RFC 9000 : QUIC: A UDP-Based Multiplexed and Secure Transport

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 28 mai 2021

Date de publication du RFC : Mai 2021

https://www.bortzmeyer.org/9000.html

Le protocole de transport QUIC <https://www.bortzmeyer.org/quic.html> vient d'être normalisé, dans une série de quatre RFC. J'ai écrit un résumé synthétique de QUIC <https://www.bortzmeyer.org/quic.html>, cet article porte sur le principal RFC du groupe, celui qui normalise le cœur de QUIC, le mécanisme de transport des données. QUIC, comme TCP, fournit aux protocoles applicatifs un service de transport des données fiable, et en prime avec authentification et confidentialité.

L'un des buts principaux de QUIC est de réduire la latence <https://www.bortzmeyer.org/latence.html>. La capacité <https://www.bortzmeyer.org/capacite.html> des réseaux informatiques augmente sans cesse, et va continuer de la faire, alors que la latence sera bien plus difficile à réduire. Or, elle est déterminante dans la perception de « vitesse » de l'utilisateur. Quels sont les problèmes que pose actuellement l'utilisation de TCP, le principal protocole de transport de l'Internet ?

— "*Head-of-line blocking*", quand une ressource rapide est bloquée par une lente située avant elle dans la file, ou quand la perte d'un seul paquet TCP bloque toute la connexion en attendant que les données manquantes soient réémises. Si on multiplexe au-dessus de TCP, comme le fait HTTP/2 (RFC 7540 [1]), tous les ruisseaux d'une même connexion doivent attendre.
— Latence due à l'ouverture de la session TLS, d'autant plus que TCP et TLS étant découplés, TLS doit attendre que TCP ait fini pour commencer sa négociation.

Le cahier des charges de QUIC était à peu près :

— Déployable aujourd'hui, sans changement important de l'Internet. Compte-tenu du nombre de boitiers intermédiaires intrusifs (cf. RFC 7663), cela exclut le développement d'un nouveau protocole de transport reposant directement sur IP : il faut passer sur TCP ou UDP. Les solutions « révolutionnaires » ont donc été abandonnées immédiatement.
— Faible latence, notamment pour le démarrage de la session.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc7540.txt

— Meilleure gestion de la mobilité (ne pas casser les sessions si on change de connectivité).
— Assurer la protection de la vie privée au moins aussi bien qu'avec TCP+TLS (donc, tout chiffrer). Le chiffrement ne sert d'ailleurs pas qu'à gêner la surveillance : il a aussi pour but d'empêcher les modifications que se permettent bien des FAI <https://www.bortzmeyer.org/enrichir-qui.html>.
— Services aux applications à peu près semblables à ceux de TCP (distribution fiable et ordonnée des données).
— Coexistence heureuse avec TCP, aucun des deux ne doit prendre toute la capacité au détriment de l'autre (cf. RFC 5348). QUIC, comme TCP, contrôle la quantité de données envoyées, pour ne pas écrouler le réseau.

Certains de ces objectifs auraient pu être atteints en modifiant TCP. Mais TCP étant typiquement mis en œuvre dans le noyau du système d'exploitation, tout changement de TCP met un temps trop long à se diffuser. En outre, si les changements à TCP sont importants, ils peuvent être bloqués par les boitiers intermédiaires, comme si c'était un nouveau protocole.

QUIC peut donc, plutôt qu'à TCP, être comparé à SCTP tournant sur DTLS (RFC 8261), et on peut donc se demander pourquoi ne pas avoir utilisé ce système. C'est parce que :

— La latence pour établir une session est très élevée. SCTP et DTLS étant deux protocoles séparés, il faut d'abord se « connecter » avec DTLS, **puis** avec SCTP (RFC 8261, section 6.1).
— Cette séparation de SCTP et DTLS fait d'ailleurs que d'autres tâches sont accomplies deux fois (alors que dans QUIC, chiffrement et transport sont intégrés, cf. RFC 9001).

Pour une comparaison plus détaillée de SCTP et QUIC, voir l'"*Internet-Draft*" `draft-joseph-quic-comparison-q` (SCTP peut aussi tourner sur UDP - RFC 6951 mais j'ai plutôt utilisé SCTP sur DTLS comme point de comparaison, pour avoir le chiffrement.) Notez que ces "*middleboxes*" intrusives sont particulièrement répandues dans les réseaux pour mobiles, type 4G (où elles sont parfois appelée "*TCP proxies*"), puisque c'est un monde où on viole bien plus facilement la neutralité du réseau <https://www.bortzmeyer.org/neutralite.html>.

Faire mieux que TCP n'est pas évident. Ce seul RFC fait 207 pages, et il y a d'autres RFC à lire. Déjà, commençons par un peu de terminologie :

— Client et serveur ont le sens habituel, le client est l'initiateur de la connexion QUIC, le serveur est le répondeur,
— Un paquet QUIC n'est pas forcément un datagramme IP ; plusieurs paquets peuvent se trouver dans un même datagramme envoyé sur UDP,
— Certains paquets déclenchent l'émission d'un accusé de réception ("*ACK-eliciting packets*") mais pas tous,
— Dans un paquet, il y a plusieurs unités de données, les trames. Il existe plusieurs types de trames, par exemple `PING` sert uniquement à déclencher l'accusé de réception, alors que `STREAM` contient des données de l'application.
— Une adresse est la combinaison d'une adresse IP, et d'un port. Elle sert à identifier une extrémité du chemin entre client et serveur mais pas à identifier une connexion.
— L'identifiant de connexion (CID, "*connection ID*") joue ce rôle. Il permet à QUIC de gérer les cas où un routeur NAT change le port source, ou bien celui où on change de type de connexion.
— Un ruisseau ("*stream*") est un canal de données à l'intérieur d'une connexion QUIC. QUIC multiplexe des ruisseaux. Par exemple, pour HTTP/3 (le RFC n'est pas encore publié), chaque ressource (image, feuille de style, etc) voyagera dans un ruisseau différent.

Maintenant, plongeons dans le RFC. Rappelez-vous qu'il est long ! Commençons par les ruisseaux (section 2 du RFC). Ils ressemblent aux ruisseaux de HTTP/2 (RFC 7540), ce qui est logique, QUIC ayant été conçu surtout en pensant à HTTP. Chaque ruisseau est un flux ordonné d'octets. Dans une même connexion QUIC, il y a plusieurs ruisseaux. Les octets d'un ruisseau sont reçus dans l'ordre où ils ont été envoyés (plus exactement, QUIC doit fournir ce service mais peut aussi fournir un service dans le désordre), ce qui n'est pas forcément le cas pour les octets d'une même connexion. (Dans le cas de HTTP, cela sert à éviter qu'une ressource lente à se charger ne bloque tout le monde.) Comme avec TCP, les données sont un flot continu, même si elles sont réparties dans plusieurs trames. La création d'un

ruisseau est très rapide, il suffit d'envoyer une trame avec l'identifiant d'un ruisseau et hop, il est créé. Les ruisseaux peuvent durer très longtemps, ou au contraire tenir dans une seule trame.

J'ai parlé de l'identifiant d'un ruisseau. Ce numéro, ce "*stream ID*" est pair pour les ruisseaux créés par le client, impair s'ils sont créés par le serveur. Cela permet à client et serveur de créer des ruisseaux sans se marcher sur les pieds. Un autre bit dans l'identifiant indique si le ruisseau est bidirectionnel ou unidirectionnel.

Une application typique va donc créer un ou plusieurs ruisseaux, y envoyer des données, en recevoir, et fermer les ruisseaux, gentiment (trame `STREAM` avec le bit `FIN`) ou brutalement (trame `RESET_STREAM`). La machine à états complète des ruisseaux figure dans la section 3 du RFC.

Comme avec TCP, il ne s'agit pas d'envoyer des données au débit maximum, sans se soucier des conséquences. Il faut se contrôler, à la fois dans l'intérêt du réseau (éviter la congestion) et dans celui du récepteur, qui a peut-être du mal à traiter tout ce qu'on lui envoie. D'où, par exemple, la trame `STOP_SENDING` qui dit à l'émetteur de se calmer.

Plus fondamentalement, le système de contrôle de QUIC est décrit dans la section 4. Il fonctionne aussi bien par ruisseau qu'au niveau de toute la connexion. C'est le récepteur qui contrôle ce que l'émetteur peut envoyer, en disant « j'accepte au total N octets ». Il le fait à l'établissement de la connexion puis, lorsqu'il peut à nouveau traiter des données, via les trames `MAX_DATA` (pour l'ensemble de la connexion) et `MAX_STREAM_DATA` (valables, elles, pour un seul ruisseau). Normalement, les quantités d'octets que l'émetteur peut envoyer sont toujours croissantes. Par exemple, à l'établissement de la connexion, le récepteur annonce qu'il peut traiter 1 024 octets. Puis, une fois qu'il a des ressources disponibles, il signalera qu'il peut traiter 2 048 octets. Si l'émetteur ne lui avait transmis que 1 024, cette augmentation lui indiquera qu'il peut reprendre l'émission. Si l'émetteur envoie plus de données qu'autorisé, le récepteur ferme brutalement la connexion. À noter que les trames de type `CRYPTO` ne sont pas concernées car elles peuvent être nécessaires pour changer les paramètres cryptographiques (RFC 9001, section 4.1.3).

J'ai parlé des connexions QUIC mais pas encore dit comment elles étaient établies. La section 5 le détaille. Comme QUIC tourne sur UDP, un certain nombre de gens n'ont pas compris le rôle d'UDP et croient que QUIC est sans connexion. Mais c'est faux, QUIC impose l'établissement d'une connexion, c'est-à-dire d'un état partagé entre l'initiateur (celui qui sollicite une connexion) et le répondeur. La négociation initiale permettra entre autres de se mettre d'accord sur les paramètres cryptographiques, mais aussi sur le protocole applicatif utilisé (par exemple HTTP). QUIC permet d'envoyer des données dès le premier paquet (ce qu'on nomme le « 0-RTT ») mais rappelez-vous que, dans ce cas, vous n'êtes plus protégé contre les attaques par rejeu. Ce n'est pas grave pour un `GET` HTTP mais cela peut être gênant dans d'autres cas.

Une fonction essentielle aux connexions QUIC est le "*connection ID*". Il s'agit d'un identificateur de la connexion, qui lui permettra notamment de survivre aux changements de connectivité (passage de 4G dehors en WiFi chez soi, par exemple) ou aux fantaisies des routeurs NAT qui peuvent subitement changer les ports utilisés. Quand un paquet QUIC arrive sur une machine, c'est ce "*connection ID*" qui sert à démultiplexer les paquets entrants, et à trouver les paramètres cryptographiques à utiliser pour le déchiffrer. Il n'y a pas qu'un "*connection ID*" mais tout un jeu, car, comme il circule en clair, il pourrait être utilisé pour suivre à la trace un utilisateur. Ainsi, quand une machine change d'adresse IP, la bonne pratique est de se mettre à utiliser un "*connection ID*" qui faisait partie du jeu de départ, mais n'a pas encore été utilisé, afin d'éviter qu'un surveillant ne fasse le rapprochement entre les deux adresses IP. Notez que le jeu de "*connection ID*" négocié au début peut ensuite être agrandi avec des trames `NEW_CONNECTION_ID`.

Dans quel cas un port peut-il changer ? QUIC utilise UDP, pour maximiser les chances de passer à travers les pare-feux et les routeurs NAT. Cela pose des problèmes si le boitier intermédiaire fait des choses bizarres. Par exemple, si un routeur NAT décide de mettre fin à une connexion TCP qui n'a rien envoyé depuis longtemps, il peut génerer un message RST ("*ReSeT*") pour couper la connexion. Rien de tel en UDP, où le routeur NAT va donc simplement supprimer de sa table de correspondance (entre adresses publiques et privées) une entrée. Après cela, les paquets envoyés par la machine externe seront jetés sans notification, ceux envoyés par la machine interne créeront une nouvelle correspondance, avec un port source différent et peut-être même une adresse IP source différente. La machine externe risque donc de ne pas les reconnaitre comme des paquets appartenant à la même connexion QUIC. En raison des systèmes de traduction d'adresses, l'adresse IP source et le port source vus par le pair peuvent changer pendant une même « session ». Pour permettre de reconnaitre une session en cours, QUIC utilise donc le "*connection ID*", un nombre de longueur variable généré au début de la connexion (un dans chaque direction) et présent dans les paquets. (Le "*connection ID*" source n'est pas présent dans tous les paquets.)

Vous avez vu qu'une connexion QUIC peut parfaitement changer d'adresse IP en cours de route. Cela aura certainement des conséquences pour tous les systèmes qui enregistrent les adresses IP comme identificateur d'un dialogue, du journal d'un serveur HTTP (qu'est-ce que Apache va mettre dans son `access_log` ?) aux surveillances de la HADOPI.

Pour limiter les risques qu'une correspondance dans un routeur faisant de la traduction d'adresses n'expire, QUIC dispose de plusieurs moyens de "*keepalive*" comme les trames de type `PING`.

Comment l'application qui utilise QUIC va-t-elle créer des connexions et les utiliser ? La norme QUIC, dans notre RFC, ne spécifie pas d'API. Elle expose juste les services que doit rendre QUIC aux applications, notamment :

— Ouvrir une connexion (si on est initiateur),
— Attendre des demandes de connexions (si on est répondeur),
— Activer les données "*early data*", c'est-à-dire envoyées dès le premier paquet (rappelez-vous que le rejeu est possible donc l'application doit s'assurer que cette première requête est idempotente),
— Configurer certaines valeurs comme le nombre maximal de ruisseaux ou comme la quantité de données qu'on est prêt à recevoir,
— Envoyer des trames `PING`, par exemple pour s'assurer que la connexion reste ouverte,
— Fermer la connexion.

Le RFC ne le spécifie pas, mais il faudra évidemment que QUIC permette à l'application d'envoyer et de recevoir des données.

Il n'y a actuellement qu'une seule version de QUIC, la 1, normalisée dans notre RFC 9000 (cf. section 15). Dans le futur, d'autres versions apparaitront peut-être, et la section 6 du RFC explique comment se fera la future négociation de version (ce qui sera un point délicat car il faudra éviter les attaques par repli). Notez que toute version future devra se conformer aux invariants du RFC 8999, une garantie de ce qu'on trouvera toujours dans QUIC.

Un point important de QUIC est qu'il n'y a pas de mode « en clair ». QUIC est forcément protégé par la cryptographie. L'établissement de la connexion impose donc la négociation de paramètres cryptographiques (section 7). Ces paramètres sont mis dans une trame `CRYPTO` qui fait partie du premier paquet envoyé. QUIC version 1 utilise TLS (RFC 9001). Le serveur est toujours authentifié, le client peut l'être. C'est aussi dans cette négociation cryptographique qu'est choisie l'application, via ALPN (RFC 7301).

Dans le cas courant, quatre paquets sont échangés, `Initial` par chacun des participants, puis `Handshake`. Mais, si le 0-RTT est accepté, des données peuvent être envoyées par l'initiateur dès le premier paquet.

Puisqu'UDP, comme IP, ne protège pas contre l'usurpation d'adresse IP <https://www.bortzmeyer.org/usurpation-adresse-ip.html>, QUIC doit valider les adresses IP utilisées, pour éviter, par exemple, les attaques par réflexion <https://www.bortzmeyer.org/attaques-reflexion.html> (section 8). Si un initiateur contacte un répondeur en disant « mon adresse IP est 2001:db8:dada::1 », il ne faut pas le croire sur parole, et lui envoyer plein de données sans vérification. QUIC doit valider l'adresse IP de son correspondant <https://www.bortzmeyer.org/returnability.html>, et le revalider lorsqu'il change d'adresse IP. À l'établissement de la connexion, c'est la réception du paquet Handshake, proprement chiffré, qui montre que le correspondant a bien reçu notre paquet Initial et a donc bien l'adresse IP qu'il prétend avoir. En cas de changement d'adresse IP, la validation vient du fait que le correspondant utilise un des "*connection ID*" qui avait été échangés précédemment ou d'un test explicite de joignabilité avec les trames PATH_CHALLENGE et PATH_RESPONSE. Sur ces migrations, voir aussi la section 9.

Pour les futures connexions, on utilisera un jeton qui avait été transmis dans une trame NEW_TOKEN et qu'on a stocké localement. (C'est ce qui permet le 0-RTT.) Le RFC ne spécifie pas le format de ce jeton, seule la machine qui l'a créé et qui l'envoie à sa partenaire a besoin de le comprendre (comme pour un "*cookie*"). Le RFC conseille également de n'accepter les jetons qu'une fois (et donc de mémoriser leur usage) pour limiter le risques de rejeu.

Tant que la validation n'a pas été faite, une machine QUIC ne doit pas envoyer plus de trois fois la quantité de données reçue (pour éviter les attaques avec amplification <https://www.bortzmeyer.org/attaques-reflexion.html>). C'est pour cela que le paquet Initial est rempli de manière à atteindre une taille (1 200 octets, exige le RFC) qui garantit que l'autre machine pourra répondre, même si elle a beaucoup à dire.

Une fois qu'on est connectés, on peut s'échanger des données, qui feront l'objet d'accusés de réception de la part du voisin (trames de type ACK). Contrairement au TCP classique, les accusés de réception ne sont pas forcément contigus, comme dans l'extension SACK du RFC 2018. Si l'accusé de réception n'est pas reçu, l'émetteur réémet, comme avec TCP.

Bon, une fois qu'on a ouvert la connexion, et échangé des données, quand on n'a plus rien à dire, que fait-on ? On raccroche. La section 10 du RFC explique comment se terminent les connexions QUIC. Cela peut se produire suite à une inactivité prolongée, suite à une fermeture explicite normale, ou bien avec le cas particulier de la fermeture sans état. Chacun des partenaires peut évidemment estimer que, s'il ne s'est rien passé depuis longtemps, il peut partir. (Cette durée maximale d'attente peut être spécifiée dans les paramètres à l'établissement de la connexion.) Mais on peut aussi raccrocher explicitement à tout moment (par exemple parce que le partenaire n'a pas respecté le protocole QUIC), en envoyant une trame de type CONNECTION_CLOSE. Cela fermera la connexion et, bien sûr, tous ses ruisseaux.

Pour que la trame CONNECTION_CLOSE soit acceptée par l'autre machine, il faut que son émetteur connaisse les paramètres cryptographiques qui permettront de la chiffrer proprement. Mais il y a un cas ennuyeux, celui où une des deux machines a redémarré, tout oublié, et reçoit des paquets d'une ancienne connexion. Comment dire à l'autre machine d'arrêter d'en envoyer ? Avec TCP, on envoie un paquet RST ("*ReSeT*") et c'est bon. Mais cette simplicité est dangereuse car elle permet également à un tiers de faire des attaques par déni de service en envoyant des « faux » paquets RST. Des censeurs ou des FAI voulant bloquer du trafic pair-à-pair ont déjà pratiqué ce genre d'attaque. La solution QUIC à ce double problème est la fermeture sans état ("*stateless reset*"). Cela repose sur l'envoi préalable d'un jeton (cela peut se faire via un paramètre lors de l'établissement de la connexion, ou via une trame NEW_CONNECTION_ID). Pour pouvoir utiliser ces jetons, il faudra donc les stocker, mais il ne sera pas nécessaire d'avoir les paramètres cryptographiques : on ne chiffre pas le paquet de fermeture sans état, il est juste authentifié (par le jeton). Si la perte de mémoire est totale (jeton stocké en mémoire non stable, et perdu), il ne reste plus que les délais de garde pour mettre fin à la connexion. Évidemment, le jeton

ne peut être utilisé qu'une fois, puisqu'un surveillant a pu le copier. Notez que les détails de ce paquet de fermeture sans état sont soigneusement conçus pour que ce paquet soit indistinguable d'un paquet QUIC « normal ».

Dans un monde idéal, tout fonctionnera comme écrit dans le RFC. Mais, dans la réalité, des machines ne vont pas suivre le protocole et vont faire des choses anormales. La section 11 du RFC couvre la gestion d'erreurs dans QUIC. Le problème peut être dans la couche de transport, ou dans l'application (et, dans ce cas, il peut être limité à un seul ruisseau). Lorsque l'erreur est dans la couche transport et qu'elle semble irrattrapable, on ferme la connexion avec une trame `CONNECTION_CLOSE`. Si le problème ne touche qu'un seul ruisseau, on peut se contenter d'une trame `RESET_STREAM`, qui met fin juste à ce ruisseau.

On a parlé de paquets et de trames. La section 12 précise ces termes :
— Les deux machines s'envoient des datagrammes UDP,
— chaque datagramme contient un ou plusieurs paquets QUIC,
— chaque paquet QUIC peut contenir une ou plusieurs trames. Chacune d'elle a un type, la liste étant dans un registre IANA `<https://www.iana.org/assignments/quic/quic.xml#quic-frame-types>`.

Parmi les paquets, il y a les paquets longs et les paquets courts. Les paquets longs, qui contiennent tous les détails, sont `Initial`, `Handshake`, `0-RTT` et `Retry`. Ce sont surtout ceux qui servent à établir la connexion. Les paquets courts sont le `1-RTT`, qui ne peut être utilisé qu'après l'établissement complet de la connexion, y compris les paramètres cryptographiques. Bref, les paquets longs (plus exactement, à en-tête long) sont plus complets, les paquets courts (à en-tête court) plus efficaces.

Les paquets sont protégés par la cryptographie. (QUIC n'a pas de mode en clair.) Mais attention, elle ne protège pas la totalité du paquet. Ainsi, le "*connection ID*" est en clair puisque c'est lui qui sera utilisé à la destination pour trouver la bonne connexion et donc les bons paramètres cryptographiques pour déchiffrer. (Mais il est protégé en intégrité donc ne peut pas être modifié par un attaquant sans que ce soit détecté.) De même, les paquets `Initial` n'ont de protection que contre la modification, pas contre l'espionnage. En revanche, les paquets qui transportent les données (`0-RTT` et `1-RTT`) sont complètement protégés. Les détails de ce qui est protégé et ce qui ne l'est pas figurent dans le RFC 9001.

La coalescence de plusieurs paquets au sein d'un seul datagramme UDP vise à augmenter les performances en diminuant le nombre de datagrammes à traiter. (J'en profite pour rappeler que la métrique importante pour un chemin sur le réseau n'est pas toujours le nombre d'octets par seconde qui peuvent passer par ce chemin. Parfois, c'est le nombre de datagrammes par seconde qui compte.) Par contre, si un datagramme qui comprend plusieurs paquets, qui eux-mêmes contiennent des trames de données de ruisseaux différents, est perdu, cela va évidemment affecter tous ces ruisseaux.

Les paquets ont un numéro, calculé différemment dans chaque direction, et partant de zéro. Ce numéro est chiffré. Les réémissions d'un paquet perdu utilisent un autre numéro que celui du paquet original, ce qui permet, contrairement à TCP, de distinguer émission et réémission.

Avec QUIC, les datagrammes ne sont jamais fragmentés (en IPv4), on met le bit DF à 1 pour éviter cela. QUIC peut utiliser la PLPMTUD (RFC 8899) pour trouver la MTU du chemin.

Le format exact des paquets est spécifié en section 17. Un paquet long (plus exactement, à en-tête long) se reconnait par son premier bit mis à 1. Il comprend un type (la liste est dans le RFC, elle n'est pas extensible, il n'y a pas de registre IANA), les deux "*connection ID*" et les données, dont la signification dépend du type. Les paquets de type `Initial` comportent entre autres dans ces données un jeton, qui pourra servir, par exemple pour les futurs connexions 0-RTT. Les paquets de type `Handshake`

contiennent des trames de type CRYPTO, qui indiquent les paramètres cryptographiques. Quant aux paquets courts, leur premier bit est à 0, et ils contiennent moins d'information, par exemple, seul le "*connection ID*" de destination est présent, pas celui de la source.

Dans sa partie non chiffrée, le paquet a un bit qui a suscité bien des débats, le "*spin bit*". Comme c'est un peu long à expliquer, ce bit a son propre article <https://www.bortzmeyer.org/quic-spin-bit.html>.

QUIC chiffre beaucoup plus de choses que TCP. Ainsi, pour les paquets à en-tête court, en dehors du "*connection ID*" et de quelques bits dont le "*spin bit*", rien n'est exposé. Par exemple, les accusés de réception sont chiffrés et on ne peut donc pas les observer. Le but est bien de diminuer la vue offerte au réseau (RFC 8546), alors que TCP expose tout (les accusés de réception, les estampilles temporelles, etc). QUIC chiffre tellement qu'il n'existe aucun moyen fiable, en observant le trafic, de voir ce qui est du QUIC et ce qui n'en est pas. (Certaines personnes avaient réclamé, au nom de la nécessité de surveillance, que QUIC se signale explicitement .)

Les différents types de trames sont tous listés en section 19. Il y a notamment :

— PADDING qui permet de remplir les paquets pour rendre plus difficile la surveillance,
— PING qui permet de garder une connexion ouverte, ou de vérifier que la machine distante répond (il n'y a pas de PONG, c'est l'accusé de réception de la trame qui en tiendra lieu),
— ACK, les accusés de réception, qui indiquent les intervalles de numéros de paquets reçus,
— CRYPTO, les paramètres cryptographiques de la connexion,
— STREAM, qui contiennent les données, **et** créent les ruisseaux ; envoyer une trame de type STREAM suffit, s'il n'est pas déjà créé, à créer le ruisseau correspondant (ils sont identifiés par un numéro contenu dans cette trame),
— CONNECTION_CLOSE, pour mettre fin à la connexion.

Les types de trame figurent dans un registre IANA <https://www.iana.org/assignments/quic/quic.xml#quic-frame-types>. On notera que l'encodage des trames n'est pas auto-descriptif : on ne peut comprendre une trame que si on connait son type. C'est plus rapide, mais moins souple et cela veut dire que, si on introduit de nouveaux types de trame, il faudra utiliser des paramètres au moment de l'ouverture de la connexion pour être sûr que l'autre machine comprenne ce type.

Bon, les codes d'erreur, désormais (section 20 du RFC). La liste complète est dans un registre IANA <https://www.iana.org/assignments/quic/quic.xml#quic-transport-error-codes>, je ne vais pas la reprendre ici. Notons quand même le code d'erreur NO_ERROR qui signifie qu'il n'y a pas eu de problème. Il est utilisé lorsqu'on ferme une connexion sans que pour autant quelque chose de mal se soit produit.

Si vous voulez une vision plus concrète de QUIC, vous pouvez regarder mon article d'analyse d'une connexion QUIC <https://www.bortzmeyer.org/quic-demo.html>.

L'une des principales motivations de QUIC est la sécurité, et il est donc logique qu'il y ait une longue section 21 consacrée à l'analyse détaillée de la sécurité de QUIC. D'abord, quel est le modèle de menace ? C'est celui du RFC 3552. En deux mots : on ne fait pas confiance au réseau, tout intermédiaire entre deux machines qui communiquent peut être malveillant. Il y a trois sortes d'attaquants : les attaquants passifs (qui ne peuvent qu'écouter), les attaquants actifs situés sur le chemin (et qui peuvent donc écouter et écrire) et les attaquants actifs non situés sur le chemin, qui peuvent écrire mais en aveugle. Voyons maintenant les attaques possibles.

La poignée de mains initiale est protégée par TLS. La sécurité de QUIC dépend donc de celle de TLS.

Les attaques par réflexion <https://www.bortzmeyer.org/attaques-reflexion.html>, surtout dangereuses quand elles se combinent avec une amplification <https://www.bortzmeyer.org/amplification-tcp.html> sont gênées, sinon complètement empêchées, par la validation des adresses IP. QUIC ne transmet pas plus de trois fois le volume de données à une adresse IP non validée. Ce point n'a pas forcément été compris par tous, et certains ont paniqué à la simple mention de l'utilisation d'UDP. C'est par exemple le cas de cet article <https://blog.nexusguard.com/could-quic-turn-into-the-next-most-prevalent-amplification-attack-vector>, qui est surtout du FUD d'un vendeur.

Du fait du chiffrement, même un attaquant actif qui serait sur le chemin et pourrait donc observer les paquets, ne peut pas injecter de faux paquets ou, plus exactement, ne peut pas espérer qu'ils seront acceptés (puisque l'attaquant ne connait pas la clé de chiffrement). L'attaquant actif qui n'est pas sur le chemin, et doit donc opérer en aveugle, est évidemment encore plus impuissant. (Avec TCP, l'attaquant actif situé sur le chemin peut insérer des paquets qui seront acceptés. Des précautions décrites dans le RFC 5961 permettent à TCP de ne pas être trop vulnérable à l'attaquant aveugle.)

La possibilité de migration des connexions QUIC (changement de port et/ou d'adresse IP) apporte évidemment de nouveaux risques. La validation du chemin doit être refaite lors de ces changements, autrement, un méchant partenaire QUIC pourrait vous rediriger vers une machine innocente.

Bien sûr, les attaques par déni de service restent possibles. Ainsi, un attaquant actif sur le chemin qui a la possibilité de modifier les paquets peut tout simplement les corrompre de façon à ce qu'ils soient rejetés. Mais il y a aussi des attaques par déni de service plus subtiles. L'attaque dite Slowloris vise ainsi à épuiser une autre machine en ouvrant beaucoup de connexions qui ne seront pas utilisées. Un serveur utilisant QUIC doit donc se méfier et, par exemple, limiter le nombre de connexions par client. Le méchant client peut aussi ouvrir, non pas un grand nombre de connexions mais un grand nombre de ruisseaux. C'est d'autant plus facile que d'ouvrir le ruisseau de numéro N (ce qui ne nécessite qu'une seule trame de type `STREAM`) ouvre tous les ruisseaux jusqu'à N, dans la limite indiquée dans les paramètres de transport.

On a vu que dans certains cas, une machine QUIC n'a plus les paramètres qui lui permettent de fermer proprement une connexion (par exemple parce qu'elle a redémarré) et doit donc utiliser la fermeture sans état ("*stateless reset*"). Dans certains cas, un attaquant peut mettre la main sur un jeton qu'il utilisera ensuite pour fermer une connexion.

Le chiffrement, comme toute technique de sécurité, a ses limites. Ainsi, il n'empêche pas l'analyse de trafic (reconnaitre le fichier récupéré à sa taille, par exemple). D'où les trames de type `PADDING` pour gêner cette attaque.

QUIC a plusieurs registres à l'IANA. Si vous voulez ajouter des valeurs à ces registres, leur politique (cf. RFC 8126) est :

— Pour les ajouts provisoires, ce sera « Examen par un expert », et le RFC donne l'instruction d'être assez libéral,
— Pour les ajouts permanents, il faudra écrire une spécification (« Spécification nécessaire »). Là aussi, le RFC demande d'être assez libéral, les registres ne manquent pas de place. Par exemple, les paramètres de transport <https://www.iana.org/assignments/quic/quic.xml#quic-transpor jouissent de 62 bits.
— Pour les types de trame <https://www.iana.org/assignments/quic/quic.xml#quic-frame-types si la majorité de l'espace prévu suit cette politique « Spécification nécessaire », une petite partie est reservée pour enregistrement via une politique plus stricte, « Action de normalisation ».

L'annexe A de notre RFC contient du pseudo-code pour quelques algorithmes utiles à la mise en œuvre de QUIC. Par exemple, QUIC, contrairement à la plupart des protocoles IETF, a plusieurs champs de taille variable. Les encoder et décoder efficacement nécessite des algorithmes astucieux, suggérés dans cette annexe.

Il existe d'ores et déjà de nombreuses mises en œuvre de QUIC, l'écriture de ce RFC ayant été faite en parallèle avec d'innombrables hackathons et tests d'interopérabilité. Je vous renvoie à la page du groupe de travail <https://github.com/quicwg/base-drafts/wiki/Implementations>, qui indique également des serveurs QUIC publics. Elles sont dans des langages de programmation très différents, par exemple celle de Cloudflare, Quiche <https://github.com/cloudflare/quiche>, est en Rust. À ma connaissance, toutes tournent en espace utilisateur mais ce n'est pas obligatoire, QUIC pourrait parfaitement être intégré dans le noyau du système d'exploitation. Une autre bonne source pour la liste des mises en œuvre de QUIC est le Wikipédia anglophone. Notez que le navigateur Web libre Firefox a désormais QUIC <https://hacks.mozilla.org/2021/04/quic-and-http-3-support-now-in-firefox>.

Quelques lectures pour aller plus loin :

— Une très bonne explication de QUIC, avec des dessins très parlants à l'Université catholique de Louvain <https://inl.info.ucl.ac.be/quic-tutorial.html> (en anglais).
— Autre très bon article d'introduction à QUIC et à ses choix principaux, celui de Cloudflare <https://blog.cloudflare.com/the-road-to-quic/>.
— Beaucoup d'articles sur QUIC ne mentionnent que ses avantages. Cet article très détaillé <https://calendar.perfplanet.com/2018/quic-and-http-3-too-big-to-fail/> expose honnêtement les limites de QUIC et les problèmes qu'il pourrait rencontrer.
— QUIC soulève des problèmes politiques intéressants comme, par exemple, « vu sa complexité, peut-on encore innover s'il faut les moyens de Google ? » En effet, qui d'autre aurait pu développer et surtout pousser QUIC et faire qu'il soit déployé ? Ces problèmes stratégiques se posent à beaucoup d'endroits dans le Web, vu son importance dans nos vies. Ils sont discutés (avec d'autres) dans l'"*Internet-Draft*" draft-martini-hrpc-quichr (dont je remercie d'ailleurs les auteurs, qui m'ont bien aidé à comprendre QUIC).
— Sur l'histoire du développement de QUIC à l'IETF et notamment sur les différents choix effectués, les décisions prises, et leurs raisons, je recommande l'article de Fastly <https://www.fastly.com/blog/maturing-of-quic> (par un des auteurs de ce RFC).
— Sur les performances, je recommande « "*Taking a long look at QUIC : an approach for rigorous evaluation of rapidly evolving transport protocols*" <https://conferences.sigcomm.org/imc/2017/papers/imc17-final39.pdf> » de Arash Molavi Kakhki, Samuel Jero et David Choffnes , qui analysait la version Google de QUIC, avec plein de mesures concrètes (leur code est en ligne <https://arashmolavi.github.io/quic/>).
— Toujours question performance, Uber a ajouté QUIC dans son application et détaille les résultats obtenus <https://eng.uber.com/employing-quic-protocol/>.
— Sur la question de l'exposition de données aux machines intermédiaires situées sur le chemin (la « vue depuis le réseau » du RFC 8546), l'article « "*A Path Layer for the Internet : Enabling Network Operations on Encrypted Protocols*" <https://nsg.ee.ethz.ch/fileadmin/user_upload/CNSM_2017.pdf> » explore le problème et propose des solutions, notamment dans le contexte de QUIC.