QUIC

Sommaire

QUIC

Présentation générale

La version actuellement utilisée de QUIC est la version 2 mais QUICv1 est encore largement utilisé.

QUIC a été développé par Jim Roskind, de Google, en 2012 et a été annoncé publiquement en 2013. Il est développé avec comme objectifs la performance et la faible latence, ce qui particulièrement utile lorsque de nombreuses requêtes sont envoyés à un serveur. Ce qui en fait un outil idéal pour les

navigateurs. Il est supporté par Chrome, Edge, Firefox et Safari.
En 2015 l'IETF (Internet Engineering Task Force) reprend QUIC en décidant de le normaliser.
Historiquement, QUIC signifie "Quick UDP Internet Connections" mais cet acronyme de Google n'est pas repris par l'IETF afin de ne pas créer d'ambiguité lors des futures évolutions du protocole.
L'IETF a adapté les travaux initiaux de Google afin de répondre aux besoins de standardisation et d'interopérabilité.
En 2021, les RFC concernant QUIC-HTTP/3 sont publiés, le protocole est officiellement standardisé.

Figure 1 - Nombre de requêtes par version HTTP (mai 2021 à mai 2022)

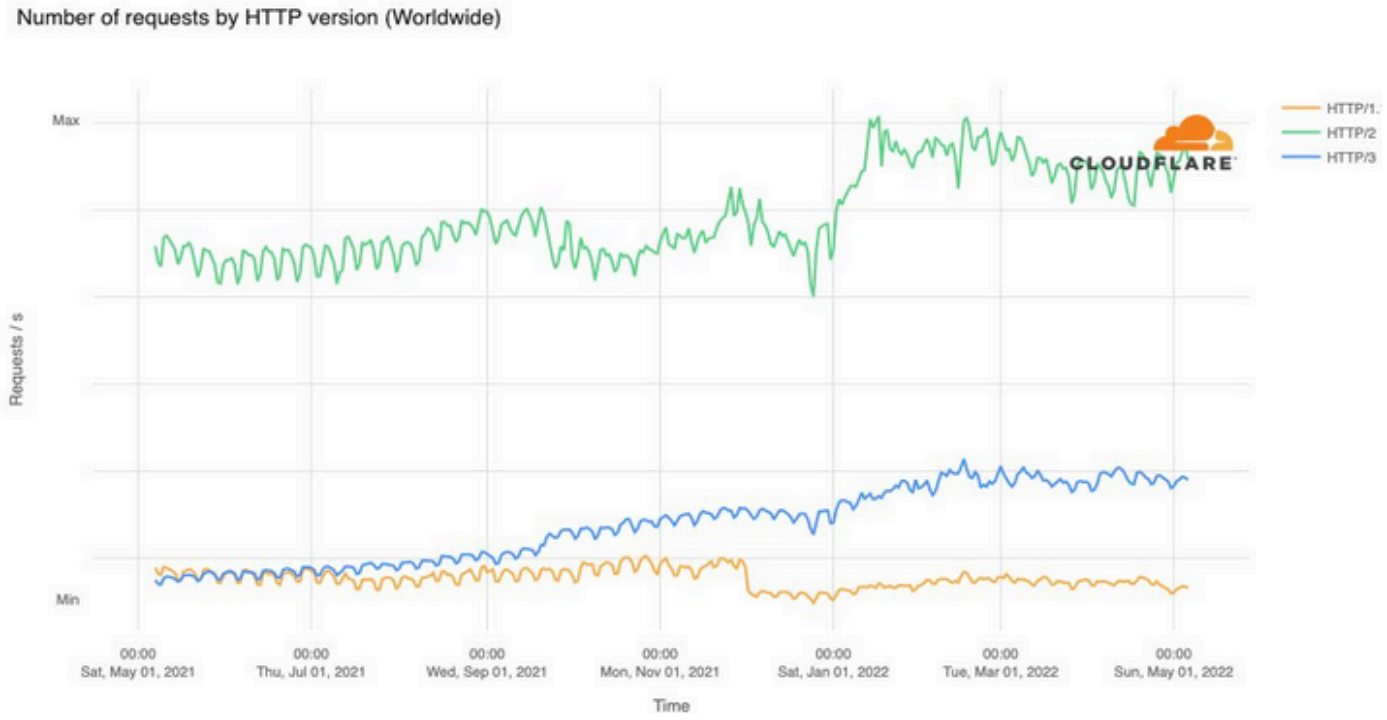
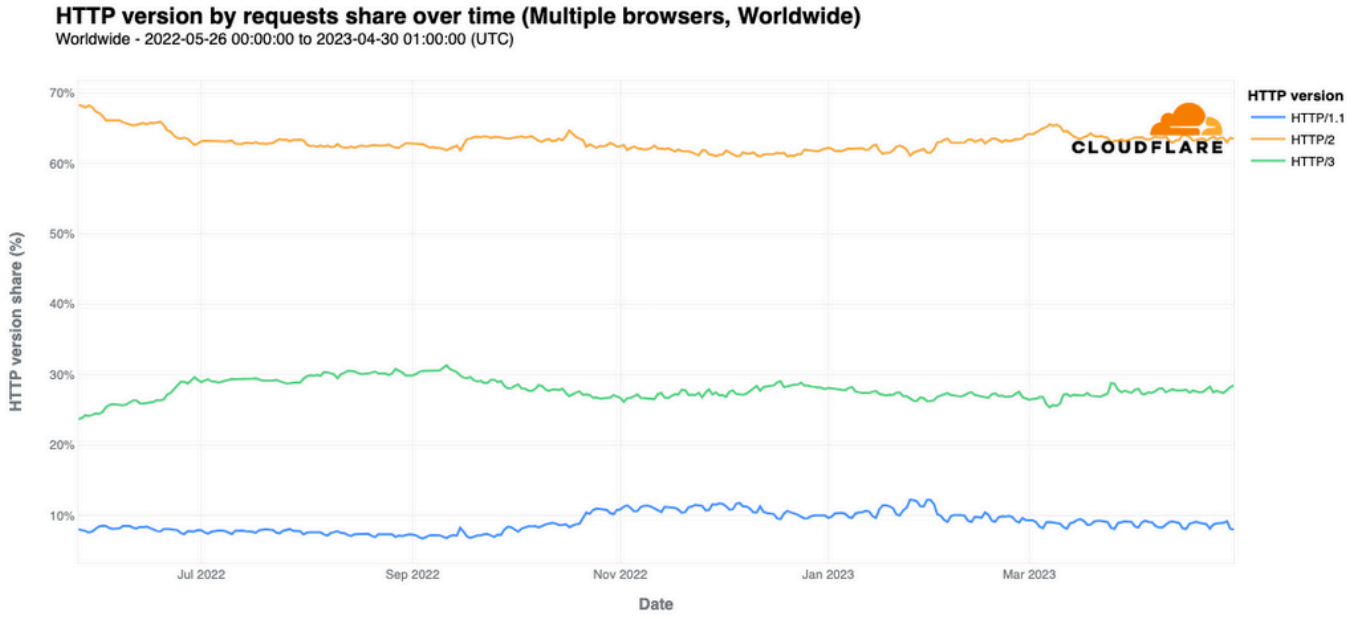


Figure 2 - Taux de requêtes par version HTTP (mai 2022 à mai 2023)

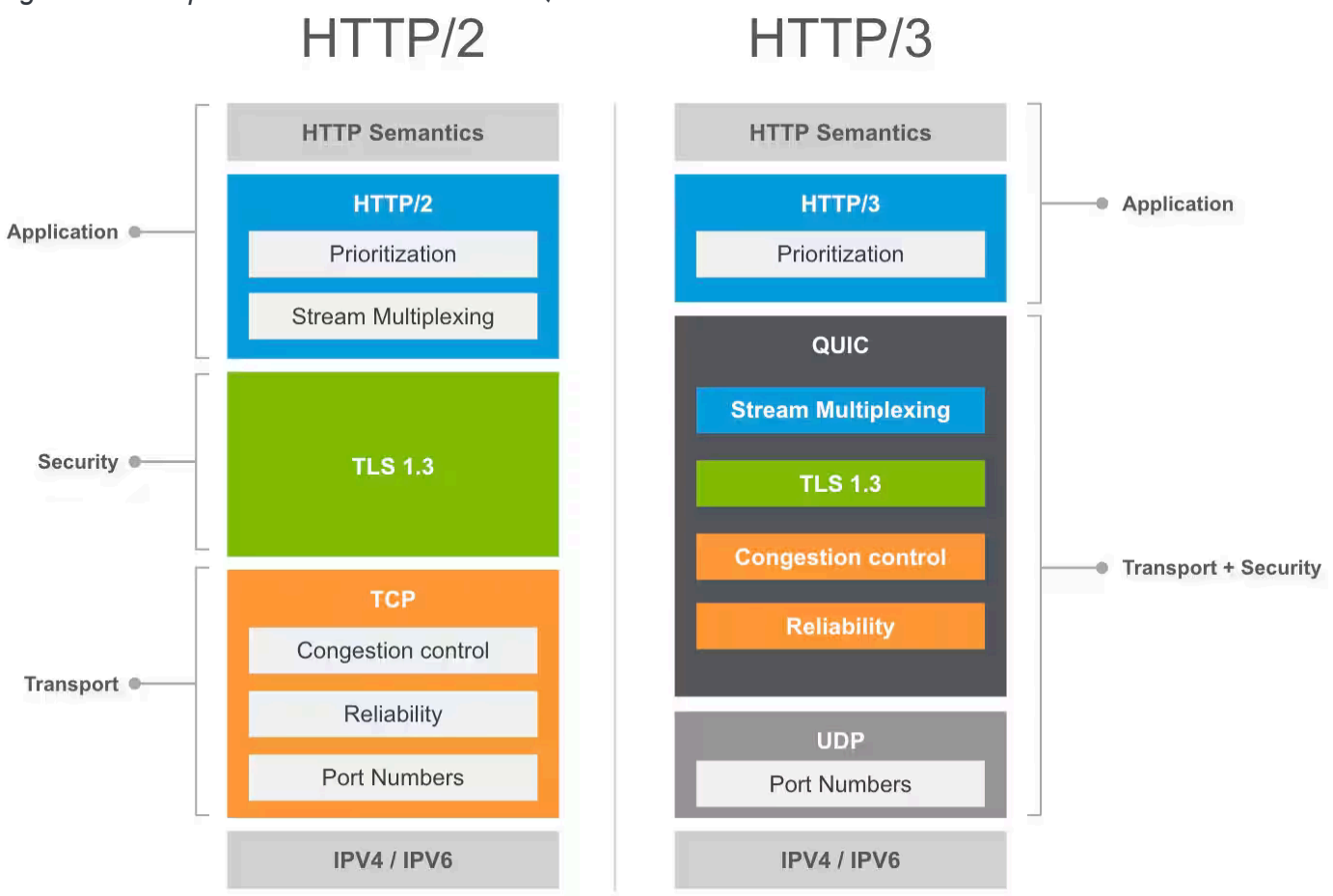


QUIC est un protocole de transport (couche ISO 4) et utilise UDP comme protocole sous-jacent. L'en-tête (header) et la charge utile (payload) sont tous deux stockés dans le payload d'un paquet UDP.
TLSv1.3 est imposé pour le chiffrement des données d'un paquet, ce qui fait de QUIC un protocole sécurisé.

Cette encapsulation permet un déploiement simple et rapide du protocole, sans avoir à modifier profondément le réseau. Bien que, comme nous le verrons plus tard, les middleboxes peuvent bloquer les paquets QUIC.

Le principal objectif de Google lors de son développement était que QUIC-HTTP/3 remplace TCP-HTTP/2.
La faible latence lors du handshake TLS (0-RTT), la gestion des changements d'IP du client (connection migration), le chiffrement des données (TLSv1.3) et le multiplexage des flux (suppression du HOL blocking) sont autant de qualité qui améliorent l'expérience utilisateur, essentiellement dans une utilisation dont la rapidité est primordiale (jeu en ligne, streaming vidéo, etc.).

Figure 3 - Comparaison TCP-HTTP/2 VS QUIC-HTTP/3

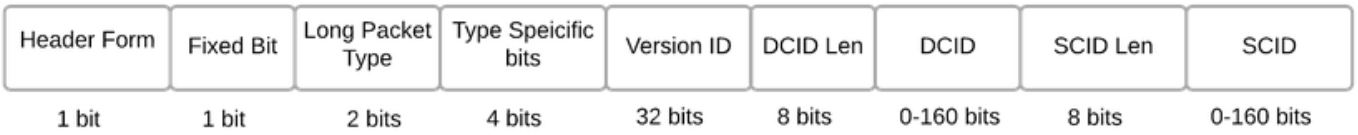


Header QUIC

QUIC dispose de deux headers différents : court et long.
Le long header est utilisé en priorité pour l'établissement d'une nouvelle connexion alors que le short header sera utilisé pour une connexion existante.

Long Header

Figure 4 - Long header



- Header Form : Identifie le type de header
- Fixed Bit : Indique l'état du paquet : si le bit n'est pas à 1, le paquet est invalide.
- Long Packet Type : Indique le type de paquet a en-tête long envoyé
- Type-Specific Bits : Bits spécifiques au paquet à en-tête long
- Version ID : Identifie la version de QUIC utilisée
- Destination Connection ID Length
- Destination Connection ID
- Source Connection ID Length
- Source Connection ID

Le Long Packet Type est composé de deux bits, le premier bit indique s'il s'agit d'un long header et le second bit indique le type de paquet à en-tête long. Voire Figure 3.

Figure 5 - Type de paquet à en-tête long selon la RFC 9000 (QUICv1)

Table 1: Long header packet types based on IETF RFC 9000

Type	Name	Description
0x00	Initial	Transports the first CRYPTO frames transmitted by the client and server during the key exchange and the Acknowledgment frames in both directions.
0x01	0-RTT	A 0-RTT packet sends "early" data from the client to the server before the handshake is completed.
0x02	Handshake	Packet is for sending and receiving encrypted handshake messages and acknowledgments between the server and the client.
0x03	Retry	The packet contains a server-generated address validation token. It's used by a server that wants to retry a connection.

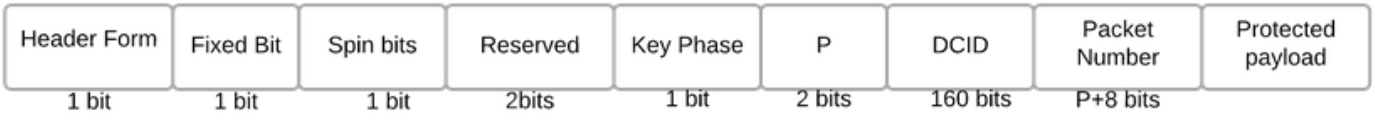
Figure 6 - Type de paquet à en-tête long selon RFC 9369 (QUICv2)

Table 2 : Long header packet type based on IETF RFC 9369

Type	Name	Description
0b01	Initial	Transports for the first CRYPTO frames transmitted by the client and server during the key exchange and the acknowledgment frames in both directions
0b10	0-RTT	A 0-RTT packets send « early » data from the client to the server before tje handshake is completed
0b11	Handshake	Packet is for sending and receiving encrypted handshake messages and acknowledgments between the server and the client
0b00	Retry	The packet contains a server-generated address validation token. It's used by a server that wants to retry a connection

Short Header

Figure 7 - Short Header



- Header Form
- Fixed Bit
- Spin Bit : bit de mesure de la latence
- Reserved Bits
- Key Phase : Indique la clé de protection du paquet

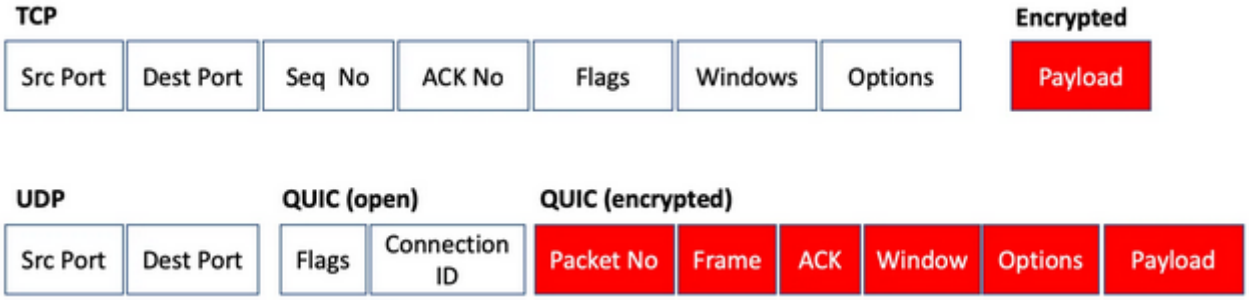
- Packet Number Length
- Destination Connexion ID
- Packet Number
- Payload

Fonctionnalités principales

Confidentialité, sécurité et intégrité

La RFC 9001 impose que QUIC soit chiffré, a minima, avec TLSv1.3. D'autres protocoles de chiffrement seront éventuellement utilisables par la suite, mais ce n'est pas encore défini.

Figure 8 - Comparaison du chiffrement entre TCP/TLS et QUIC



Lors du handshake QUIC, le premier paquet (appelé initial packet) n'est pas directement chiffré avec TLSv1.3, mais est tout de même protégé par un chiffrement dont la clé est une dérivation du DCID et d'un salt (grain de sel) défini dans la RFC 9001 pour QUICv1 et dans la RFC 9369 pour QUICv2.

Figure 9 - Le processus de chiffrement pour un paquet initial décrit dans la RFC 9001

```
initial_salt = 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a
initial_secret = HKDF-Extract(initial_salt,
                              client_dst_connection_id)

client_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "client in", "",
                                          Hash.length)
server_initial_secret = HKDF-Expand-Label(initial_secret,
                                          "server in", "",
                                          Hash.length)
```

Figure 10 - Salt QUICv2, tel que défini dans la RFC 9369

```
3.3.1.  Initial Salt

   The salt used to derive Initial keys in Section 5.2 of [QUIC-TLS]
   changes to:

   initial_salt = 0x0dede3def700a6db819381be6e269dcbf9bd2ed9

   This is the first 20 bytes of the sha256sum of "QUICv2 salt".
```

Ce chiffrement facilement réversible peut permettre à une middlebox de lire le paquet et, par extension, contribu à une ossification du protocole. Cette méthode sera problème vouée à évoluer dans les prochaines versions de QUIC afin de corriger ce problème (IETF - Protected QUIC Initial Packets).

Outre ce problème sur l'initial packet, quand le handshake est terminé, l'ensemble des données (une partie du header et la totalité du payload) sont chiffrés avec TLSv1.3 et le DPI ou l'analyse du paquet ne sont plus possibles.
Cela contribue fortement à la sécurité des données lors des échanges avec le protocole QUIC et, de facto, à leurs confidentialités.

Bien que QUIC se sert d'UDP comme vecteur de transport, il apporte la fiabilité des données qu'UDP n'a pas. Comme TCP, QUIC utilise des mécanismes d'acquittements (ACK) et de numéro de séquence (Sequence Number (SN)) pour vérifier la bonne connexion et la bonne transmission des paquets.
A la différence de TCP, QUIC chiffre les numéros ACK et SN dans son en-tête.

QUIC utilise la fonctionnalité de TLS qui permet le session resumption (reprise de session). Un ticket de session est généré lors d'une connexion à un serveur inconnu (ou qui n'a pas été contacté depuis longtemps). Ce ticket de session contient une clé de session qui est partagée entre le client et le serveur. Si cette clé de session n'est pas supprimée du cache des machines, alors la connexion peut reprendre sous forme de 0-RTT.

0-RTT/1-RTT

0-RTT

Quand une connexion a déjà été établie avec un serveur, ce dernier garde les informations de sessions et la connexion peut reprendre sans nécessiter un nouveau handshake.
Le client n'a donc qu'à envoyer un *NewSessionTicket* dans la trame *CRYPTO* et ainsi reprendre la session TLS.

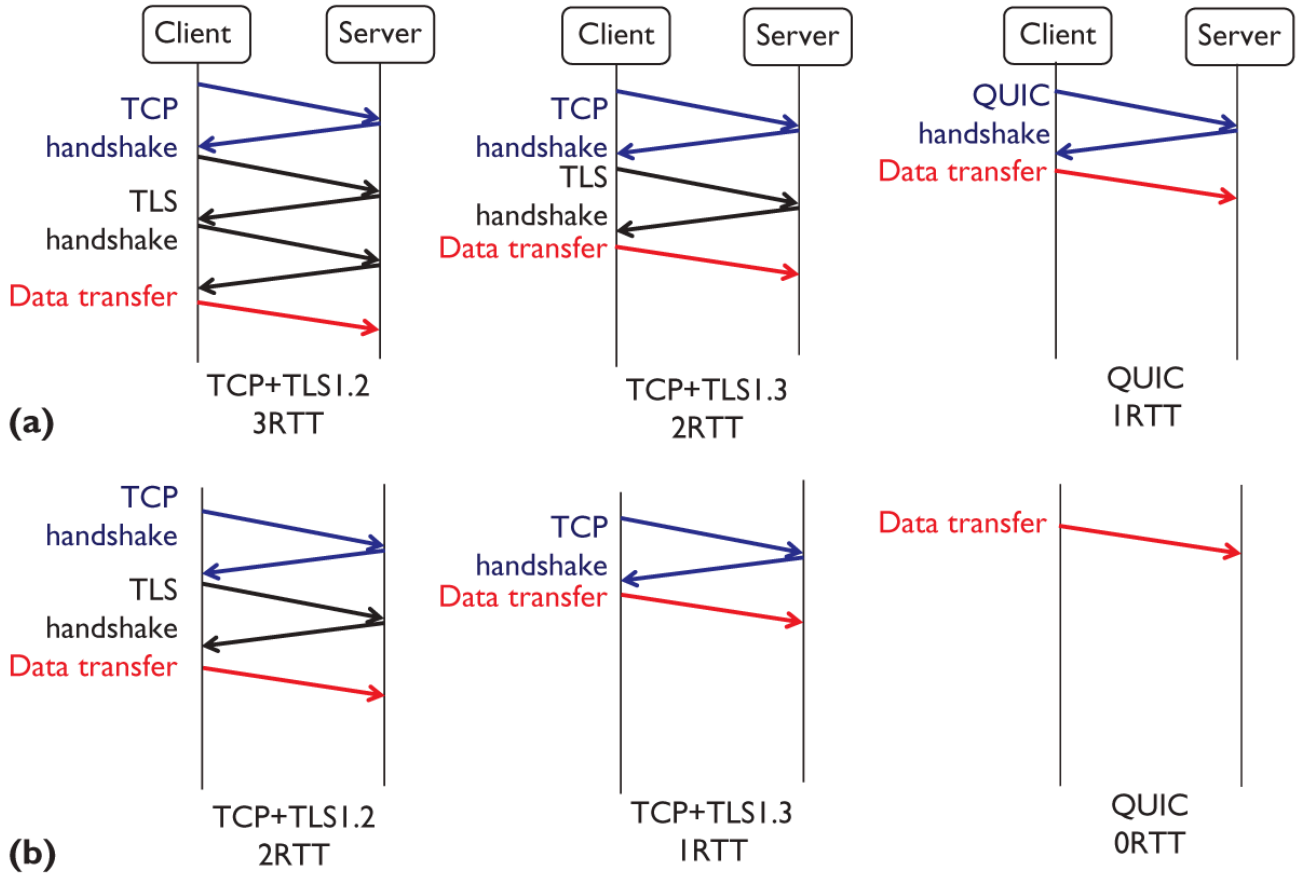
A contrario, TCP a besoin de refaire un handshake TCP et un handshake TLS, même dans le cadre d'une connexion qui avait déjà été établie.

Notez que la fonction 0-RTT peut représenter un risque de sécurité. Un pirate peut très envoyer une copie du dernier paquet que nous avons envoyé et, si l'application qui reçoit le paquet est sensible à cette requête, elle peut y répondre comme s'il s'agissait d'une demande légitime.

1-RTT

Dans un souci d'optimisation de latence, QUIC aborde le handshake TLS de connexion différemment de TCP.

Fig. 11 - TCP/TLS handshake VS QUIC handshake



TLS est intégré à QUIC de telle sorte que TLS n'a plus besoin de faire un handshake à part. Les informations de connexions TLS sont directement présentes dans le paquel QUIC ce qui réduit le nombre de RTT nécessaires à l'établissement d'une connexion sécurisée.

Congestion & latence

QUIC apporte une gestion de la congestion qu'UDP n'a pas.

Comme vu précédemment, la gestion de TLS par QUIC permet déjà la réduction des RTT, ce qui conduit directement à une réduction des échanges client/serveur et donc à une réduction de la latence.

Le contrôle des congestion de QUIC est en parti basé sur celui de TCP (NewReno). Néanmoins, quelques différences de traitement sont notables.
Tous les paquets QUIC ne demandent pas obligatoirement un ACK. Il est demandé dans la majorité des cas, mais cela n'a rien d'une nécessité. Cela réduit, dans une moindre mesure, la congestion réseau.

Pour traiter les paquets n'ayant pas d'ACK (ou ayant un ACK qui s'est perdu), QUIC dispose d'une fonction de sonde appelée PTO (Probe TimeOut).
Le PTO est un timer enclenché en l'absence de réception de paquet ACK ou d'un paquet ultérieur à celui envoyé précédemment. Quand le timer arrive à zéro, QUIC envoie un paquet au récepteur pour s'assurer que la connexion est toujours présente. Ce paquet sonde requiert une réponse ACK.
La réponse ACK peut contenir deux possibilités :

- L'ACK demandé par la sonde contient l'ACK du paquet perdu, auquel cas QUIC en déduit que seule la réponse ACK a été perdue.
- L'ACK demandé par la sonde ne contient pas l'ACK du paquet perdu, auquel cas QUIC réémet le paquet.

Le PTO est calculé de la façon suivante :

```
PTO = SRTT + max(4*RTTVAR, kGranularity) + Max Ack Delay
```

Où :

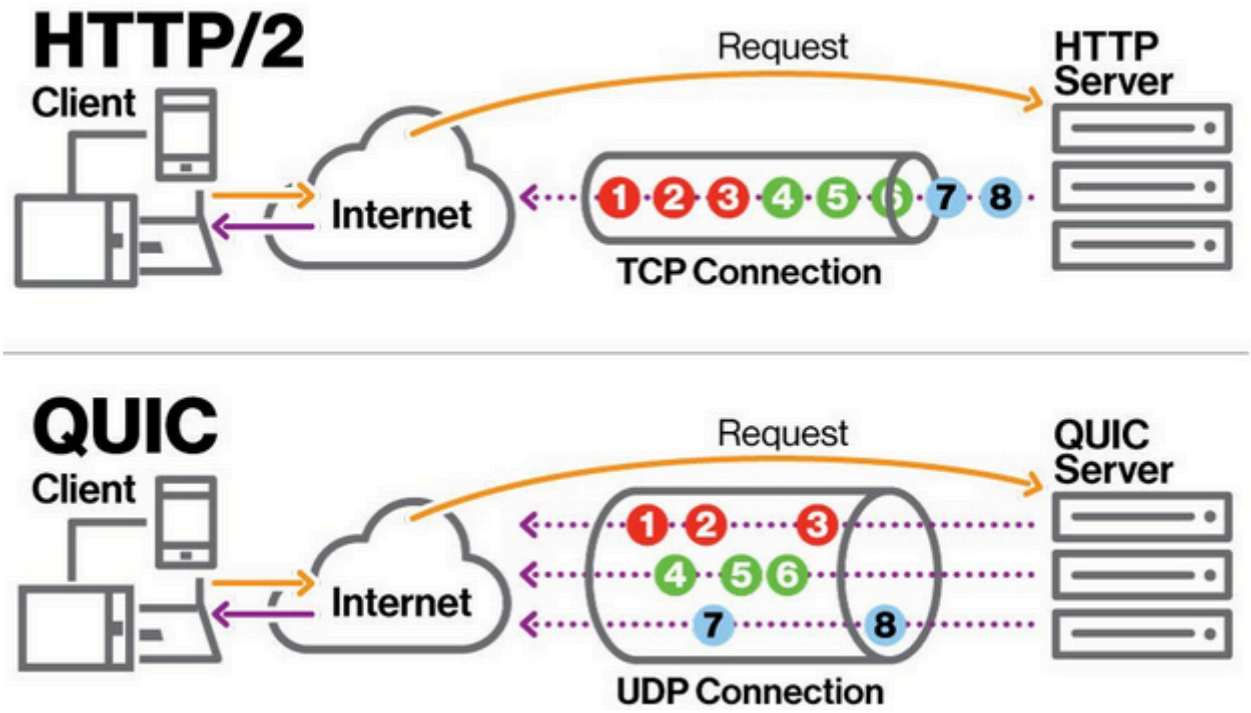
- SRTT (Smoothed RTT) : moyenne pondérée des RTT précédents
- RTTVAR (RTT Variation) : mesure de variabilité des RTT
- Max Ack Delay : le délai maximum que le récepteur peut attendre pour envoyer un ACK
- kGranularity : constante minimale pour éviter un PTO trop court

Les paquets QUIC qui n'arrivent plus dans le bon ordre pourrait faire croire, à tort, à des pertes de paquets et obliger QUIC à les réémettre. Ce problème est minoré avec TCP du fait que les middleboxes pouvaient, grâce au header circulant en clair, remettre les paquet dans le bon ordre et limiter la réémissions.
Les middleboxes ne pouvant plus faire la même chose avec QUIC (et c'est volontaire), le protocole ne définit plus le bon ordre des paquets avec un SN (Sequence Number) comme avec TCP, mais utilise un PN (Packet Number) qui est incrémenté unitairement à chaque nouveau paquet. Il est donc beaucoup simple de déduire l'ordre des paquets et de savoir quel paquet a été perdu.

Multiplexage & head of line blocking

Le Head of Line blocking (ou HOL blocking) arrive quand un paquet se retrouve bloqué. Avec TCP, si un paquet doit être réémis, alors les paquets suivants (qui eux n'ont pas besoin d'être réémis) sont en attente et ne peuvent pas être traités.
Le multiplexage des fluxs utilisé par QUIC permet de grandement attenué ce problème. Ainsi, si un paquet doit être réémis, il ne bloque que le flux concerné, les autres fluxs peuvent continuer d'être traités par le récepteur.
Le multiplexage réduit également la latence : plusieurs fluxs (images, textes, sons) peuvent être émis en même temps.

Figure 12 - Schéma comparatif de la gestion des fluxs TCP VS QUIC



Notons que HTTP/2 permet le multiplexage des fluxs, mais, à la différence de QUIC, un flux bloqué avec TCP (à cause d'un paquet perdu ou retardé) entraine le blocage de tous les autres fluxs : c'est ce qu'on appelle le HOL blocking.
Avec HTTP/3 (qui nécessite le protocole QUIC), seul le flux bloqué est en attente. Les fluxs en parallèle peuvent continuer leur transmission.

Gestion des erreurs

Pour détecter et gérer les erreurs dans la transmission des paquets, QUIC propose de plusieurs fonctionnalités.
La première, commune à TCP, est la présence d'un *checksum* (somme de contrôle) dans le header QUIC. Ce checksum permet de vérifier l'intégrité des données.
Contraitement à TCP, le checksum présent dans le header QUIC est chiffré, ce qui augmente sa robustesse.

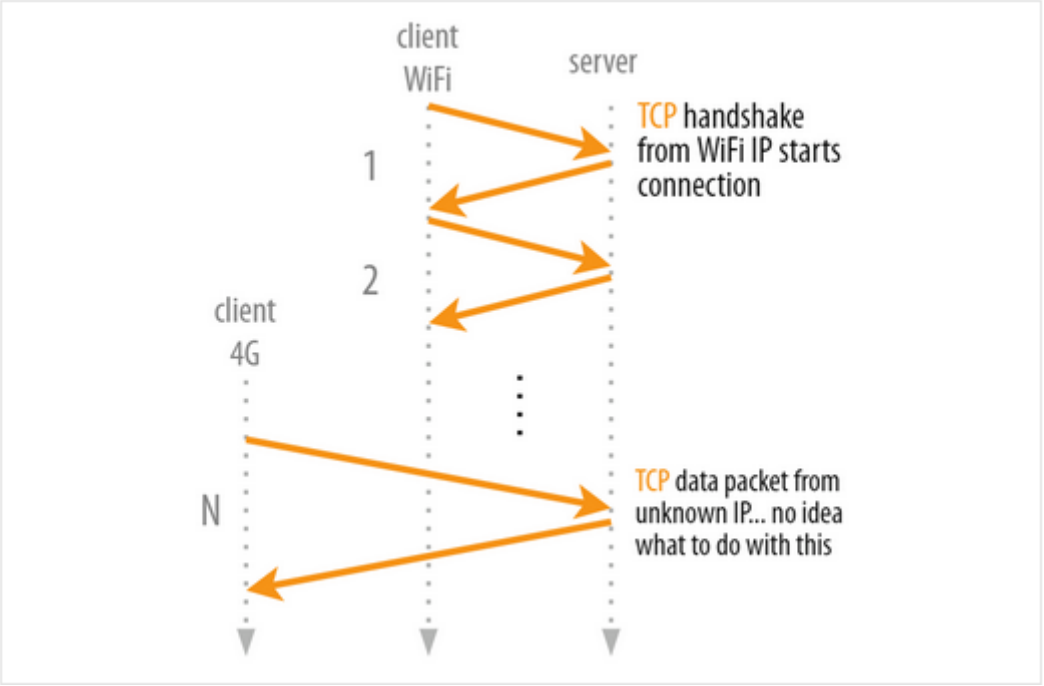
QUIC utilise, dans son header, un PN (Packet Number). Ici, contrairement à TCP et son SN (Sequence Number), l'incrémentation du PN se fait unitairement à chaque échange :

- Le premier paquet à le PN 1
- Le seconde le PN 2
- etc.
  Retrouver un paquet perdu est donc beaucoup plus trivial qu'avec TCP dont l'incrémentation se fait en fonction de la taille du paquet précédent.
  Pour améliorer la sécurité, le PN est chiffré avec TLSv1.3 dans le header de QUIC.

Connection migration

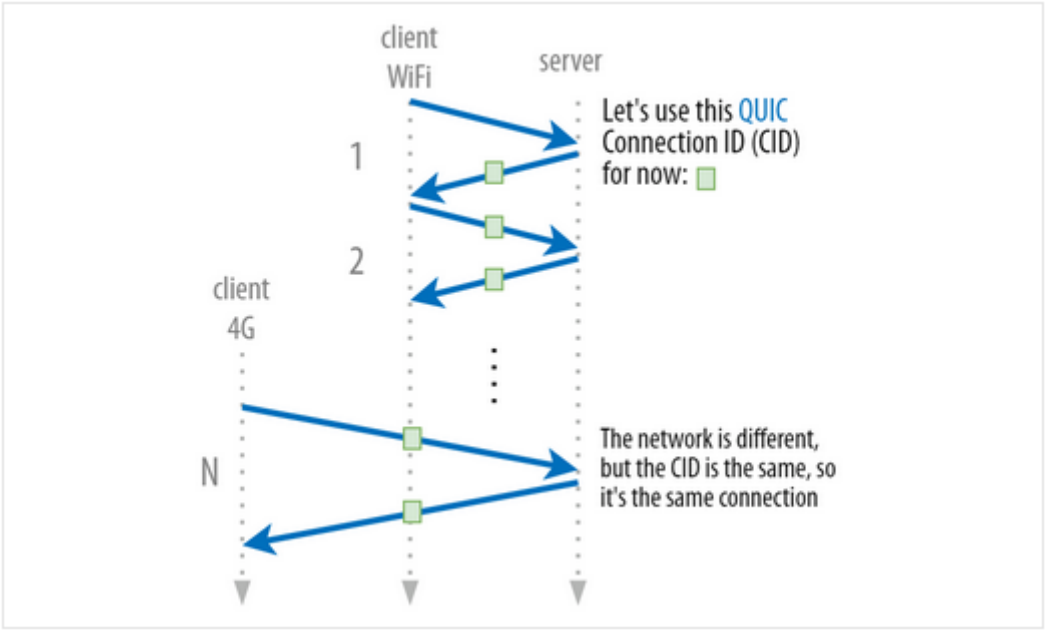
QUIC a été pensé pour être en accord avec les réseaux modernes (WiFi, 4G, etc.). Il comprend un mécanisme permettant de poursuivre une connexion malgré un changement d'adresse IP.
TCP, étant plus ancien, ne prévoit pas ce cas de figure dans son fonctionnement. Et un changement d'adresse IP nécessite l'établissement d'une nouvelle connexion.

Figure 13 - En cas de changement d'IP, TCP doit établir une nouvelle connexion



Pour ce faire, au lieu de se baser uniquement sur l'adresse IP pour déterminer un récepteur ou un émetteur, QUIC utilise des CID (Connection ID). Il existe un SCID (Source Connection ID) et un DCID (Destination Connection ID) qui sont tous deux intégrés dans le header.

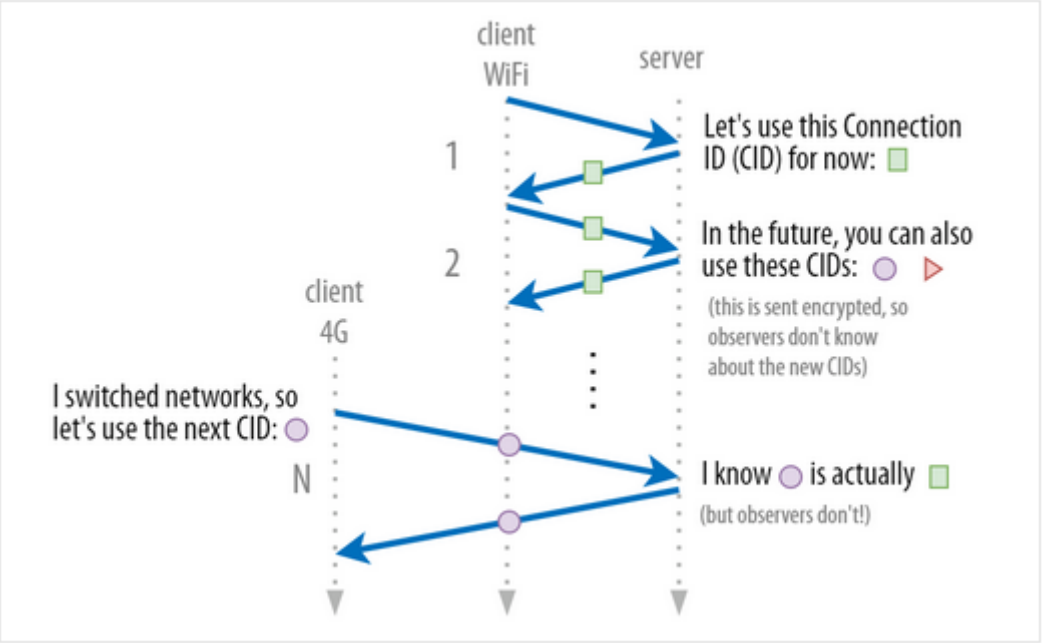
Figure 14 - Lors d'un changement de réseau, le CID reste le même. La connexion n'est pas perdue



Ces CID peuvent être modifiés en cas de changement de réseau. Si cela arrive, l'émetteur envoie au serveur un paquet contenant son nouveau CID. Le serveur vérifie la validité de ce nouveau CID et s'en

sert pour poursuivre la connexion existante.
Pour améliorer la confidentialité et éviter les attaques, les CID sont chiffrés dans les headers QUIC.

Figure 15 - Un nouveau CID est généré au changement de réseau. Son chiffrement empêche le spoofing



Performances de QUIC VS TCP

Une étude de 2016 - faite avant la standardisation de QUIC - a comparé SPDY3.1 (un ancètre de HTTP/2) sur QUIC versus HTTP/2 sur TCP [Does QUIC make the Web faster]. Il en ressort que QUIC est 90% pour performant sur une connexion médioce (bande passante faible, latence élevée) que TCP. Mais que ce chiffre descend à 60% sur une connexion de meilleure qualité.
QUIC s'illustrant particulièrement lorsque les pages contiennent des éléments lourds (ex : streamign vidéo HD).

Une étude plus récente - postérieure à la standardisation de QUIC et de HTTP/3 - nous indique que QUIC peut être moins performant, sur des connexions à haute vitesse (>500Mbps) [Zhang and al.].
UDP génère beaucoup plus de lecture de paquet dans le noyau du système d'exploitation, ce qui entraine une augmentation de la charge CPU. Cela s'explique car la version de QUIC utilisée dans l'étude n'implémente pas le GRO (UDP Generic Receive Offload) qui permettrait de combiner plusieurs paquet UDP en un seul gros datagramme, et, par conséquent, réduire la charge de traitement.
A contrario, TCP utilise TSO (TCP Segmentation Offload), qui est une méthode similaire à GRO, ce qui explique la charge moindre.

Figure 16 - Tests préliminaires de téléchargement de fichiers

Testbed	Download Time (s)		CPU Usage (%)	
	HTTP/2	HTTP/3	HTTP/2	HTTP/3
Desktop, Ethernet	9.32	18.60 (+99%)	77.5	96.9
Pixel 5, low-band 5G	37.11	78.65 (+112%)	121.55	161.77
Pixel 5, mmWave 5G	30.10	63.20 (+110%)	128.43	165.20

Figure 17 - Débit sur le navigateur Chrome lors d'un téléchargement

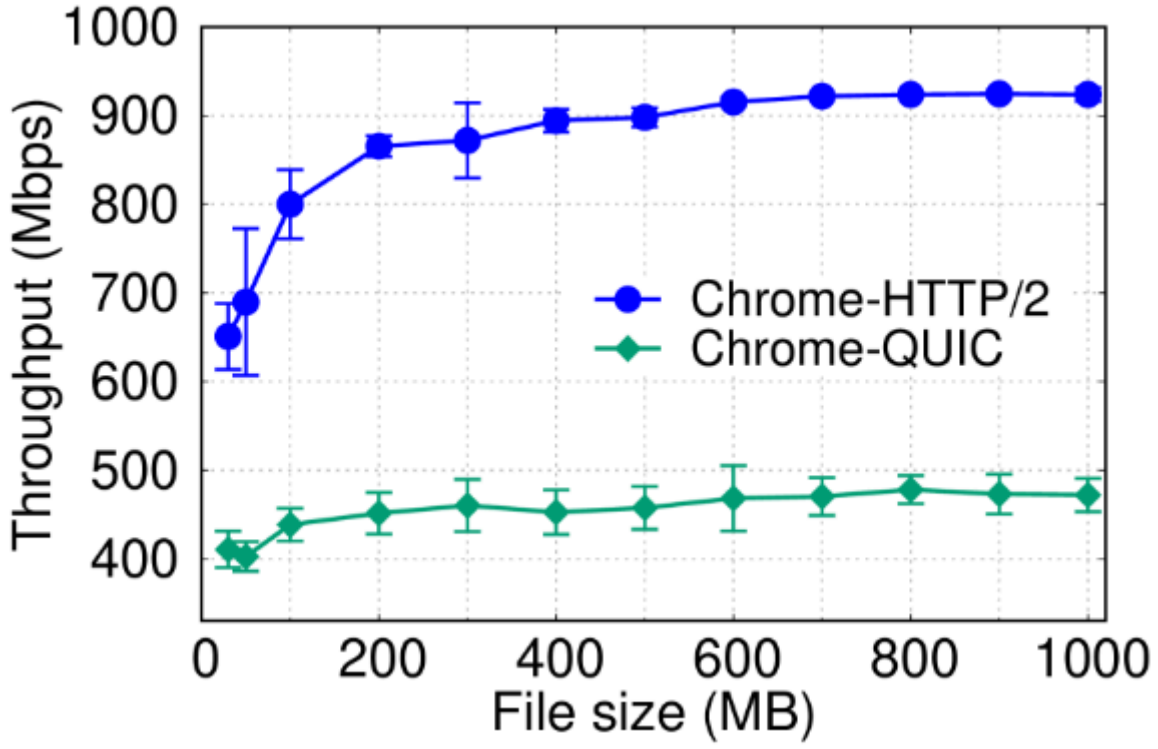


Figure 18 - Utilisation du CPU du navigateur Chrome

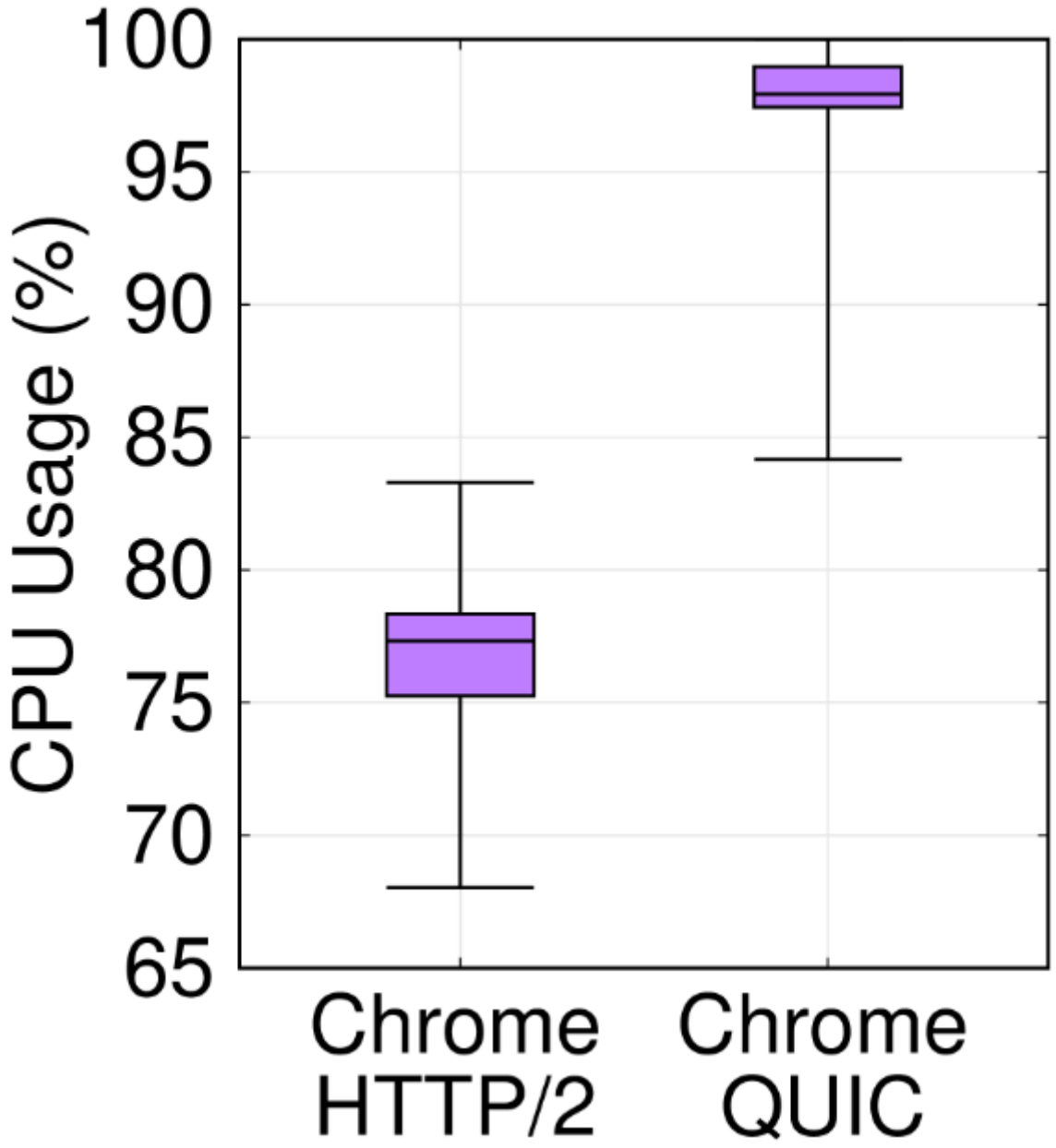
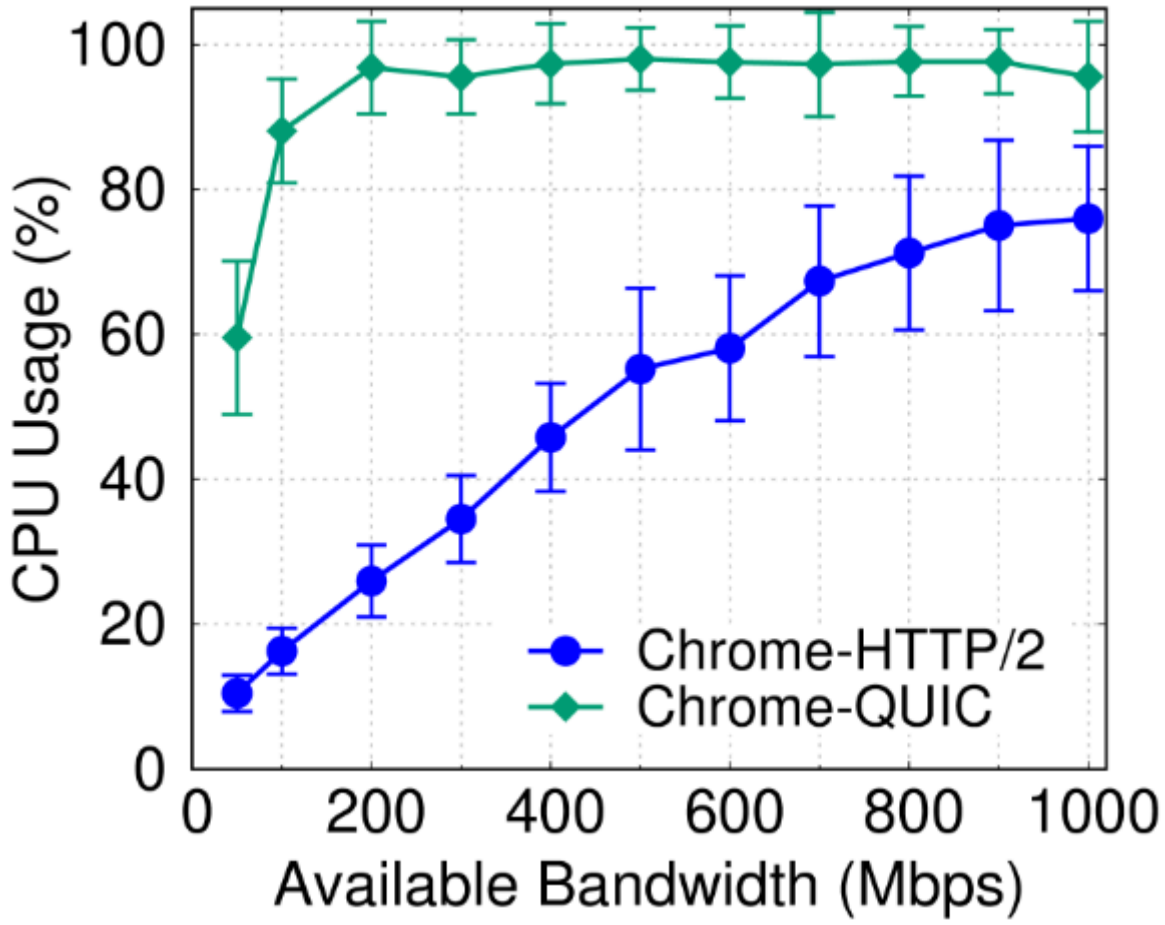
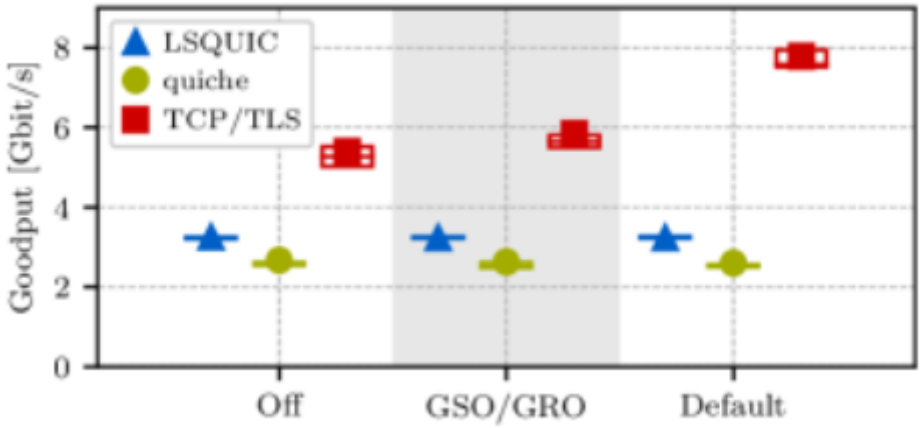


Figure 19 - Débit et utilisation du CPU du navigateur Chrome lors du téléchargement d'un fichier avec une bande-passante limitée



Une seconde étude [Jaegr and al.] confirme que les différentes implémentations actuelles de QUIC, même lorsque que GRO/GSO sont activés, sousperforment par rapport à TCP sur des connexions haut débit.
QUIC étant un protocole très récent, les implémentations du protocole, tout comme les implémentations de GRO/GOS nécessitent encore un très gros travail d'optimisation pour profiter pleinement des capacités de QUIC.

Figure 20 - Impact du déchargement matériel sur le débit, par défaut GRO/GSO et TSO sont activés



Cette étude souligne d'autres points importants pour la bonne évolution de QUIC. Actuellement, les implémentations ont un tampon de réception UDP qui n'est pas adapté à QUIC, ce qui entraine des pertes de paquets et, par conséquent, une perte de débit. La grande disparité des implémentations

(LSQUIC, quiche, picoquic, etc.) est aussi un frein. Un plus grand universalisme du protocole aiderait et facileterait son évolution.

Ossification des protocoles et neutralité du Net

Mise en place de QUICv2 afin de modifier le salt de l'initial packet de remplacer le type des long header packet.

QUIC, dans son approche fondamentale, souhaite supprimer l'ossification des protocoles, essentiellement du à TCP.
Les middleboxes (routeur, FW, proxy, IDS, etc.) ont du mal à supporter une évolution du protocole TCP. Un exemple typique est la mise en place de TLS1.3 et les problèmes rencontrés par ces middleboxes. Le chiffrement du hanshake et du SNI (Server Name Identification) a pu contraindre ces boxes a prendre les paquets TLS1.3 comme des paquets corrompus.

Bien que le but de ces boxes puisse parfois être louable (sécurité, priorisation du trafic, etc) cela se fait uniquement grâce à une analyse de la couche transport, ce qui pose un problème évident de neutralité. De plus, il arrive parfois (souvent ?) que ces mêmes boxes s'octroient le droit de faire du DPI, voire utilise la lisibilité des informations de TCP pour bloquer le trafic BitTorrent en envoyant à paquet RST à la place du serveur.

Le chiffrement apporté par QUIC empêche donc une analyse directe des paquets et des en-têtes, bien qu'il soit toujours possible de faire une analyse indirecte du trafic (et des protocoles utilisés) via les RTT.

Cette ossification pourrait apporter une certaine réticence des entreprises à accepter le protocole QUIC, prétextant un besoin d'analyse des paquets à des fins de sécurité du réseau.
Néanmoins, l'existence même de la RFC 7258 nous rappelle que l'observation persistente est une attaque. Un affaiblissement du réseau permettant une écoute légale est un affaiblissement exploitable par les attaquants.

Monitoring du protocole

Mise en place du protocole

Conclusion

Ressources

- Bortzmeyer - QUIC est normalisé (FR)

- Bortzmeyer - Review RFC 9000 (FR)
- Bortzmeyer - Review RFC 9001 (FR)
- WashU/CSE - IETF QUIC v1 Design (EN)
- Akamai - Deliver Fast, Reliable, and Secure Web Experiences with HTTP/3 (EN)
- Cloudflare - Examining HTTP/3 usage one year one (EN)
- InternetSociety - How QUIC helps you seamlessly connect to different networks (EN)
- Univertsity of Houston/DCS - Does QUIC make the Web faster ? (EN)
- arXivLabs - Zhang and al.(2024) - QUIC is not Quick Enough over Fast Internet (EN)
- arXivLabs - Jaeger and al. (2023) - QUIC on the Highway: Evaluating Performance on High-Rate Links (EN)
- Tailscale - Increasing QUIC and UDP throughput (EN)
- RFC 8999 - Independant Properties of QUIC (EN)
- RFC 9000 - QUIC: A UDP-Based Multiplexed and Secure Transport (EN)
- RFC 9001 - Using TLS to Secure QUIC (EN)
- RFC 9002 - QUIC Loss Detection and Congestion Control (EN)
- RFC 9369 - QUIC Version 2 (EN)