RFC 9293 : Transmission Control Protocol (TCP)

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 20 août 2022

Date de publication du RFC : Août 2022

https://www.bortzmeyer.org/9293.html

Le protocole de transport TCP est l'un des piliers de l'Internet. La suite des protocoles Internet est d'ailleurs souvent appelée TCP/IP, du nom de ses deux principaux protocoles. Ce nouveau RFC est la norme technique de TCP, remplaçant l'antique RFC 793 [1], qui était vieux de plus de quarante ans, plus vieux que la plupart des lecteurices de ce blog. Il était temps de réviser et de rassembler en un seul endroit tous les détails sur TCP.

TCP est donc un protocole de transport. Rappelons brièvement ce qu'est un protocole de transport et à quoi il sert. L'Internet achemine des paquets IP de machine en machine. IP ne garantit pas leur arrivée : les paquets peuvent être perdus (par exemple parce qu'un routeur n'avait plus de place dans ses files d'attente), peuvent être dupliqués, un paquet peut passer devant un autre, pourtant envoyé avant, etc. Pour la grande majorité des applications, ce n'est pas acceptable. La couche de transport est donc chargée de remédier à cela. Première (en partant du bas) couche à être de bout en bout (les routeurs et autres équipements intermédiaires n'y participent pas, ou plus exactement ne devraient pas y participer), elle se charge de suivre les paquets, de les remettre dans l'ordre, et de demander aux émetteurs de ré-émettre si un paquet s'est perdu. C'est donc un rôle crucial, puisqu'elle évite aux applications de devoir gérer ces opérations très complexes. (Certaines applications, par exemple de vidéo, n'ont pas forcément besoin que chaque paquet arrive, et n'utilisent pas TCP.) Le RFC 8095 contient une comparaison détaillée des protocoles de transport de l'IETF.

Avant de décrire TCP, tel que spécifié dans ce RFC 9293, un petit mot sur les normes techniques de l'Internet. TCP avait originellement été normalisé dans le RFC 761 en 1980 (les couches 3 - IP et 4 étaient autrefois mêlées en une seule). Comme souvent sur l'Internet, le protocole existait déjà avant sa normalisation et était utilisé. La norme avait été rapidement révisée dans le RFC 793 en 1981. Depuis, il n'y avait pas eu de révision générale, et les RFC s'étaient accumulés. Comprendre TCP nécessitait donc de lire le RFC 793 puis d'enchainer sur plusieurs autres. Un RFC, le RFC 7414, avait même été écrit dans

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc793.txt

le seul but de fournir un guide de tous ces RFC à lire. Et il fallait également tenir compte de l'accumulation d'errata <https://www.rfc-editor.org/errata_search.php?rfc=793&rec_status=15&presentation=table>. Désormais, la situation est bien plus simple, tout TCP a été consolidé dans un RFC central, notre RFC 9293. (Notez que ce travail de consolidation a été aussi fait pour HTTP et SMTP, qui ne sont plus décrits par les RFC d'origine mais par des versions à jour, mais pas pour le DNS, qui reste le principal ancien protocole qui aurait bien besoin d'une remise à jour complète des documents qui le spécifient.) Ne vous faites toutefois pas d'illusion : malgré ses 114 pages, ce RFC 9293 ne couvre pas tout, et la lecture d'autres RFC reste nécessaire. Notamment, TCP offre parfois des choix multiples (par exemple pour les algorithmes de contrôle de la congestion, une partie cruciale de TCP), qui ne sont pas décrits intégralement dans la norme de base. Voyez par exemple le RFC 7323, qui n'a pas été intégré dans le RFC de base, et reste une extension optionnelle.

Attaquons-nous maintenant à TCP. (Bien sûr, ce sera une présentation très sommaire, TCP est forcément plus compliqué que cela.) Son rôle est de fournir aux applications un flux d'octets fiable et ordonné : les octets que j'envoie à une autre machine arriveront dans l'ordre et arriveront tous (ou bien TCP signalera que la communication est impossible, par exemple en cas de coupure du câble). TCP découpe les données en **segments**, chacun voyageant dans un datagramme IP. Chaque octet est numéroté (attention : ce sont bien les octets et pas les segments qui sont numérotés) et cela permet de remettre dans l'ordre les paquets, et de détecter les pertes. En cas de perte, on retransmet. Le flux d'octets est bidirectionnel, la même connexion TCP permet de transmettre dans les deux sens. TCP nécessite l'établissement d'une connexion, donc la mémorisation d'un état, par les deux pairs qui communiquent. Outre les tâches de fiabilisation des données (remettre les octets dans l'ordre, détecter et récupérer les pertes), TCP fournit du démultiplexage, grâce aux numéros de port. Ils permettent d'avoir plusieurs connexions TCP entre deux machines, qui sont distinguées par ces numéros de port (un pour la source et un pour la destination) dans l'en-tête TCP. Le bon fonctionnement de TCP nécessite d'édicter un certain nombre de règles et le RFC les indique avec "*MUST-n*" où N est un entier. Par exemple, "*MUST-2*" et "*MUST-3*" indiqueront que la somme de contrôle est obligatoire, aussi bien en envoi qu'en vérification à la réception.

Après ces concepts généraux, la section 3 de notre RFC rentre dans les détails concrets, que je résume ici. D'abord, le format de cet en-tête que TCP ajoute derrière l'en-tête IP et devant les données du segment. Il inclut les numéros de port, le numéro de séquence (le rang du premier octet dans les données), celui de l'accusé de réception (le rang du prochain octet attendu), un certain nombre de booléens ("*flags*", ou bits de contrôle, qui indiquent, par exemple, s'il s'agit d'une connexion déjà établie, ou pas encore, ou la fin d'une connexion), la taille de la fenêtre (combien d'octets peuvent être envoyés sans accusé de réception, ce qui permet d'éviter de surcharger le récepteur), une somme de contrôle, et des options, de taille variable (l'en-tête contient un champ indiquant à partir de quand commencent les données). Une option de base est MSS ("*Maximum Segment Size*") qui indique quelle taille de segment on peut gérer. Il existe d'autres options <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xml#tcp-parameters-1> comme les accusés de réception sélectifs du RFC 2018 ou comme le facteur multiplicatif de la taille de fenêtre du RFC 7323.

Avant d'expliquer la dynamique de TCP, le RFC définit quelques variables indispensables, qui font partie de l'état de la connexion TCP. Pour l'envoi de données, ce sont par exemple `SND.UNA` (ces noms sont là pour comprendre le RFC, mais un programme qui met en œuvre TCP peut évidemment nommer ces variables comme il veut), qui désigne le numéro de séquence du début des données envoyées, mais qui n'ont pas encore fait l'objet d'un accusé de réception, ou `SND.NXT`, le numéro de séquence du début des données pas encore envoyées ou encore `SND.WND`, la taille de la fenêtre d'envoi. Ainsi, on ne peut pas envoyer d'octets dont le rang serait supérieur à `SND.UNA + SND.WND`, il faut attendre des accusés de réception (qui vont incrémenter `SND.UNA`) ou une augmentation de la taille de la fenêtre. Pour la réception, on a `RCV.NXT`, le numéro de séquence des prochains paquets si tout est normal, et `RCV.WND`, la taille de la fenêtre de réception.

TCP est un protocole à état et il a donc une machine à états, décrite en section 3.3.2. Parmi les états, `LISTEN`, qui indique un TCP en train d'attendre un éventuel pair, `ESTAB` (ou `ESTABLISHED`), où TCP

envoie et reçoit des données, et plusieurs états qui apparaissent lors de l'ouverture ou de la fermeture d'une connexion.

Et les numéros de séquence ? Un point important de TCP est qu'on ne numérote pas les segments mais les octets. Chaque octet a son numéro et les accusés de réception référencent ces numéros. On n'accuse évidemment pas réception de chaque octet individuel. Les accusés de réception sont cumulatifs ; ils désignent un octet et, implicitement, tous les octets précédents. Quand un récepteur prévient qu'il a bien reçu l'octet N, cela veut dire que tous ceux avant N ont aussi été reçus.

Les numéros ne partent pas de 1 (ni de zéro), ils sont relatifs à un ISN ("*Initial Sequence Number*", rappelez-vous qu'il y a un glossaire, en section 4 du RFC) qui est choisi aléatoirement (pour des raisons de sécurité, cf. RFC 6528) au démarrage de la session. Des logiciels comme Wireshark savent cela et peuvent calculer puis afficher des numéros de séquence qui partent de 1, pour faciliter la lecture.

Les numéros de séquence sont stockés sur 32 bits et il n'y a donc que quatre milliards (et quelques) valeurs possibles. C'était énorme quand TCP a été créé mais cela devient de plus en plus petit, relativement aux capacités des réseaux modernes. Comme TCP repart de zéro quand il atteint le numéro de séquence maximum, il y a un risque qu'un « vieux » paquet soit considéré comme récent. À 1 Gb/s, cela ne pouvait se produire qu'au bout de 34 secondes. Mais à 100 Gb/s, il suffit d'un tiers de seconde ! Les extensions du RFC 7323 deviennent alors indispensables.

Tout le monde sait bien que TCP est un protocole orienté connexion. Cela veut dire qu'il faut pouvoir créer, puis supprimer, les connexions. La section 3.5 de notre RFC décrit cet établissement de connexion, via la fameuse « triple poignée de mains ». En gros, un des TCP (TCP n'est pas client-serveur, et la triple poignée de mains marche également quand les deux machines démarrent la connexion presque en même temps) envoie un paquet ayant l'option SYN (pour "*synchronization*"), le second répond avec un paquet ayant les options ACK (accusé de réception du SYN) et SYN, le premier TCP accuse réception. Avec trois paquets, les deux machines sont désormais d'accord qu'elles sont connectées. En d'autres termes (ici, machine A commence) :

— Machine A ⇒ Machine B : je veux te parler,
— B ⇒ A : j'ai noté que tu voulais me parler et je veux te parler aussi,
— A ⇒ B : j'ai noté que tu voulais me parler.

Pour mettre fin à la connexion, un des deux TCP envoie un paquet ayant l'option FIN (pour "*finish*"), auquel l'autre répond de même.

Conceptuellement, TCP gère un flux d'octets sans séparation. TCP n'a pas la notion de message. Si une application envoie (en appelant `print`, `write` ou autre fonction du langage de programmation utilisé) les octets "AB" puis "CD", TCP transmettra puis livrera à l'application distante les quatre octets "ABCD" dans l'ordre, sans indiquer qu'il y avait eu deux opérations d'écriture sur le réseau. Mais, pour envoyer les données, TCP doit les segmenter en paquets IP distincts. La fragmentation IP n'étant pas bonne pour les performances, et étant peu fiable de toute façon, l'idéal est que ces paquets IP aient une taille juste inférieure à la MTU du chemin. Cela nécessite de découvrir cette MTU (section 3.7.2), ou bien d'adopter la solution paresseuse de faire des paquets de 1 280 octets (la MTU minimum, en IPv6). Comme la solution paresseuse ne serait pas optimale en performances, les mises en œuvre de TCP essaient toutes de découvrir la MTU du chemin. Cela peut se faire avec la méthode PMTUD, décrite dans les RFC 1191 et RFC 8201, mais qui a l'inconvénient de nécessiter qu'ICMP ne soit pas bloqué. Or, beaucoup de "*middleboxes*" programmées avec les pieds, ou configurées par des incompétents, bloquent ICMP, empêchant le fonctionnement de PMTUD. Une alternative est PLPMTUD, normalisée dans le RFC 4821, qui ne dépend plus d'ICMP.

Une fois la connexion établie, on peut envoyer des données. TCP va les couper en segments, chaque segment voyageant dans un paquet IP.

Une mise en œuvre de TCP n'envoie pas forcément immédiatement les octets qu'on lui a confiés. Envoyer des petits paquets n'est pas efficace : s'il y a peu de données, les en-têtes du paquet forment la majorité du trafic et, de toute façon, le goulet d'étranglement dans le réseau n'est pas forcément lié au nombre d'octets, il peut l'être au nombre de paquets. Un des moyens de diminuer le nombre de paquets contenant peu d'octets est l'algorithme de Nagle. Le principe est simple : quand TCP reçoit un petit nombre d'octets à envoyer, il ne les transmet pas tout de suite mais attend un peu pour voir si l'application ne va pas lui confier des octets supplémentaires. Décrit dans le RFC 896, et recommandé dans le RFC 1122, cet algorithme est aujourd'hui présent dans la plupart des mises en œuvres de TCP. Notre RFC 9293 le recommande, mais en ajoutant que les applications doivent pouvoir le débrayer. C'est notamment indispensable pour les applications interactives, qui n'aiment pas les délais, même courts. (Sur Unix, cela se fait avec l'option `TCP_NODELAY` passée à `setsockopt`.)

Un des rôles les plus importants de TCP est de lutter contre la congestion. Il ne faut pas envoyer le plus d'octets possible, ou alors on risque d'écrouler le réseau (RFC 2914). Plusieurs mécanismes doivent être déployés : démarrer doucement (au début de la connexion, TCP ne connait pas le débit que peut supporter le réseau et doit donc être prudent), se calmer rapidement si on constate que des paquets sont perdus (cela peut être un signe de congestion en aval), etc. Ces mesures pour éviter la congestion sont absolument obligatoires (comme tous les biens communs, l'Internet est vulnérables aux parasites, un TCP égoïste qui ne déploierait pas ces mesures anti-congestion serait un mauvais citoyen du réseau). Elles sont décrites plus précisément dans les RFC 1122, RFC 2581, RFC 5681 et RFC 6298. Notez qu'ils énoncent des principes, mais pas forcément les algorithmes exacts à utiliser, ceux-ci pouvant évoluer (et ils l'ont fait depuis la sortie du RFC 793, voir les RFC 5033 et RFC 8961).

Un point qui surprend souvent les programmeur·ses qui écrivent des applications utilisant TCP est que rien n'indique qu'une connexion TCP est coupée, par exemple si un câble est sectionné. C'est seulement lorsqu'on essaie d'écrire qu'on l'apprendra (ce comportement est une conséquence de la nature sans connexion d'IP ; pourquoi prévenir d'une coupure alors qu'IP va peut-être trouver un autre chemin ?). Si on veut être mis au courant tout de suite, on peut utiliser des "*keep-alives*", des segments vides qui seront envoyés de temps en temps juste pour voir s'ils arrivent. Cette pratique est contestée (entre autre parce qu'elle peut mener à abandonner une connexion qui n'est que temporairement coupée, et aussi parce que, même si les "*keep-alives*" passent, cela ne garantit pas que la prochaine écriture de données passera). Notre RFC demande donc que les "*keep-alives*" soient désactivés par défaut, et que l'application puisse contrôler leur activation (sur Unix, c'est l'option `SO_KEEPALIVE` à utiliser avec `setsockopt`).

Pendant les quarante ans écoulés depuis la sortie du RFC 793, bien des choses ont changé. Des options ont été ajoutées mais on a vu aussi certaines options qui semblaient être des bonnes idées être finalement abandonnées. C'est le cas du mécanisme de données urgentes, qui n'a jamais vraiment marché comme espéré. Une mise en œuvre actuelle de TCP doit toujours le gérer, pour pouvoir parler avec les anciennes, mais on ne peut pas compter dessus (RFC 6093).

On a parlé plusieurs fois de l'utilisation de TCP par les applications. Pour cela, il faut que TCP offre une API à ces applications. Notre RFC ne décrit pas d'API concrète (cela dépend de toute façon du langage de programmation, et sans doute du système d'exploitation). Il se contente d'une description fonctionnelle de l'interface : les services qu'elle doit offrir à l'application. On y trouve :

— L'ouverture de la connexion, en indiquant l'adresse et le port distant, mais aussi l'adresse locale (MUST-43), pour pouvoir choisir l'adresse si la machine en a plusieurs.
— L'envoi de données, dont TCP garantit qu'elles arriveront toutes et dans l'ordre.
— La lecture de données. Rappelez-vous que TCP fournit un flot continu d'octets, deux envois de données feront peut-être une seule lecture ou bien un envoi deux lectures. C'est une des erreurs les plus courante des programmeurs débutants que de penser que s'ils font un `write` d'un côté, un `read` de l'autre lira tout ce que le `write` a écrit et seulement cela.
— La fermeture propre (avec envoi, et réémission si nécessaire, des données écrites) de la connexion.

— Et quelques autres services auxiliaires que je ne détaille pas ici (voir la section 3.9.1).

Rappelez-vous que l'API « socket » n'est qu'un des moyens pour l'application de parler à TCP. Et, à propos de cette API, notez que le RFC utilise le terme de "*socket*" avec un sens très différent (cf. le glossaire en section 4).

Et l'autre interface, « sous » TCP, l'interface avec la couche réseau (section 3.9.2) ? TCP peut fonctionner avec des couches réseau variées mais, en pratique, c'est toujours IP, dans une de ses deux versions, v4 ou v6. Un des points délicats est le traitement des messages ICMP que TCP peut recevoir. Au minimum, ils doivent être assignés à une connexion existante. Mais cela ne veut pas dire qu'il faut systématiquement agir lors de la réception d'un de ces messages (RFC 5461). Par exemple, il faut ignorer les messages ICMP de répression de la source (RFC 6633). Et il ne faut pas fermer la connexion juste parce qu'on a reçu une erreur ICMP (MUST-56), car elles peuvent être temporaires.

TCP étant un protocole orienté connexion, la connexion va avoir un **état**, représenté par un automate fini. Ainsi, l'ouverture d'une connexion par le premier TCP va le faire passer par les états CLOSED, SYN-SENT puis enfin ESTABLISHED. Récupérée sur Wikimedia Commons <https://commons.wikimedia.org/wiki/File:TCP_state_diagram.jpg>, voici une représentation de cet automate :

La section 5 de notre RFC décrit les changements depuis le RFC 793. TCP reste le même et des TCP d'« avant » peuvent interopérer avec ceux qui suivent notre RFC récent. Les principaux changements sont :

— La résolution d'un grand nombre d'errata <https://www.rfc-editor.org/errata_search.php?rfc=793&presentation=table> accumulés depuis quarante ans.
— La « démobilisation » du mécanisme de données urgentes (cf. RFC 6093).
— L'incorporation d'une bonne partie des RFC comme le RFC 1011 ou le RFC 1122.

Le RFC 793 avait été écrit par Jon Postel (comme tant de RFC de cette époque), et notre nouveau RFC 9293 lui rend hommage, en estimant que la longue durée de vie du RFC 793 montre sa qualité.

En application des changements de ce nouveau RFC sur TCP, un registre IANA, celui des bits de l'entête <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xml#tcp-header-flags> comme URG (dont on a vu qu'il était désormais démobilisé), a été mis à jour.

La section 7 de notre nouveau RFC 9293 n'avait pas d'équivalent dans l'ancien RFC 793. Elle concerne la sécurité de TCP. TCP, par lui-même, fournit peu de sécurité (il protège quand même un peu contre l'usurpation d'adresse IP <https://www.bortzmeyer.org/usurpation-adresse-ip.html> et l'injection de trafic, surtout contre un attaquant qui n'est pas sur le chemin, cf. RFC 5961). Par exemple, il ne fournit pas de services cryptographiques, donc pas de confidentialité ou de vraie garantie d'intégrité. (Son concurrent QUIC <https://www.bortzmeyer.org/quic.html>, lui, intègre systématiquement la cryptographie.) Si on veut davantage de sécurité pour TCP, il faut mettre IPsec en dessous de TCP ou bien TLS au-dessus. TCP a aussi des mécanismes de sécurité qui lui sont propres, comme AO, normalisé dans le RFC 5925 (mais très peu déployé en pratique). Autre expérience qui n'a pas été un succès, tcpcrypt (RFC 8548).

Il reste des attaques contre lesquelles la cryptographie ne fournit pas vraiment de solution. C'est le cas du "*SYN flooding*", ou d'autres attaques par déni de service.

Mëme si on chiffre les données applicatives avec TLS, TCP expose quand même au réseau tout son fonctionnement. Cette vue depuis le réseau (RFC 8546) soulève plusieurs problèmes (elle peut par exemple faciliter des actions contraires à la neutralité du réseau <https://www.bortzmeyer.org/neutralite.html>) et c'est pour cela que le plus récent QUIC <https://www.bortzmeyer.org/quic.html> masque une grande partie de son fonctionnement (ce qui n'a pas été sans entrainer des polémiques <https://www.bortzmeyer.org/quic-spin-bit.html>). Toujours côté vie privée,

le comportement des mises en œuvre de TCP est suffisamment différent pour qu'il soit possible dans certains cas d'identifier le système d'exploitation d'une machine à distance (ce n'est pas forcément grave, mais il faut le savoir).

Pour les programmeurs et programmeuses, l'annexe A du RFC contient différentes observations utiles pour la mise en œuvre concrète de TCP. Par exemple, cette annexe explique que la validation des numéros de séquence des segments TCP entrants, validation qui est nécessaire pour empêcher un grand nombre d'attaques, peut rejeter des segments légitimes. Ce problème n'a pas de solution idéale.

Un autre point intéressant concerne l'algorithme de Nagle. Cet algorithme représente un compromis entre la réactivité de l'application (qui nécessite d'envoyer les données le plus vite possibles) et l'optimisation de l'occupation du réseau (qui justifie d'attendre un peu pour voir si on ne va pas recevoir de nouvelles données à envoyer). Une modification de l'algorithme de Nagle, décrite dans le document `draft-minshall-nagle` peut rendre le choix moins douloureux.

Enfin, l'annexe B résume sous forme d'un tableau quelles sont les parties de la norme TCP qui **doivent** être mises en œuvre et quelles sont celles qu'on peut remettre à plus tard.

Si vous voulez sérieusement apprendre TCP, vous pouvez bien sûr lire le RFC en entier, mais il est sans doute plus sage de commencer par un bon livre comme le CNP3 <`https://www.bortzmeyer.org/cnp3.html`>.

Un pcap d'une connexion TCP typique est utilisé ici pour illustrer le fonctionnement de TCP (fichier (en ligne sur `https://www.bortzmeyer.org/files/tcp-typique.pcap`)). Vu par tshark, cela donne :

```
8.445772 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 TCP 94 33672 → 443 [SYN] Seq=0 Win=64800 Len=0 MSS
8.445856 2605:4500:2:245b::42 → 2a01:4f8:c010:7eb7::106 TCP 94 443 → 33672 [SYN, ACK] Seq=0 Ack=1 Win=6426
8.534636 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 TCP 86 33672 → 443 [ACK] Seq=1 Ack=1 Win=64896 Len
8.534897 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 SSLv3 236 Client Hello
8.534989 2605:4500:2:245b::42 → 2a01:4f8:c010:7eb7::106 TCP 86 443 → 33672 [ACK] Seq=1 Ack=151 Win=64128 L
8.548101 2605:4500:2:245b::42 → 2a01:4f8:c010:7eb7::106 TLSv1.2 1514 Server Hello
8.548111 2605:4500:2:245b::42 → 2a01:4f8:c010:7eb7::106 TCP 1514 443 → 33672 [PSH, ACK] Seq=1429 Ack=151 W
8.548265 2605:4500:2:245b::42 → 2a01:4f8:c010:7eb7::106 TLSv1.2 887 Certificate, Server Key Exchange, Serve
8.636922 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 TCP 86 33672 → 443 [ACK] Seq=151 Ack=2857 Win=6361
8.636922 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 TCP 86 33672 → 443 [ACK] Seq=151 Ack=3658 Win=6284
8.649899 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 TCP 86 33672 → 443 [FIN, ACK] Seq=151 Ack=3658 Win
8.650160 2605:4500:2:245b::42 → 2a01:4f8:c010:7eb7::106 TCP 86 443 → 33672 [FIN, ACK] Seq=3658 Ack=152 Win
8.739117 2a01:4f8:c010:7eb7::106 → 2605:4500:2:245b::42 TCP 86 33672 → 443 [ACK] Seq=152 Ack=3659 Win=6412
```

Ici, `2a01:4f8:c010:7eb7::106` (une sonde RIPE Atlas <`https://atlas.ripe.net/`>) veut faire du HTTPS (donc, port 443) avec le serveur `2605:4500:2:245b::42`. Tout commence par la triple poignée de mains (les trois premiers paquets, SYN, SYN+ACK, ACK). Une fois la connexion établie, tshark reconnait que les machines font du TLS mais, pour TCP, ce ne sont que des données applicatives (contrairement à QUIC, TCP sépare connexion et chiffrement). On note qu'il n'y a pas eu d'optimisation des accusés de réception. Par exemple, l'accusé de réception du ClientHello TLS voyage dans un paquet séparé, alors qu'il aurait pu être inclus dans la réponse applicative (le ServerHello), probablement parce que celle-ci a mis un peu trop de temps à être envoyée. À la fin de la connexion, chaque machine demande à terminer (FIN). Une analyse automatique plus complète, avec `tshark -V` figure dans (en ligne sur `https://www.bortzmeyer.org/files/tcp-typique.txt`).

Ah, au fait, si vous avez pu lire cet article, c'est certainement grâce à TCP…(Ce blog ne fait pas encore de HTTP sur QUIC…).
