improvement for the tail of resources compared to both Round-Robin (SA-ECF) and FCFS.

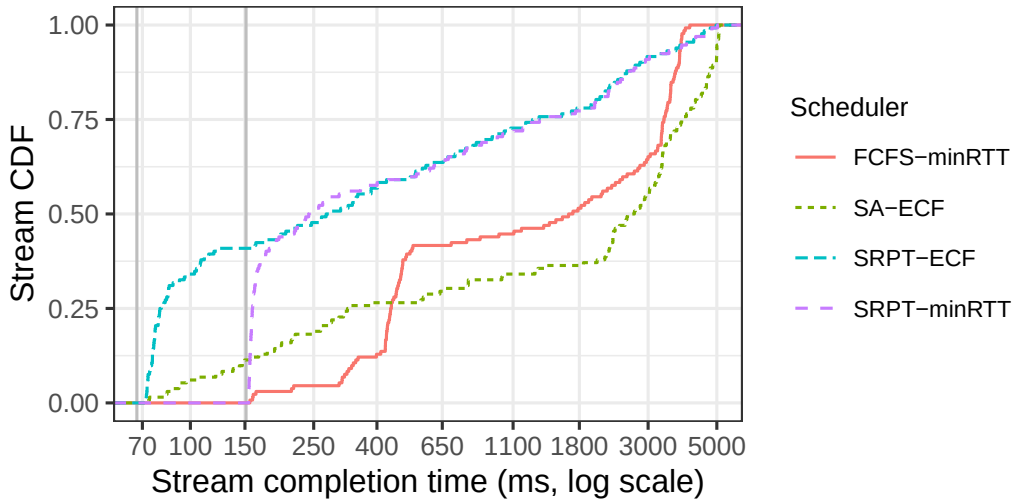

Figure 4.11 Simulation of a wikipedia page load by replaying a trace (`webpagetest.org` ID `200413_FH_478b18e178c0fbf2ec9312686630e510`, run 3, connection 2). Path 0: 1050 Kbit/s, 67 ms. Path 1: 750 Kbit/s, 151 ms. Path latency is indicated with the vertical bars. SRPT-ECF exhibits low completion times, thanks to its combination of ECF (left part of the CDF) and SRPT (tail of the CDF).

To better understand this second effect, I plot the number of active messages at each instant in Figure 4.12. SRPT-based schedulers reduce the number of active messages as much as possible by prioritizing small messages, thereby completing them sooner. With FCFS, small messages suffer when they have to wait behind a larger message, increasing both the backlog of active messages and the completion time. SA-ECF accumulates even more work because of Round-Robin, but smaller messages do not have to wait for large messages to finish, yielding better completion time for small messages compared to FCFS. Overall, combining SRPT and ECF gives the best results.

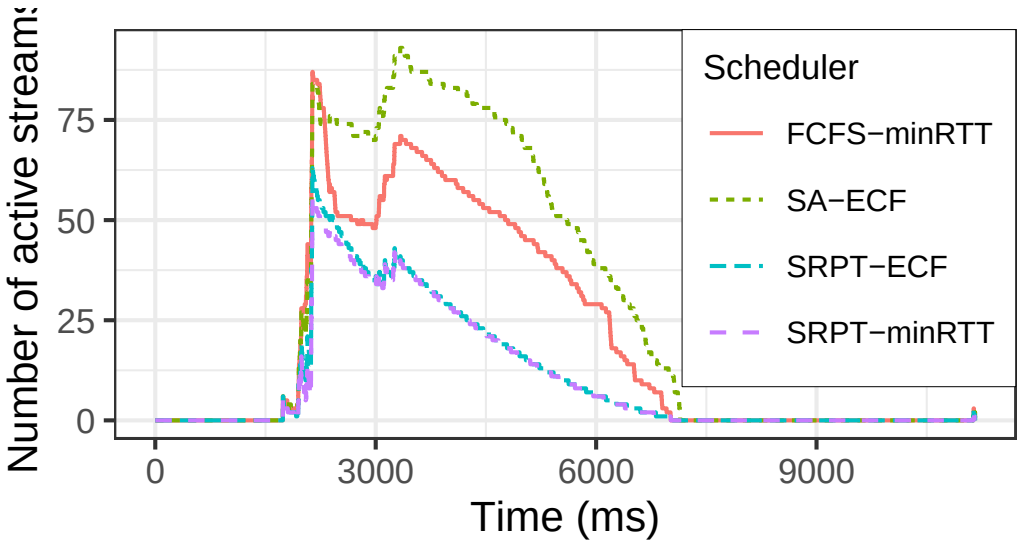

Figure 4.12 Scheduler occupation over time during the trace replay. Messages are created in three visible bursts. The schedulers have different strategies for handling the backlog of active messages.

4.8 Practical considerations

I now discuss several practical aspects of the SRPT-ECF algorithm and how it could be implemented in practice.

4.8.1 Dealing with network variability and uncertainty

All results so far are based on the simple network model from Section 4.4. In real networks, the latency and available capacity can vary significantly over time, and this can have a large impact on the effectiveness of scheduling algorithms. I sketch some ways in which SRPT-ECF could be adapted to cope with this issue.

The main problem is the uncertainty associated with measured latency and capacity: past measurements may not be good predictors of future network conditions. In addition, it takes one RTT for the sender to become aware of a significant change in the network. The key challenge is to anticipate these unexpected changes while keeping the associated cost low.

On a theoretical side, it will be difficult to ensure the *stability* property from Section 4.5.2 under changing network conditions. Thus, the scheduler should expect to take decisions that may partially contradict earlier decisions.

On a practical side, the DEMS scheduler [36] introduced several interesting techniques for MPTCP: using the one-way delay difference between paths instead of the RTT, controlled data duplication at the end of a message, adaptation based on the variance of latency samples. Controlled duplication is interesting because it provides optimal completion time at the end of a message, even if the delay estimation used by ECF was wrong. These techniques could be adapted to apply to a SRPT-ECF implementation, although there are challenges to be solved: should each message benefit from duplication, which means delaying subsequent messages? Or should duplication only be applied to the last message when there is nothing left to transmit.

4.8.2 Congestion control: pacing vs. congestion window

I discuss two approaches to sending congestion-controlled data (classical congestion window and pacing) and their impact on scheduling.

Pacing

Pacing is being used by recent TCP congestion control algorithms such as BBR [13]. These algorithms "pace" outgoing packets at the estimated bottleneck capacity to avoid sending large bursts of data at once when the congestion window is not full.

The network model, with its continuous model of transmission and its assumption of a perfect congestion control, is a perfect *pacing* model: between time t and $t + dt$, a sender can transmit $B_p \cdot dt$ bits of data on path p where B_p is the capacity of the path. This infinitesimal amount of data will experience a delay of D_p before reaching the remote peer.

Classical congestion window

When using a classical congestion window mechanism, the steady-state is reached when the congestion window is full: the connection enters a "ACK pacing" phase in which new packets are sent for each incoming ACK, naturally pacing packets at the estimated bottleneck capacity.

When the congestion window is not full, the sender can send several packets in a burst. These packets will accumulate in the bottleneck buffer, and will be released from the buffer at a rate equal to the bottleneck capacity. In the end, the delay of

each individual packet is the same as in the "pacing" case (assuming the buffer does not overflow): packets are waiting in the bottleneck buffer instead of being paced by the sender.

Pacing and scheduling

In the context of multipath scheduling, pacing has an additional benefit: it allows to schedule data as late as possible, thus better adapting to real-time changes in network conditions. Essentially, pacing converts *queuing delay* at the bottleneck into *waiting delay* at the sender. The advantage is that the sender has more control over data still waiting in its sending buffer than data stuck in the bottleneck buffer. This provides better *reactivity*, a concept described in Section 2.3.2 of Chapter 2.

However, pacing interacts poorly with MinRTT. A naive implementation of MinRTT will be blocked after sending just one packet on the low-latency path, thinking that it is unavailable because its congestion window is filled. As a result, it will start sending on higher-latency paths even though it would have been beneficial to wait and send all data on the lowest-latency path. This partially explains the poor performance of MinRTT in the evaluation in Section 4.7: since the model implements a perfect pacing mechanism, MinRTT never has the opportunity to send whole messages on the low-latency path. As a result, all messages bear the cost of the high-latency path.

A real MinRTT implementation could take pacing into account and estimate when it should wait and send all data on the lowest-latency path. However, it comes at the expense of more complexity and deviates from the simple stateless idea of MinRTT.

4.8.3 Buffering strategy

The buffering strategy of the sender can have an important impact on performance and reactivity. To ensure good reactivity, it is necessary to serialize data as late as possible before it is actually transmitted on the network (see Section 2.3.2 of Chapter 2). However, in a multipath situation, this becomes even more important: changing network conditions can frequently cause the sender to change its scheduling decisions and use another path. Such a scheduling change needs to be applied quickly, without leaving buffered data that would be sent according to the old scheduling decision.

4.8.4 Streaming use-cases and infinite messages

In the scheduling model (Section 4.2), I assumed that the scheduler knows about the total message size in advance. In practice, this may not always be possible: a HTTP resource can be produced "on-the-fly", i.e. data starts being sent before the response has been fully computed. Similarly, a reverse proxy might fetch and forward data from an upstream server without knowing the total size in advance. Buffering the whole message before starting to send data to the client would be counter-productive.

This situation can be modelled with an *infinite message size*. The behaviour of SRPT-ECF when faced with a message of infinite size is the following:

Stream scheduling (SRPT) within a priority class, an infinite message has lower priority than any other finite message. This is consistent with the regular behaviour of SRPT-ECF, because the Isolated Remaining Completion Time of an infinite message is infinite. However, if several messages with infinite size need to be ordered, the algorithm is modified to serve them with Round-Robin.

Path allocation (ECF) an infinite message is allocated to all paths. This is consistent with the regular behaviour of SRPT-ECF, because an infinite message is larger than any threshold (Section 4.4.2).

When the message is almost finished, it needs to inform the scheduler about its remaining size. At this point, SRPT-ECF can resume its original behaviour because it now has access to the message size. It is important to know the message size before the very end of the message, because SRPT-ECF needs the size to bring the most benefits near the end of messages: SRPT prioritizes small messages, and ECF tries to ensure their simultaneous completion on all paths. Theoretically, this avoids having regrets.

4.9 Conclusion

I have presented a scheduling model that allows to design and analyse stream-aware multipath schedulers. To obtain good performance, a scheduler needs to work on both the **stream scheduling** aspect (made possible by multiplexing) and on the **path allocation** aspect (the more classical multipath scheduling problem). Combining the two aspects opens many opportunities for optimisation but also means that a large number of different approaches are possible. To my knowledge, these two

sub-problems had not been clearly defined before, although scheduling algorithms that implicitly solve these sub-problems together were proposed recently [87, 23].

Within this model, I proposed a new stream-aware multipath scheduler called SRPT-ECF, where a variant of SRPT (Shortest Remaining Processing Time) is used for stream scheduling and ECF is used for path allocation. Despite the apparent simplicity of the algorithm, the combination of SRPT and ECF achieves an optimal pattern of completion times. This brings highly desirable properties such as allowing very different types of streams to coexist peacefully: small or important messages enjoy strict priority to use the lowest-latency paths, while bulk transfers can still use other higher-latency paths to make progress. This has the additional advantage of lowering the risk of starving large flows, because they can exploit unused resources offered by the multipath setup. All these properties can be ensured without any manual tuning or classification from the application: it just needs to provide expected message sizes to the scheduler. That being said, the application can still provide strict priorities to the scheduler to indicate that a given flow is important even though it may be larger.

Evaluation on a HTTP/2 trace shows that SRPT-ECF yields much better message completion time than other stream-aware schedulers. In particular, Round-Robin stream scheduling stands out for behaving poorly: since it tries to handle all streams concurrently, it accumulates a large number of concurrent streams and this delays the completion time of all streams. Said differently, SRPT allows streams to cooperate with each other: a stream can tolerate short-term unfairness and starvation if this allows another stream to achieve better results. In contrast, Round-Robin provides short-term fairness, but this comes at the cost of higher completion time for all streams.

In fact, Round-Robin could make sense in two situations:

- for truly concurrent flows, for instance if they belong to different users. User A would find it unfair if all resources are first given to User B, and would prefer sharing resources with B using Round-Robin, even though the overall completion time of User A is the same in both situations. The short-term fairness offered by Round-Robin also avoids starvation, which is important in a competitive setting;

- for “streaming” use-cases, where any byte of received data can be exploited by the application. In that case, the overall completion time is not very important and short-term fairness matters more, even for cooperative flows.

But in the message-based multiplexing model considered here, all flows belong to the same user application and are atomic: short-term fairness is not relevant and the overall completion time of a message is the appropriate metric. As such, Round-Robin yields very poor performance in this situation. This fact has been only recently realized for HTTP/2 [110, 71].

Overall, I expect this work to lead to better MPQUIC schedulers, especially in the context of HTTP/3. SRPT-ECF itself is a first step, and will probably need some adaptation and extensions to be deployable in real networks.

5 Conclusion

This work explored several directions to reduce latency in the Internet. In my early work on adaptive routing [52], my approach was to let routers discover the best paths that minimize end-to-end latency. Indeed, routers are naturally in a position to have several available paths to select from when taking routing decisions. However, it turned out to be difficult to implement in practice, because routers have a poor view of end-to-end performance metrics such as latency. I implemented a proof-of-concept router based on OpenFlow that performs passive RTT measurements by observing timestamps in TCP packets, but this approach would not scale very well and is very specific to TCP.

After this initial work, I turned my focus on *end-to-end mechanisms* to improve latency, that is, working on the end-hosts themselves. This matches the end-to-end model of the Internet where most of the intelligence is supposed to be located at the periphery of the network.

The first end-to-end mechanism that I studied is *multiplexing* in Chapter 2. Multiplexing allows several messages or data flows to share a common connection, thereby sharing information about the current network state: latency, loss events... It is a first example of *useful cooperation* between data flows.

In Chapter 3, I applied this multiplexing idea to DNS. With the current DNS-over-UDP approach, each DNS request and response starts from scratch without having any information about the network state. As a result, the retransmission timeout used to detect lost messages is extremely high, ranging from 1 to 5 seconds depending on the implementation. By multiplexing several DNS requests on a single persistent TCP connection, more information can be obtained on the network state, and it can be reused for future requests. I showed that this DNS-over-TCP technique can significantly reduce tail latency in case of packet loss: in one experiment, the 99th percentile of query latency was reduced from 3022 ms to 109 ms. Multiplexing DNS requests on a persistent connection had been proposed before as "Connection-oriented DNS" [114]. However, to my knowledge, Chapter 3 is the first work to give specific evidence that multiplexing DNS requests on a persistent connection can improve latency in case of packet loss. More precisely, I isolated specific mechanisms such as Fast Retransmit that explain this latency improvement. Lastly, I

ran experiments to ensure that the DNS infrastructure could withstand the additional load of widespread DNS-over-TCP or DNS-over-TLS usage.

This was a first example that demonstrates the benefits of cooperation between data flows: information and measurements can be shared to better adjust to network conditions and improve performance. However, cooperation sometimes has drawbacks. In the case of DNS-over-TCP, this comes in the form of *Head-of-Line blocking*: since TCP ensures in-order delivery, a lost query will block subsequent queries from being processed, even they have been successfully received. Solving Head-of-Line blocking requires more advanced transport protocols, such as SCTP or QUIC, that are aware of the multiplexed nature of the data they transport. This way, they can ensure in-order delivery only when it makes sense and avoid unnecessary head-of-line blocking in the other cases.

In Chapter 4, I generalize this multiplexing idea by adding a new dimension: the ability to use several network paths. Multipath transport protocols, in particular MPTCP, have been extensively studied for more than ten years. However, MPTCP is based on the same single-stream model as TCP. Transport protocol extensions such as CMT-SCTP [49] and more recently MPQUIC [19, 105] allow to multiplex several streams of data in the same multipath connection. This has the potential to solve long-standing issues with MPTCP such as head-of-line blocking caused by asymmetric delays on the available paths, or the fact that MPTCP has no visibility on multiplexed data — for instance when HTTP/2 multiplexes several streams on a single (MP)TCP flow.

From a high-level perspective, combining the ability to multiplex several data streams *and* the ability to transmit data on several paths yields a very rich and interesting scheduling problem. Indeed, such a "stream-aware" scheduler has more information available compared to a MPTCP scheduler: it has visibility on how many streams have data to send, how much data is queued for each stream, and so on. It can also take different scheduling decisions for different streams. Until recently, this rich scheduling problem had not been identified clearly: the MPQUIC design proposals [19, 105] were focused on other design aspects and simply reused existing MPTCP schedulers, without exploiting the additional information made available to the scheduler. To my knowledge, the first proposals that identified this scheduling problem as such and developed schedulers to take advantage of this new paradigm were published in 2018: the "Dynamic path- and Stream-based Scheduling" proposal for CMT-SCTP [23] and the "Stream-Aware Earliest Completion First" scheduler for MPQUIC [87].

Chapter 4 focuses on this scheduling problem that I call the "stream-aware multipath scheduling problem" following the terminology from [87]. It introduces a scheduling model that can be used to reason about stream-aware multipath scheduler. Using this model, I then propose a new stream-aware multipath scheduler called SRPT-ECF. This scheduler exploits the size of each stream, and possibly priorities set by the application, to compute an appropriate order in which streams should be processed. It then ensures that each stream gets delivered in the smallest possible amount of time, exploiting multiple paths if it helps. Using a simplified model, SRPT-ECF yields an optimal sequence of stream completion time.

So far, this scheduling work remains rather theoretical. While I am confident that SRPT-ECF can be implemented as part of a MPQUIC or CMT-SCTP implementation, this remains to be proven. In particular, it would need more work to account for the unpredictable conditions found in real networks: this calls for both theoretical and practical extensions.

5.1 Perspectives

5.1.1 Dealing with measurement uncertainties

Most of the work in this thesis was based on predictable network conditions, either because the experimental conditions were controlled on a testbed, or because I selected a simple network model for the sake of tractability. Real networks are much less predictable, and this is especially true for a distributed network like the Internet. Network conditions can change over a large timescale (routing changes, diurnal traffic patterns, persistent congestion, outages) as well as over a small timescale (physical layer noise and interference, link rate adaptation, episodic congestion, synchronisation between senders, bursts of cross-traffic). In addition, because of network latency, any measurement such as RTT or loss rate is necessarily outdated by the time its result is known.

As a result, when decisions need to be taken based on the current network state, sub-optimal decisions become unavoidable. One specific example is the "remaining size" threshold that I compute in Section 4.4: it is used by ECF to decide on which path it will send data, and it depends on estimated network conditions. If the value of this threshold is wrong because of incorrect network measurements, then the completion time of the flow will be increased. Practical counter-measures have to

be found to overcome this uncertainty, such as the controlled replication scheme introduced by the DEMS scheduler for MPTCP [36].

The lack of reliable up-to-date measurements affects thin-stream communication the most, because of the low amount of traffic and the sporadic nature of this kind of communication. Only a small number of RTT measurements can be taken, and measurement results quickly become outdated.

This is also challenging on a theoretical level: few scheduling models integrate the notion of latency, because they are usually focused on processing time in the context of servers. I introduced models that account for latency (Sections 4.2 and 4.4.1 of Chapter 4), but many properties only work if the latency is fixed. Under uncertain network conditions, results such as the *stability* of a scheduler or the optimality of SRPT-ECF (Section 4.5.2 of Chapter 4) no longer hold. More advanced tools are needed, such as considering the expectation of parameters or minimizing a regret in hindsight. For instance, my earlier work on latency-sensitive routing [52] was resilient to outdated and noisy latency measurements by using an advanced optimization method from Game Theory. More work is needed in the context of stream-aware multipath scheduling to account for uncertainty in measurements.

5.1.2 More cooperation between thin-stream and bulk-transfer communications

We have seen that thin-stream communication is affected by its lack of reliable measurements. In addition, it is highly affected by packet loss because efficient recovery would need good RTT estimation and sustained traffic: thin streams lack both. This affects DNS (Chapter 3), TCP SYN packets [11], instant messaging traffic, signalisation traffic such as Radius...

Such loss-sensitive traffic could benefit from more cooperation with other types of communication patterns, such as bulk transfers. The idea of the cooperation is the following: bulk-transfer flows enjoy timely feedback from the network because they always have data to send, thus eliciting a continuous stream of acknowledgments. As such, they are able to detect a packet loss in just one RTT, and they have access to more accurate and up-to-date RTT measurements. Cooperation means that they could provide loss detection and accurate RTT measurements to the thin-stream flows.

Done correctly, this cooperation would significantly improve the performance and reactivity of thin-stream flows. However, it is also very challenging: bulk transfers

should not impair thin-stream flows, because that would nullify any advantage brought by the cooperation. Even with appropriate prioritization, we have seen that buffer management can have a significant impact on reactivity (see Section 2.3.2 in Chapter 2). In addition, bulk transfers might create congestion that will increase the delay experienced by thin-stream flows, both at the bottleneck and at the sender: congestion control might prevent the sender from transmitting new data from thin-stream flows. One solution could be to always reserve some space in the congestion control window for thin-stream data.

This cooperation could be implemented as flow multiplexing on a common connection, and this combination of flows could be scheduled with the scheduler proposed in Chapter 4. For instance, DNS traffic could share the same QUIC connection as HTTP traffic. This would introduce a large number of practical challenges though, such as: cross-layer dependencies, more complexity in the DNS infrastructure and increased concentration of power in the hands of a few actors. Nevertheless, such a scheme was already suggested in the past [40] although it was later retracted [41].

A limitation of the multiplexing technique studied so far is that all flows must be handled by the same application. It does not allow flows from different applications or from different hosts to be multiplexed and collaborate with each other.

Overcoming this limitation would allow broader cooperation between flows. There has been a proposal to use HTTP/3 to carry and demultiplex flows for several applications on the same host [94]. For even broader cooperation among flows from several devices, WAN aggregation can be used: this is detailed in the next section.

5.1.3 WAN aggregation

The basic idea of WAN aggregation, also sometimes called “multilink aggregation”, is to exploit several paths to the Internet directly from the home router.[1] This has similar advantages to using a multipath transport such as Multipath TCP, but it is simpler to deploy: any end-device connected to the home router will transparently benefit from the advantages of multipath connectivity.

The expected benefits of WAN aggregation are fault tolerance (which allows to tolerate a link failure by re-routing on another path), better reliability (to work around packet loss and congestion on a path), and bandwidth aggregation.

There are two main existing WAN aggregation approaches:

[1] I call it “home router” because it helps to understand the concept, but the concept can of course be applied to an “office router” at a business office.

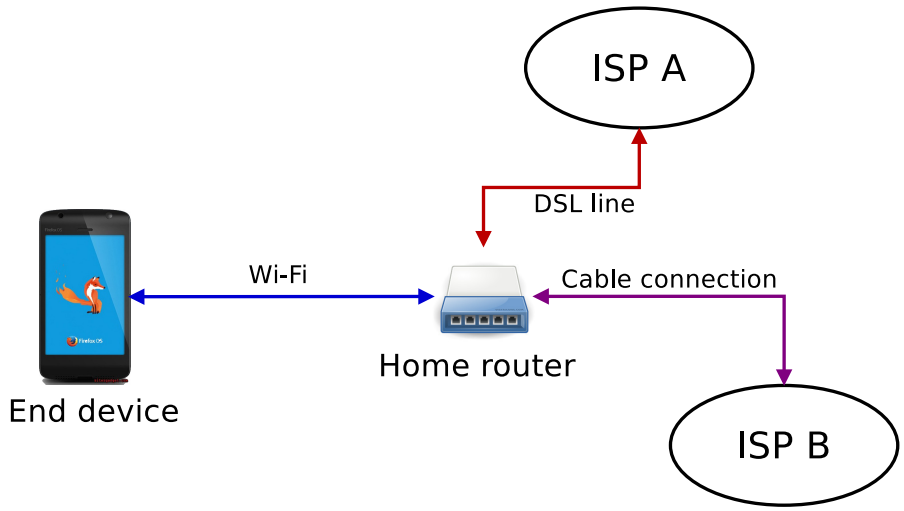

Figure 5.1 An example "network-based" multi-homing situation that is amenable to WAN aggregation. This aggregation will be transparent for the devices connected to the local network.

simple or "single-ended" The home router simply maps each outgoing connection to a single path. It can either map all connections to a single path and keep another path as a fallback, or it can load-balance connections on the available paths. There are several limitations: a given connection can only use a single path, and existing connections cannot be migrated to another path in case of path failure. The reason for that is that this approach often relies on NAT (Network Address Translation) to select the correct outgoing IP address before mapping a connection to a path. It has the advantage of not needing any server-side component.

tunneled or "dual-ended" The home router establishes one or several tunnels towards a server relay. The home router and the server relay can then cooperate to transmit user data across the available paths; traffic is then relayed by the server towards its destination on the Internet. It provides a very flexible and powerful system: for example, a single user connection can be mapped to several paths simultaneously to provide an increased throughput, data can be duplicated to provide extra reliability, error correcting codes can be added to avoid retransmission...

A popular open-source solution for this second approach is to setup a proxy on the home router that communicates with the relay server using Multipath TCP. Shadowsocks[2] is commonly used for the proxy part, with a setup that allows transparent

[2] `https://shadowsocks.org`

proxying of user connections. Other open-source solutions that are not based on Multipath TCP exist, such as Glorytun[3] and MLVPN[4].

A slight variation of the "dual-ended" approach is marketed as "SD-WAN" for business customers. It aims at offering a less expensive solution for the "WAN", that is, building a private network that interconnects several branch offices of the same business entity. Traditionally, a WAN is built with expensive dedicated lines managed with MPLS (Multi-Protocol Label Switching). SD-WAN solutions replace this architecture with a dual-ended WAN aggregation approach, typically tunneling all traffic through a central location to interconnect the branch offices. The main advantages are cost reduction and flexibility, because any network connection can be exploited (FTTH, xDSL, LTE) instead of using expensive dedicated links. However, reliability is generally much lower than with dedicated links, which is why it becomes useful to aggregate several connections to improve the overall reliability. In essence, two or more low-reliability links are used to replace a single high-reliability link, for a fraction of the price. It is a workable strategy thanks to a variant of the "power of two choices" [75]: it is unlikely that all links would fail or misbehave at the same time.

The SD-WAN market is well-identified and booming, with a large number of actors proposing SD-WAN solutions: Cisco Meraki, VMware (VeloCloud), Citrix, Oracle, CenturyLink, Aruba, Riverbed, Fortinet, Silver Peak, Peplink...

Improving WAN aggregation

In the dual-ended case, many flows from several devices are aggregated by the router into a single tunnel. This opens opportunities for cooperation between flows belonging to different devices: as developed above, it can benefit sensitive flows because they will recover faster, but it is also challenging because of Head-of-Line blocking and possible interference.

Current open-source approaches for WAN aggregation either schedule each packet individually or exploit Multipath TCP. Both approaches fail to consider individual streams, which can lead to sub-optimal scheduling and impair the latency experienced by thin-stream flows.

The SRPT-ECF scheduler that I proposed in Chapter 4 would be a good fit for this situation, since it is able to optimize the latency experienced by each multiplexed

[3] `https://github.com/angt/glorytun`
[4] `http://zehome.github.io/MLVPN/`

stream. Small sensitive flows would have priority and would use the lowest-latency path, while bulk transfers would get access to spare resources and exploit high-latency paths.

In practice, it could exploit a recent proposal for tunneling TCP connections within QUIC streams [86]. The home router would maintain a single MPQUIC connection to the relay server, would terminate TCP connections initiated by devices from the local network, and tunnel them into the MPQUIC connection. The MPQUIC scheduler would have visibility on each tunneled connection, and could use SRPT-ECF to optimally schedule them.

This approach raises a number of interesting challenges:

- Would SRPT-ECF scale to a very large number of flows in practice?
- Would SRPT-ECF be competitive compared to SD-WAN algorithms that explicitly classify flows? Some SD-WAN solutions implement rules that classify application traffic (Youtube videos, Voice-over-IP, gaming traffic...) and use this classification to decide how to schedule flows on the available paths. SRPT-ECF has the advantage of not requiring any classification, because it only looks at the flow size: this is simpler, more robust, and allows any new application to be deployed without needed an update of classification rules. However, it may under-perform compared to SD-WAN solutions with explicit classification rules.
- How should congestion control be handled? All combined flows from all devices, once tunneled into a single MPQUIC connection, would be subject to the congestion control rules of a single connection. If n tunneled flows are competing with a single external connection at the bottleneck, assuming fairness, the n flows would collectively only obtain half of the available capacity. Each flow would thus get a lower share of bottleneck capacity than if it was running outside of the MPQUIC tunnel. Solving this problem would require extensive modelization and experimentation with new congestion control algorithms.

5.1.4 Large-scale impact of multipath

Finally, we can envision the impact of a large-scale deployment of multipath technologies, possibly through WAN aggregation. What would be the resulting impact

on the Internet as a whole? Would it lead to a stable system, or would we witness large and persistent shifts of traffic as multipath protocols adjust to measured performance? Would there be a significant effect on congestion?

Several decades ago in ARPANET, the routing algorithm would choose routes based on measured latency [63]. It eventually had to be changed because it caused severe stability issues under high load. To some extent, widespread use of multipath would have a similar effect: it allows traffic to shift from one path to another based on latency and loss measurements. As such, it could also lead to large-scale instabilities.

Even if large-scale multipath is stable, it may still converge to undesirable situations where no user is happy. In Game Theory, a stable situation is called a Nash Equilibrium. A popular example of undesirable Nash Equilibrium in the context of transportation networks is the Braess paradox [33]: adding a new shortcut road to a network can worsen the travel time for all users. In the context of multipath communication networks, similar situations may exist: switching all users from single-path to multipath may decrease the performance obtained by each user, possibly because of increased cross-traffic and congestion, higher variability in delays...

Braess' paradox happens when each user selfishly optimizes its own performance metric. To move away from a Nash Equilibrium, either some users need to accept to lose some performance so that other users can gain, or some external constraints need to be enforced. Could large-scale cooperation between network users avoid undesirable Nash Equilibria, and if so, how could such a cooperation be achieved? For TCP, Kelly and others have shown that congestion control acts like an implicit coordination mechanisms between users, and that it can be designed to converge to a stable state with fair sharing of network capacity [59, 61]. This line of work on congestion control has been extended to Multipath TCP [60, 62, 111]. There may be other cooperation mechanisms besides congestion control.

Overall, the impact of local decisions on the global state of a network and the question of cooperation between users of a network are open-ended questions that can yield plenty of research questions. It can be approached with theoretical tools such as Game Theory, but it also needs a practical view of multipath protocols that make sense in the current and future Internet.

Bibliography

[1] Rachel Albert, Anjul Patney, David Luebke, and Joohwan Kim. "Latency Requirements for Foveated Rendering in Virtual Reality". In: *ACM Trans. Appl. Percept.* 14.4 (Sept. 2017). DOI: 10.1145/3127589. URL: https://doi-org.ins2i.bib.cnrs.fr/10.1145/3127589. | cit. on p. 2

[2] Stefan Alfredsson, Giacomo Del Giudice, Johan Garcia, et al. "Impact of TCP congestion control on bufferbloat in cellular networks". In: *2013 IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*. June 2013, pp. 1–7. DOI: 10.1109/WoWMoM.2013.6583408. | cit. on p. 1

[3] M. Allman, K. Avrachenkov, U. Ayesta, J. Blanton, and P. Hurtig. *Early Retransmit for TCP and Stream Control Transmission Protocol (SCTP)*. RFC 5827 (Experimental). RFC. Fremont, CA, USA: RFC Editor, May 2010. DOI: 10.17487/RFC5827. URL: https://www.rfc-editor.org/rfc/rfc5827.txt. | cit. on pp. 14, 41, 42, 43

[4] Fred Baker, Chris Bowers, and Jen Linkova. *Enterprise Multihoming using Provider-Assigned Addresses without Network Prefix Translation: Requirements and Solution*. Internet-Draft draft-ietf-rtgwg-enterprise-pa-multihoming-06. Work in Progress. Internet Engineering Task Force, May 2018. 48 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-rtgwg-enterprise-pa-multihoming-06. | cit. on p. 7

[5] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Ed. by IvanI. Ivanov, Marten Sinderen, Frank Leymann, and Tony Shan. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. DOI: 10.1007/978-3-319-04519-1_1. | cit. on pp. 8, 10, 48

[6] M. Belshe, R. Peon, and M. Thomson (Ed.) *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2015. DOI: 10.17487/RFC7540. URL: https://www.rfc-editor.org/rfc/rfc7540.txt. | cit. on pp. 21, 29, 60, 61, 62

[7] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*. RFC 1945 (Informational). RFC. Fremont, CA, USA: RFC Editor, May 1996. DOI: 10.17487/RFC1945. URL: https://www.rfc-editor.org/rfc/rfc1945.txt. | cit. on p. 58

[8] Mike Bishop. *Hypertext Transfer Protocol Version 3 (HTTP/3)*. Internet-Draft draft-ietf-quic-http-28. Work in Progress. Internet Engineering Task Force, May 2020. 70 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-28. | cit. on pp. 60, 62, 69

[9] Andrea Bittau, Daniel B. Giffin, Mark J. Handley, et al. *Cryptographic protection of TCP Streams (tcpcrypt)*. Internet-Draft draft-ietf-tcpinc-tcpcrypt-11. Work in Progress. Internet Engineering Task Force, Nov. 2017. 31 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-tcpinc-tcpcrypt-11. | cit. on p. 14

[10] Timm Böttger, Felix Cuadrado, Gianni Antichi, et al. "An Empirical Study of the Cost of DNS-over-HTTPS". In: *Proceedings of the Internet Measurement Conference*. IMC '19. Amsterdam, Netherlands: Association for Computing Machinery, Oct. 2019, pp. 15–21. DOI: 10.1145/3355369.3355575. URL: http://doi.org/10.1145/3355369.3355575 (visited on July 16, 2020). | cit. on p. 44

[11] Tristan Braud, Martin Heusse, and Andrzej Duda. "The Virtue of Gentleness: Improving Connection Response Times with SYN Priority Active Queue Management". In: *IFIP Networking 2018 Conference (IFIP Networking 2018)*. Zurich, Switzerland, 2018. URL: https://hal.archives-ouvertes.fr/hal-01797063. | cit. on pp. 13, 14, 98

[12] B. Briscoe, A. Brunstrom, A. Petlund, et al. "Reducing Internet Latency: A Survey of Techniques and Their Merits". In: *IEEE Communications Surveys Tutorials* 18.3 (2016), pp. 2149–2196. DOI: 10.1109/COMST.2014.2375213. | cit. on pp. 4, 13, 14, 24, 32

[13] Neal Cardwell, Yuchung Cheng, C Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-Based Congestion Control". In: *Queue* 14.5 (2016), p. 50. | cit. on p. 90

[14] Neal Cardwell, Yuchung Cheng, C. Stephen Gunn, Soheil Hassas Yeganeh, and Van Jacobson. "BBR: Congestion-based Congestion Control". In: *Commun. ACM* 60.2 (Jan. 2017), pp. 58–66. DOI: 10.1145/3009824. URL: http://doi.acm.org/10.1145/3009824. | cit. on pp. 1, 14

[15] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain. *TCP Fast Open*. RFC 7413 (Experimental). RFC. Fremont, CA, USA: RFC Editor, Dec. 2014. DOI: 10.17487/RFC7413. URL: https://www.rfc-editor.org/rfc/rfc7413.txt. | cit. on p. 13

[16] S. Cherry. "Edholm's law of bandwidth". In: *IEEE Spectrum* 41.7 (July 2004). Conference Name: IEEE Spectrum, pp. 58–60. DOI: 10.1109/MSPEC.2004.1309810. | cit. on p. 1

[17] T. Chown (Ed.), J. Arkko, A. Brandt, O. Troan, and J. Weil. *IPv6 Home Networking Architecture Principles*. RFC 7368 (Informational). RFC. Fremont, CA, USA: RFC Editor, Oct. 2014. DOI: 10.17487/RFC7368. URL: https://www.rfc-editor.org/rfc/rfc7368.txt. | cit. on p. 7

[18] Quentin De Coninck, Matthieu Baerts, Benjamin Hesmans, and Olivier Bonaventure. "A First Analysis of Multipath TCP on Smartphones". en. In: *SpringerLink*. Springer, Cham, Mar. 2016, pp. 57–69. DOI: 10.1007/978-3-319-30505-9_5. URL: https://link-springer-com.gaelnomade-1.grenet.fr/chapter/10.1007/978-3-319-30505-9_5 (visited on June 22, 2017). | cit. on p. 8

[19] Quentin De Coninck and Olivier Bonaventure. "Multipath QUIC: Design and Evaluation". In: *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 160–166. DOI: 10.1145/3143361.3143370. URL: http://doi.acm.org/10.1145/3143361.3143370. | cit. on pp. 71, 96

[20] Thomas Dreibholz, Robin Seggelmann, Michael Tüxen, and Erwin Paul Rathgeb. "Transmission scheduling optimizations for concurrent multipath transfer". In: *Proceedings of the 8th International Workshop on Protocols for Future, Large-Scale and Diverse Network Transports (PFLDNeT)*. Vol. 8. 2010. | cit. on pp. 75, 76

[21] Nandita Dukkipati, Neal Cardwell, Yuchung Cheng, and Matt Mathis. *Tail Loss Probe (TLP): An Algorithm for Fast Recovery of Tail Losses*. Internet-Draft draft-dukkipati-tcpm-tcp-loss-probe-01. Work in Progress. Internet Engineering Task Force, Feb. 2013. 20 pp. URL: https://datatracker.ietf.org/doc/html/draft-dukkipati-tcpm-tcp-loss-probe-01. | cit. on p. 43

[22] Eric Dumazet. *tcp: TCP NOTSENT LOWAT socket option*. Online; accessed on 20 August 2020. July 2013. URL: https://lwn.net/Articles/560082/. | cit. on p. 23

[23] Johan Eklund, Karl-Johan Grinnemo, and Anna Brunstrom. "Using multiple paths in SCTP to reduce latency for signaling traffic". en. In: *Computer Communications* 129 (Sept. 2018), pp. 184–196. DOI: 10.1016/j.comcom.2018.07.016. URL: http://www.sciencedirect.com/science/article/pii/S0140366417310873 (visited on Sept. 2, 2020). | cit. on pp. 75, 93, 96

[24] M. S. Elbamby, C. Perfecto, M. Bennis, and K. Doppler. "Toward Low-Latency and Ultra-Reliable Virtual Reality". In: *IEEE Network* 32.2 (2018), pp. 78–84. | cit. on p. 2

[25] Benevid Felix, Igor Steuck, Aldri Santos, Stefano Secci, and Michele Nogueira. "Redundant Packet Scheduling by Uncorrelated Paths in Heterogeneous Wireless Networks". In: *2018 IEEE Symposium on Computers and Communications (ISCC)*. ISSN: 1530-1346. June 2018, pp. 00498–00503. DOI: 10.1109/ISCC.2018.8538641. | cit. on p. 8

[26] S. Ferlin, O Alay, O. Mehani, and R. Boreli. "BLEST: Blocking estimation-based MPTCP scheduler for heterogeneous networks". In: *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. May 2016, pp. 431–439. DOI: 10.1109/IFIPNetworking.2016.7497206. | cit. on pp. 26, 28, 72

[27] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7230. URL: https://www.rfc-editor.org/rfc/rfc7230.txt. | cit. on pp. 16, 19, 20

[28] R. Fielding (Ed.) and J. Reschke (Ed.) *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2014. DOI: 10.17487/RFC7231. URL: https://www.rfc-editor.org/rfc/rfc7231.txt. | cit. on p. 19

[29]R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2068 (Proposed Standard). RFC. Obsoleted by RFC 2616. Fremont, CA, USA: RFC Editor, Jan. 1997. DOI: 10.17487/RFC2068. URL: https://www.rfc-editor.org/rfc/rfc2068.txt. | cit. on p. 59

[30]Tobias Flach, Nandita Dukkipati, Andreas Terzis, et al. "Reducing web latency: the virtue of gentle aggression". In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 159–170. | cit. on p. 14

[31]The Apache Software Foundation. *Apache HTTP Server Version 2.4 - Apache MPM event*. URL: https://httpd.apache.org/docs/2.4/mod/event.html (visited on June 2, 2020). | cit. on p. 58

[32]The Apache Software Foundation. *Overview of new features in Apache HTTP Server 2.4*. URL: https://httpd.apache.org/docs/2.4/new_features_2_4.html (visited on June 2, 2020). | cit. on p. 58

[33]Marguerite Frank. "The braess paradox". In: *Mathematical Programming* 20.1 (1981), pp. 283–302. | cit. on p. 103

[34]Alexander Frommgen, Tobias Erbshäußer, Alejandro Buchmann, Torsten Zimmermann, and Klaus Wehrle. "ReMP TCP: Low latency multipath TCP". In: *2016 IEEE International Conference on Communications (ICC)*. ISSN: 1938-1883. May 2016, pp. 1–7. DOI: 10.1109/ICC.2016.7510787. | cit. on p. 8

[35]K. J. Grinnem Grinnem, T. Andersson, and A. Brunstrom. "Performance Benefits of Avoiding Head-of-Line Blocking in SCTP". In: *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services - (icas-isns'05)*. Oct. 2005, pp. 44–44. DOI: 10.1109/ICAS-ICNS.2005.73. | cit. on p. 24

[36]Yihua Ethan Guo, Ashkan Nikravesh, Z. Morley Mao, Feng Qian, and Subhabrata Sen. "Accelerating Multipath Transport Through Balanced Subflow Completion". In: *Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking*. MobiCom '17. Snowbird, Utah, USA: Association for Computing Machinery, Oct. 2017, pp. 141–153. DOI: 10.1145/3117811.3117829. URL: http://doi.org/10.1145/3117811.3117829 (visited on Apr. 9, 2020). | cit. on pp. 76, 90, 98

[37]Sangtae Ha, Injong Rhee, and Lisong Xu. "CUBIC: A New TCP-friendly High-speed TCP Variant". In: *SIGOPS Oper. Syst. Rev.* 42.5 (July 2008), pp. 64–74. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105. | cit. on pp. 1, 14

[38]Ruud van der Ham. "salabim: discrete event simulation and animation in Python". In: *Journal of Open Source Software* 3.27 (2018), p. 767. | cit. on p. 86

[39]Mario Hock, Roland Bless, and Martina Zitterbart. "Experimental evaluation of BBR congestion control". In: *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. Oct. 2017, pp. 1–10. DOI: 10.1109/ICNP.2017.8117540. | cit. on p. 1

[40] Paul E. Hoffman. *Running DNS in Existing QUIC Connections*. Internet-Draft draft-hoffman-dns-in-existing-quic-00. Work in Progress. Internet Engineering Task Force, Apr. 2017. 6 pp. URL: https://datatracker.ietf.org/doc/html/draft-hoffman-dns-in-existing-quic-00. | cit. on p. 99

[41] Paul E. Hoffman. *Running DNS in Existing QUIC Connections*. Internet-Draft draft-hoffman-dns-in-existing-quic-01. Work in Progress. Internet Engineering Task Force, May 2017. 2 pp. URL: https://datatracker.ietf.org/doc/html/draft-hoffman-dns-in-existing-quic-01. | cit. on p. 99

[42] Toke Høiland-Jørgensen. "On the Bleeding Edge: Debloating Internet Access Networks". PhD thesis. Karlstad University Press, 2016. | cit. on pp. 5, 27

[43] Austin Hounsel, Kevin Borgolte, Paul Schmitt, Jordan Holland, and Nick Feamster. "Analyzing the Costs (and Benefits) of DNS, DoT, and DoH for the Modern Web". In: *Proceedings of the Applied Networking Research Workshop*. ANRW '19. Montreal, Quebec, Canada: Association for Computing Machinery, 2019, pp. 20–22. DOI: 10.1145/3340301.3341129. URL: https://doi.org/10.1145/3340301.3341129. | cit. on p. 44

[44] Austin Hounsel, Kevin Borgolte, Paul Schmitt, Jordan Holland, and Nick Feamster. "Comparing the Effects of DNS, DoT, and DoH on Web Performance". In: *Proceedings of The Web Conference 2020*. WWW '20. Taipei, Taiwan: Association for Computing Machinery, Apr. 2020, pp. 562–572. DOI: 10.1145/3366423.3380139. URL: http://doi.org/10.1145/3366423.3380139 (visited on July 16, 2020). | cit. on pp. 44, 45

[45] Christian Huitema, Melinda Shore, Allison Mankin, Sara Dickinson, and Jana Iyengar. *Specification of DNS over Dedicated QUIC Connections*. Internet-Draft draft-huitema-quic-dnsoquic-07. Work in Progress. Internet Engineering Task Force, Jan. 2018. 19 pp. URL: https://datatracker.ietf.org/doc/html/draft-huitema-quic-dnsoquic-07. | cit. on pp. 43, 55

[46] Per Hurtig, Karl-Johan Grinnemo, Anna Brunstrom, et al. "Low-Latency Scheduling in MPTCP". In: *IEEE/ACM Transactions on Networking* 27.1 (Feb. 2019). Conference Name: IEEE/ACM Transactions on Networking, pp. 302–315. DOI: 10.1109/TNET.2018.2884791. | cit. on p. 28

[47] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, and Takahiro Hirofuchi. "Using the EXECO Toolkit to Perform Automatic and Reproducible Cloud Experiments". In: *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 02*. CLOUDCOM '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 158–163. DOI: 10.1109/CloudCom.2013.119. URL: http://dx.doi.org/10.1109/CloudCom.2013.119. | cit. on p. 48

[48] Nuruddeen Iya, Nicolas Kuhn, Fabio Verdicchio, and Gorry Fairhurst. "Analyzing the impact of bufferbloat on latency-sensitive applications". In: *2015 IEEE International Conference on Communications (ICC)*. ISSN: 1938-1883. June 2015, pp. 6098–6103. DOI: 10.1109/ICC.2015.7249294. | cit. on p. 1

[49]J. R. Iyengar, P. D. Amer, and R. Stewart. “Concurrent Multipath Transfer Using SCTP Multihoming Over Independent End-to-End Paths”. In: *IEEE/ACM Transactions on Networking* 14.5 (Oct. 2006), pp. 951–964. DOI: 10.1109/TNET.2006.882843. | cit. on pp. 6, 96

[50]Jana Iyengar and Martin Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. Internet-Draft draft-ietf-quic-transport-24. Work in Progress. Internet Engineering Task Force, Nov. 2019. 156 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-24. | cit. on pp. 19, 64, 67

[51]Nick Jones. *NGINX structural enhancements for HTTP/2 performance*. https://blog.cloudflare.com/nginx-structural-enhancements-for-http-2-performance/. Online; accessed on 20 August 2020. May 2019. URL: https://blog.cloudflare.com/nginx-structural-enhancements-for-http-2-performance/. | cit. on p. 23

[52]B. Jonglez and B. Gaujal. “Distributed and Adaptive Routing Based on Game Theory”. In: *2017 29th International Teletraffic Congress (ITC 29)*. Vol. 1. Sept. 2017, pp. 1–9. DOI: 10.23919/ITC.2017.8064333. | cit. on pp. 9, 95, 98

[53]Baptiste Jonglez, Sinan Birbalta, and Martin Heusse. *Improving end-to-end delay at the application layer*. International Summer School on Latency Control for Internet of Services. Poster. June 2017. URL: https://hal.inria.fr/hal-01632191. | cit. on p. 11

[54]Baptiste Jonglez, Sinan Birbalta, and Martin Heusse. “Poster: persistent DNS connections for improved performance”. In: *2019 IFIP Networking Conference, Networking 2019, Warsaw, Poland, May 20-22, 2019*. IEEE, 2019, pp. 1–2. DOI: 10.23919/IFIPNetworking46909.2019.8999394. | cit. on p. 11

[55]Baptiste Jonglez and Bruno Gaujal. *Distributed Adaptive Routing in Communication Networks*. en. report. Inria ; Univ. Grenoble Alpes, Oct. 2016, p. 25. URL: https://hal.inria.fr/hal-01386832. | cit. on p. 10

[56]Baptiste Jonglez and Bruno Gaujal. “Distributed and Adaptive Routing Based on Game Theory”. In: *ALGOTEL 2017 - 19èmes Rencontres Francophones sur les Aspects Algorithmiques des Télécommunications*. Quiberon, France, May 2017. URL: https://hal.archives-ouvertes.fr/hal-01517911. | cit. on p. 10

[57]Baptiste Jonglez, Martin Heusse, and Bruno Gaujal. “SRPT-ECF: challenging Round-Robin for stream-aware multipath scheduling”. In: *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*. IEEE, 2020, pp. 719–724. URL: https://ieeexplore.ieee.org/document/9142713. | cit. on p. 12

[58]J. Jung, A. W. Berger, and Hari Balakrishnan. “Modeling TTL-based Internet caches”. In: *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No.03CH37428)*. Vol. 1. Mar. 2003, 417–426 vol.1. DOI: 10.1109/INFCOM.2003.1208693. | cit. on p. 53

[59]F P Kelly, A K Maulloo, and D K H Tan. “Rate control for communication networks: shadow prices, proportional fairness and stability”. en. In: *Journal of the Operational Research Society* 49.3 (Mar. 1998), pp. 237–252. DOI: 10.1057/palgrave.jors.2600523. URL: https://doi.org/10.1057/palgrave.jors.2600523 (visited on Sept. 2, 2020). | cit. on p. 103

[60]Frank P. Kelly and Thomas Voice. “Stability of end-to-end algorithms for joint routing and rate control”. In: *Computer Communication Review* 35.2 (2005), pp. 5–12. DOI: 10.1145/1064413.1064415. URL: https://doi.org/10.1145/1064413.1064415. | cit. on p. 103

[61]Frank Kelly et al. “Fairness and stability of end-to-end congestion control”. In: *European journal of control* 9.2-3 (2003), pp. 159–176. | cit. on p. 103

[62]Ramin Khalili, Nicolas Gast, Miroslav Popovic, and Jean-Yves Le Boudec. “MPTCP is not Pareto-Optimal: Performance Issues and a Possible Solution”. In: *IEEE/ACM Transactions on Networking* (2013), p. 15. DOI: 10.1109/TNET.2013.2274462. URL: https://hal.inria.fr/hal-01086030. | cit. on p. 103

[63]Atul Khanna and John Zinky. “The revised ARPANET routing metric”. In: *ACM SIGCOMM Computer Communication Review* 19.4 (1989), pp. 45–56. | cit. on pp. 1, 103

[64]E. Kohler, M. Handley, and S. Floyd. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340 (Proposed Standard). RFC. Updated by RFCs 5595, 5596, 6335, 6773. Fremont, CA, USA: RFC Editor, Mar. 2006. DOI: 10.17487/RFC4340. URL: https://www.rfc-editor.org/rfc/rfc4340.txt. | cit. on p. 17

[65]N. Kuhn, E. Lochin, A. Mifdaoui, et al. “DAPS: Intelligent delay-aware packet scheduling for multipath transport”. In: *2014 IEEE International Conference on Communications (ICC)*. June 2014, pp. 1222–1227. DOI: 10.1109/ICC.2014.6883488. | cit. on pp. 26, 28, 72

[66]Adam Langley, Alistair Riddoch, Alyssa Wilk, et al. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: ACM, 2017, pp. 183–196. DOI: 10.1145/3098822.3098842. URL: http://doi.acm.org/10.1145/3098822.3098842. | cit. on pp. 14, 19, 24, 29, 60

[67]Sungwon Lee and Dongkyun Kim. “Two Fast Retransmit Techniques in UWSNs with ACK Indiscretion Problem”. In: 2014 (Jan. 2014), pp. 1–9. | cit. on p. 15

[68]Yeon-sup Lim, Erich M. Nahum, Don Towsley, and Richard J. Gibbens. “ECF: An MPTCP Path Scheduler to Manage Heterogeneous Paths”. In: *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT ’17. New York, NY, USA: ACM, 2017, pp. 147–159. DOI: 10.1145/3143361.3143376. URL: http://doi.acm.org/10.1145/3143361.3143376. | cit. on pp. 28, 72, 77

[69]Igor Lopez, Marina Aguado, Christian Pinedo, and Eduardo Jacob. “SCADA Systems in the Railway Domain: Enhancing Reliability through Redundant MultipathTCP”. In: *2015 IEEE 18th International Conference on Intelligent Transportation Systems*. ISSN: 2153-0017. Sept. 2015, pp. 2305–2310. DOI: 10.1109/ITSC.2015.372. | cit. on p. 8

[70]Katerina Mania, Bernard D. Adelstein, Stephen R. Ellis, and Michael I. Hill. “Perceptual Sensitivity to Head Tracking Latency in Virtual Environments with Varying Degrees of Scene Complexity”. In: *Proceedings of the 1st Symposium on Applied Perception in Graphics and Visualization*. APGV ’04. Los Angeles, California, USA: Association for Computing Machinery, 2004, pp. 39–47. DOI: 10.1145/1012551.1012559. URL: https://doi-org.ins2i.bib.cnrs.fr/10.1145/1012551.1012559. | cit. on p. 2

[71]Robin Marx, Tom De Decker, Peter Quax, and Wim Lamotte. “Of the Utmost Importance: Resource Prioritization in HTTP/3 over QUIC”. In: *Proceedings of the 15th International Conference on Web Information Systems and Technologies, WEBIST 2019, Vienna, Austria, September 18-20, 2019*. Ed. by Alessandro Bozzon, Francisco José Domínguez Mayo, and Joaquim Filipe. ScitePress, 2019, pp. 130–143. DOI: 10.5220/0008191701300143. URL: https://doi.org/10.5220/0008191701300143. | cit. on pp. 62, 94

[72]Patrick Meenan. *Better HTTP/2 Prioritization for a Faster Web*. https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/. Online; accessed on 17 June 2020. May 2019. URL: https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/. | cit. on p. 62

[73]Patrick Meenan. *HTTP/3 Prioritization Proposal*. https://github.com/pmeenan/http3-prioritization-proposal. Online; accessed on 20 August 2020. June 2019. URL: https://github.com/pmeenan/http3-prioritization-proposal. | cit. on p. 62

[74]Patrick Meenan. *Optimizing HTTP/2 prioritization with BBR and tcp notsent lowat*. https://blog.cloudflare.com/http-2-prioritization-with-nginx/. Online; accessed on 20 August 2020. Oct. 2018. URL: https://blog.cloudflare.com/http-2-prioritization-with-nginx/. | cit. on p. 23

[75]Michael Mitzenmacher. “The power of two choices in randomized load balancing”. In: *IEEE Transactions on Parallel and Distributed Systems* 12.10 (2001), pp. 1094–1104. | cit. on p. 101

[76]J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896 (Historic). RFC. Obsoleted by RFC 7805. Fremont, CA, USA: RFC Editor, Jan. 1984. DOI: 10.17487/RFC0896. URL: https://www.rfc-editor.org/rfc/rfc896.txt. | cit. on pp. 18, 25

[77]NGINX. *NGINX SSL Performance*. https://www.nginx.com/wp-content/uploads/2014/07/NGINX-SSL-Performance.pdf. Online; accessed on 14 December 2018. July 2014. URL: https://www.nginx.com/wp-content/uploads/2014/07/NGINX-SSL-Performance.pdf. | cit. on p. 54

[78]Ashkan Nikravesh, Yihua Guo, Feng Qian, Z. Morley Mao, and Subhabrata Sen. “An in-depth understanding of multipath TCP on mobile devices: measurement and system design”. In: *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*. MobiCom ’16. New York City, New York: Association for Computing Machinery, Oct. 2016, pp. 189–201. DOI: 10.1145/2973750.2973769. URL: http://doi.org/10.1145/2973750.2973769 (visited on Apr. 9, 2020). | cit. on pp. 26, 30

[79] *NLNOG RING: A distributed worldwide network for measuring and troubleshooting the Internet*. en-US. URL: https://ring.nlnog.net/ (visited on June 15, 2018). | cit. on p. 8

[80] Kazuho Oku and Lucas Pardue. *Extensible Prioritization Scheme for HTTP*. Internet-Draft draft-ietf-httpbis-priority-01. Work in Progress. Internet Engineering Task Force, July 2020. 20 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-priority-01. | cit. on p. 62

[81] Christoph Paasch, Simone Ferlin, Ozgu Alay, and Olivier Bonaventure. "Experimental Evaluation of Multipath TCP Schedulers". In: *Proceedings of the 2014 ACM SIGCOMM Workshop on Capacity Sharing Workshop*. CSWS '14. New York, NY, USA: ACM, 2014, pp. 27–32. DOI: 10.1145/2630088.2631977. URL: http://doi.acm.org/10.1145/2630088.2631977. | cit. on pp. 7, 27

[82] Giorgos Papastergiou, Gorry Fairhurst, David Ros, et al. "De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives". In: *IEEE Communications Surveys & Tutorials* 19.1 (2017), pp. 619–639. DOI: 10.1109/COMST.2016.2626780. URL: http://ieeexplore.ieee.org/document/7738442/ (visited on June 21, 2018). | cit. on p. 59

[83] Tommy Pauly, Eric Kinnear, and David Schinazi. *An Unreliable Datagram Extension to QUIC*. Internet-Draft draft-ietf-quic-datagram-00. Work in Progress. Internet Engineering Task Force, Feb. 2020. 9 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-datagram-00. | cit. on p. 17

[84] V. Paxson, M. Allman, J. Chu, and M. Sargent. *Computing TCP's Retransmission Timer*. RFC 6298 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, June 2011. DOI: 10.17487/RFC6298. URL: https://www.rfc-editor.org/rfc/rfc6298.txt. | cit. on p. 43

[85] Andreas Petlund. "Improving latency for interactive, thin-stream applications over reliable transport". PhD thesis. University of Oslo, 2009. | cit. on pp. 3, 14, 43

[86] Maxime Piraux and Olivier Bonaventure. *Tunneling TCP inside QUIC*. Internet-Draft draft-piraux-quic-tunnel-tcp-02. Work in Progress. Internet Engineering Task Force, Aug. 2020. 12 pp. URL: https://datatracker.ietf.org/doc/html/draft-piraux-quic-tunnel-tcp-02. | cit. on p. 102

[87] Alexander Rabitsch, Per Hurtig, and Anna Brunstrom. "A Stream-Aware Multipath QUIC Scheduler for Heterogeneous Paths". In: *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC*. EPIQ'18. New York, NY, USA: ACM, 2018, pp. 29–35. DOI: 10.1145/3284850.3284855. URL: http://doi.acm.org/10.1145/3284850.3284855. | cit. on pp. 11, 69, 74, 80, 81, 87, 93, 96, 97

[88] Costin Raiciu, Sebastien Barre, Christopher Pluntke, et al. "Improving Datacenter Performance and Robustness with Multipath TCP". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 266–277. DOI: 10.1145/2018436.2018467. URL: http://doi.acm.org/10.1145/2018436.2018467. | cit. on pp. 6, 7

[89]Mohammad Rajiullah, Per Hurtig, Anna Brunstrom, Andreas Petlund, and Michael Welzl. “An evaluation of tail loss recovery mechanisms for TCP”. In: *ACM SIGCOMM Computer Communication Review* 45.1 (2015), pp. 5–11. | cit. on p. 43

[90]E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, Aug. 2018. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/rfc/rfc8446.txt. | cit. on pp. 54, 55, 59

[91]Vaspol Ruamviboonsuk, Ravi Netravali, Muhammed Uluyol, and Harsha V. Madhyastha. “Vroom: Accelerating the Mobile Web with Server-Aided Dependency Resolution”. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM ’17. Los Angeles, CA, USA: Association for Computing Machinery, Aug. 2017, pp. 390–403. DOI: 10.1145/3098822.3098851. URL: http://doi.org/10.1145/3098822.3098851 (visited on June 16, 2020). | cit. on p. 62

[92]G. Sarwar, R. Boreli, E. Lochin, A. Mifdaoui, and G. Smith. “Mitigating Receiver’s Buffer Blocking by Delay Aware Packet Scheduling in Multipath Data Transfer”. In: *2013 27th International Conference on Advanced Information Networking and Applications Workshops*. Mar. 2013, pp. 1119–1124. DOI: 10.1109/WAINA.2013.80. | cit. on pp. 26, 28, 72

[93]Arjuna Sathiaseelan, Raffaello Secchi, and Godred Fairhurst. “Enhancing TCP to Support Rate-limited Traffic”. In: *Proceedings of the 2012 ACM Workshop on Capacity Sharing*. CSWS ’12. Nice, France: ACM, 2012, pp. 39–44. DOI: 10.1145/2413219.2413230. URL: http://doi.acm.org/10.1145/2413219.2413230. | cit. on p. 14

[94]David Schinazi. *Using QUIC Datagrams with HTTP/3*. Internet-Draft draft-schinazi-quic-h3-datagram-04. Work in Progress. Internet Engineering Task Force, Apr. 2020. 6 pp. URL: https://datatracker.ietf.org/doc/html/draft-schinazi-quic-h3-datagram-04. | cit. on p. 99

[95]Robin Seggelmann, Michael Tüxen, and Erwin P. Rathgeb. “Stream scheduling considerations for SCTP”. In: *SoftCOM 2010, 18th International Conference on Software, Telecommunications and Computer Networks*. Sept. 2010, pp. 412–416. | cit. on pp. 25, 63, 64

[96]H2O web server. *HTTP/2 Directives > Latency Optimization*. Online; accessed on 29 August 2020. URL: https://h2o.examp1e.net/configure/http2_directives.html#latency-optimization. | cit. on p. 23

[97]Hang Shi, Yong Cui, Feng Qian, and Yuming Hu. “DTP: Deadline-aware Transport Protocol”. In: *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*. APNet ’19. Beijing, China: Association for Computing Machinery, Aug. 2019, pp. 1–7. DOI: 10.1145/3343180.3343191. URL: http://doi.org/10.1145/3343180.3343191 (visited on Apr. 23, 2020). | cit. on p. 64

[98]Xiang Shi, Lin Wang, Fa Zhang, and Zhiyong Liu. “FStream: Flexible Stream Scheduling and Prioritizing in Multipath-QUIC”. In: *2019 IEEE 25th International Conference on Parallel and Distributed Systems (ICPADS)*. ISSN: 1521-9097. Dec. 2019, pp. 921–924. DOI: 10.1109/ICPADS47876.2019.00136. | cit. on p. 75

[99] P. Srisuresh and M. Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. RFC 2663 (Informational). RFC. Fremont, CA, USA: RFC Editor, Aug. 1999. DOI: 10.17487/RFC2663. URL: https://www.rfc-editor.org/rfc/rfc2663.txt. | cit. on p. 59

[100] R. Stewart (Ed.) *Stream Control Transmission Protocol*. RFC 4960 (Proposed Standard). RFC. Updated by RFCs 6096, 6335, 7053. Fremont, CA, USA: RFC Editor, Sept. 2007. DOI: 10.17487/RFC4960. URL: https://www.rfc-editor.org/rfc/rfc4960.txt. | cit. on pp. 17, 18, 24, 29, 60, 63

[101] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. *Stream Control Transmission Protocol (SCTP) Partial Reliability Extension*. RFC 3758 (Proposed Standard). RFC. Fremont, CA, USA: RFC Editor, May 2004. DOI: 10.17487/RFC3758. URL: https://www.rfc-editor.org/rfc/rfc3758.txt. | cit. on p. 17

[102] R. Stewart, Q. Xie, K. Morneault, et al. *Stream Control Transmission Protocol*. RFC 2960 (Proposed Standard). RFC. Obsoleted by RFC 4960, updated by RFC 3309. Fremont, CA, USA: RFC Editor, Oct. 2000. DOI: 10.17487/RFC2960. URL: https://www.rfc-editor.org/rfc/rfc2960.txt. | cit. on p. 60

[103] Ars Technica. *SpaceX Starlink speeds revealed as beta users get downloads of 11 to 60 Mbps*. URL: https://arstechnica.com/information-technology/2020/08/spacex-starlink-beta-tests-show-speeds-up-to-60mbps-latency-as-low-as-31ms/ (visited on Aug. 14, 2020). | cit. on p. 5

[104] Martin Thomson and Sean Turner. *Using TLS to Secure QUIC*. Internet-Draft draft-ietf-quic-tls-24. Work in Progress. Internet Engineering Task Force, Nov. 2019. 49 pp. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-tls-24. | cit. on pp. 54, 55

[105] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz. "Multipath QUIC: A Deployable Multipath Transport Protocol". In: *2018 IEEE International Conference on Communications (ICC)*. May 2018, pp. 1–7. DOI: 10.1109/ICC.2018.8422951. | cit. on pp. 71, 96

[106] Thomas Waltemate, Irene Senna, Felix Hülsmann, et al. "The Impact of Latency on Perceptual Judgments and Motor Performance in Closed-Loop Interaction in Virtual Reality". In: *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology*. VRST '16. Munich, Germany: Association for Computing Machinery, 2016, pp. 27–35. DOI: 10.1145/2993369.2993381. URL: https://doi-org.ins2i.bib.cnrs.fr/10.1145/2993369.2993381. | cit. on p. 2

[107] Jing Wang, Yunfeng Gao, and Chenren Xu. "A Multipath QUIC Scheduler for Mobile HTTP/2". In: *Proceedings of the 3rd Asia-Pacific Workshop on Networking 2019*. APNet '19. Beijing, China: Association for Computing Machinery, Aug. 2019, pp. 43–49. DOI: 10.1145/3343180.3343185. URL: http://doi.org/10.1145/3343180.3343185 (visited on Feb. 13, 2020). | cit. on p. 76

[108]Xiao Sophia Wang, Aruna Balasubramanian, Arvind Krishnamurthy, and David Wetherall. “Demystifying Page Load Performance with WProf”. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 473–485. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/wang_xiao. | cit. on p. 86

[109]Yaogong Wang, Injong Rhee, and Sangtae Ha. “Augment SCTP multi-streaming with pluggable scheduling”. In: *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Apr. 2011, pp. 810–815. DOI: 10.1109/INFCOMW.2011.5928924. | cit. on p. 64

[110]Maarten Wijnants, Robin Marx, Peter Quax, and Wim Lamotte. “HTTP/2 Prioritization and its Impact on Web Performance”. In: *Proceedings of the 2018 World Wide Web Conference*. WWW ’18. Lyon, France: International World Wide Web Conferences Steering Committee, Apr. 2018, pp. 1755–1764. DOI: 10.1145/3178876.3186181. URL: http://doi.org/10.1145/3178876.3186181 (visited on May 24, 2020). | cit. on pp. 61, 62, 94

[111]Damon Wischik, Mark Handley, and Costin Raiciu. “Control of Multipath TCP and Optimization of Multipath Routing in the Internet”. en. In: *Network Control and Optimization*. Ed. by Rudesindo Núñez-Queija and Jacques Resing. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2009, pp. 204–218. DOI: 10.1007/978-3-642-10406-0_14. | cit. on p. 103

[112]Kiran Yedugundla, Simone Ferlin, Thomas Dreibholz, et al. “Is multi-path transport suitable for latency sensitive traffic?” In: *Computer Networks* 105 (Aug. 2016), pp. 1–21. DOI: 10.1016/j.comnet.2016.05.008. URL: http://www.sciencedirect.com/science/article/pii/S1389128616301396 (visited on June 12, 2017). | cit. on p. 27

[113]T. Ylonen and C. Lonvick (Ed.) *The Secure Shell (SSH) Protocol Architecture*. RFC 4251 (Proposed Standard). RFC. Updated by RFC 8308. Fremont, CA, USA: RFC Editor, Jan. 2006. DOI: 10.17487/RFC4251. URL: https://www.rfc-editor.org/rfc/rfc4251.txt. | cit. on p. 59

[114]L. Zhu, Z. Hu, J. Heidemann, et al. “Connection-Oriented DNS to Improve Privacy and Security”. In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 171–186. DOI: 10.1109/SP.2015.18. | cit. on pp. 29, 44, 95

List of Figures

List of Tables

Abstract

The network technologies that underpin the Internet have evolved significantly over the last decades, but one aspect of network performance has remained relatively unchanged: latency. In 25 years, the typical capacity or “bandwidth” of transmission technologies has increased by 5 orders of magnitude, while latency has barely improved by an order of magnitude. Indeed, there are hard limits on latency, such as the propagation delay which remains ultimately bounded by the speed of light.

This diverging evolution between capacity and latency is having a profound impact on protocol design and performance, especially in the area of transport protocols. It indirectly caused the Bufferbloat problem, whereby router buffers are persistently full, increasing latency even more. In addition, the requirements of end-users have changed, and they expect applications to be much more reactive. As a result, new techniques are needed to reduce the latency experienced by end-hosts.

This thesis aims at reducing the experienced latency by using end-to-end mechanisms, as opposed to “infrastructure” mechanisms. Two end-to-end mechanisms are proposed. The first is to multiplex several messages or data flows into a single persistent connection. This allows better measurements of network conditions (latency, packet loss); this, in turn, enables better adaptation such as faster retransmission. I applied this technique to DNS messages, where I show that it significantly improves end-to-end latency in case of packet loss. However, depending on the transport protocol used, messages can suffer from Head-of-Line blocking: this problem can be solved by using QUIC or SCTP instead of TCP.

The second proposed mechanism is to exploit multiple network paths (such as Wi-Fi, wired Ethernet, 4G). The idea is to use low-latency paths for latency-sensitive network traffic, while bulk traffic can still exploit the aggregated capacity of all paths. This idea was partially realized by Multipath TCP, but it lacks support for multiplexing. Adding multiplexing allows data flows to cooperate and ensures that the scheduler has better visibility on the needs of individual data flows. This effectively amounts to a scheduling problem that was identified only very recently in the literature as “stream-aware multipath scheduling”. My first contribution is to model this scheduling problem. As a second contribution, I proposed a new stream-aware multipath scheduler, SRPT-ECF, that improves the performance of small flows without impacting larger flows. This scheduler could be implemented as part of a MPQUIC (Multipath QUIC) implementation. More generally, these results open new opportunities for cooperation between flows, with applications such as improving WAN aggregation.

Résumé

Les technologies réseau qui font fonctionner Internet ont beaucoup évolué depuis ses débuts, mais il y a un aspect de la performance des réseaux qui a peu évolué : la latence. En 25 ans, le débit disponible en couche physique a augmenté de 5 ordres de grandeur, tandis que la latence s’est à peine améliorée d’un ordre de grandeur. La latence est en effet limitée par des contraintes physiques fortes comme la vitesse de la lumière.

Cette évolution différenciée du débit et de la latence a un impact important sur la conception des protocoles et leur performance, et notamment sur les protocoles de transport comme TCP. En particulier, cette évolution est indirectement responsable du phénomène de “Bufferbloat” qui remplit les tampons des routeurs et exacerbe encore davantage le problème de la latence. De plus, les utilisateurs sont de plus en plus demandeurs d’applications très réactives. En conséquence, il est nécessaire d’introduire des nouvelles techniques pour réduire la latence ressentie par les utilisateurs.

Le but de cette thèse est de réduire la latence ressentie en utilisant des mécanismes de bout en bout, par opposition aux mécanismes d’infrastructure réseau. Deux mécanismes de bout en bout sont proposés. Le premier consiste à multiplexer plusieurs messages ou flux de données dans une unique connexion persistante. Cela permet de mesurer plus finement les conditions du réseau (latence, pertes de paquet) et de mieux s’y adapter, par exemple avec de meilleures retransmissions. J’ai appliqué cette technique à DNS et je montre que la latence de bout en bout est grandement améliorée en cas de perte de paquet. Cependant, en utilisant un protocole comme TCP, il peut se produire un phénomène de blocage en ligne qui dégrade les performances. Il est possible d’utiliser QUIC ou SCTP pour s’affranchir de ce problème.

Le second mécanisme proposé consiste à exploiter plusieurs chemins, par exemple du Wi-Fi, une connexion filaire, et de la 4G. L’idée est d’utiliser les chemins de faible latence pour transporter le trafic sensible en priorité, tandis que le reste du trafic peut profiter de la capacité combinée des différents chemins. Multipath TCP implémente en partie cette idée, mais ne tient pas compte du multiplexage. Intégrer le multiplexage donne davantage de visibilité au scheduler sur les besoins des flux de données, et permettrait à ceux-ci de coopérer. Au final, on obtient un problème d’ordonnancement qui a été identifié très récemment, “l’ordonnancement multi-chemins sensible aux flux”. Ma première contribution est de modéliser ce problème. Ma seconde contribution consiste à proposer un nouvel algorithme d’ordonnancement pour ce problème, SRPT-ECF, qui améliore la performances des petits flux de données sans impacter celle des autres flux. Cet algorithme pourrait être utilisé dans une implémentation de MPQUIC (Multipath QUIC). De façon plus générale, ces résultats ouvrent des perspectives sur la coopération entre flux de données, avec des applications comme l’agrégation transparente de connexions Internet.