RFC 7413 : TCP Fast Open

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 19 décembre 2014

Date de publication du RFC : Décembre 2014

https://www.bortzmeyer.org/7413.html

Quand on parle de performances sur l'Internet, on se focalise souvent exclusivement sur la capacité <https://www.bortzmeyer.org/capacite.html> du réseau. C'est, par exemple, l'argument quasi-exclusif des FAI : « Avec la fibre Machin, passez à la vitesse supérieure, XXX Mb/s ». Mais un autre facteur de la qualité de l'expérience utilisateur est la latence <https://www.bortzmeyer.org/latence.html>, à savoir le temps qu'il faut attendre pour recevoir une réponse. Par exemple, pour accéder à une page Web, avant même d'envoyer le premier octet « utile », il faut réaliser une « triple poignée de mains » (RFC 793 [1], section 3.4) avec le serveur, pour établir la connexion, et cette poignée de mains nécessite pas moins de trois paquets, donc il faudra attendre trois fois le temps d'un aller simple, avant de pouvoir foncer et envoyer les images de chats (ou de loutres). Ce nouveau RFC, encore officiellement expérimental, mais déjà largement déployé, vise à raccourcir ce délai d'ouverture d'une connexion avec le serveur.

L'idée est ancienne (on la trouve par exemple dans le système T/TCP du RFC 1644, et dans les autres systèmes résumés en section 8) mais le problème est plus difficile à résoudre qu'il n'en a l'air, notamment si on veut garder un bon niveau de sécurité (le problème qui a tué T/TCP). L'idée est ancienne, car devoir attendre trois voyages avant de pouvoir envoyer les données contribue sérieusement à la latence d'applications comme le Web. Si le serveur est à 100 ms de distance, on attendra au moins 300 ms avant que des données ne commencent le voyage. Et, contrairement à la capacité, la latence ne s'améliore pas avec le temps et les progrès de l'électronique. Des mesures faites sur Chrome montre que la triple poignée de mains de TCP est responsable de 25 % de la latence moyenne des requêtes HTTP. Il existe des solutions pour certaines applications. Par exemple, pour HTTP, on peut utiliser les connexions persistantes (RFC 7230, section 6.3). L'établissement de ces connexions prendra autant de temps qu'avant mais ce « coût » ne sera payé qu'une fois : les requêtes/réponses ultérieures avec le même serveur HTTP réutiliseront la connexion TCP existante (et, si elles ne viennent pas, la connexion finit par être coupée, par exemple au bout de cinq minutes pour Chrome). Le problème est que ces

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

connexions persistantes ne sont pas assez répandues. Les mesures des auteurs du RFC (Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A., Raghavan, B., « "*TCP Fast Open*" <http://cseweb.ucsd.edu/~ssradhak/Papers/fastopen-conext11.pdf> », "*Proceedings of 7th ACM CoNEXT Conference*", 2011) montrent que 35 % des requêtes nécessitent la création d'une nouvelle connexion TCP. Il serait agréable de pouvoir accélérer ces créations. La triple poignée de mains prend trois paquets, SYN (du client vers le serveur), SYN + ACK (du serveur vers le client) et ACK (du client vers le serveur). Les données ne voyagent qu'après. L'idée de base de TCP Fast Open <http://conferences.sigcomm.org/co-next/2011/papers/1569470463.pdf> (TFO), technique venant de chez Google, est de mettre les données dès le premier paquet SYN du client.

Le plus drôle est que le TCP original n'interdit pas de mettre des données dans le paquet SYN (section 2 de notre RFC). Bien au contraire, c'est explicitement autorisé par la section 3.4 du RFC 793 (« "*Although these examples do not show connection synchronization using data-carrying segments, this is perfectly legitimate [...]*" »). Mais avec une grosse restriction, que les données ne soient délivrées au serveur d'applications qu'une fois la triple poignée de mains terminée, ce qui en supprime l'intérêt. Cette règle a pour but d'éviter que des données soient envoyées deux fois (le paquet SYN peut être perdu et retransmis). Les variantes de TCP qui ont essayé d'optimiser le temps d'ouverture de connexion ont en général choisi de sacrifier de la sécurité, afin de maintenir la sémantique de TCP, ce qui n'est en général pas considéré comme acceptable de nos jours (RFC 7414). TCP Fast Open sacrifie, lui, la non-duplication des données : un serveur Fast Open peut, dans certains cas (par exemple si le serveur a redémarré entre les deux SYN), recevoir deux fois les mêmes données envoyées dans un paquet SYN. Ce n'est pas toujours si dramatique que ça en a l'air. Pour HTTP, le premier paquet de données du client sera sans doute un GET /something HTTP/1.1 et cette requête peut être effectuée deux fois sans conséquences graves. Première chose à retenir, donc, sur Fast Open : il ne convient pas à toutes les applications. TCP Fast Open est juste un compromis. Les systèmes ne doivent donc **pas** l'activer par défaut (les exemples plus loin montrent comment un programme Unix peut activer explicitement Fast Open). La section 4.2 insistera sur ce point.

Et la sécurité ? Le TCP normal présente une vulnérabilité : les paquets SYN n'ayant aucune forme d'authentification, un attaquant peut, en trichant sur son adresse IP, envoyer des paquets SYN sans révéler son identité et ces paquets, s'ils sont assez abondants, peuvent remplir la file d'attente du serveur (attaque dite « "*SYN flood*" »). C'est encore pire avec Fast Open puisque ces requêtes en attente comprennent des données, et peuvent déclencher l'exécution de commandes complexes (GET /horrible-page-dont-la-génération-nécessite-10000-lignes-de-Java-ou-de-PHP HTTP/1.1...) Les techniques du RFC 4987 ne sont en général pas appliquables à Fast Open. C'est pour cela que Fast Open ajoute un composant essentiel : un petit gâteau ("*cookie*") généré par le serveur et que le client devra transmettre pour bénéficier du Fast Open.

La section 3 décrit en termes généraux le protocole. À la première connexion Fast Open d'une machine vers une autre, le client envoie l'option TCP (pour les options TCP, voir la section 3.1 du RFC 793) 34 "*TCP Fast Open Cookie*" (désormais dans le registre IANA <https://www.iana.org/assignments/tcp-parameters/tcp-parameters.xhtml#tcp-parameters-1> mais attention, les mises en œuvre actuelles utilisent souvent la valeur expérimentale précédente) avec un contenu vide. Si le serveur gère Fast Open, il répondra (dans le SYN + ACK) avec un gâteau ("*cookie*"), généré par lui et imprévisible. Dans les connexions TCP ultérieures, le client renverra l'option "*Fast Open Cookie*" avec le même gâteau. Le serveur le reconnaitra alors. Les requêtes SYN comportant ce gâteau pourront inclure des données, elles seront transmises tout de suite aux applications qui le demandent (et on aura alors du beau TCP Fast Open) et le premier paquet de réponse pourra inclure des données (dans les limites données par le RFC 5681). On voit donc que la première connexion entre deux machines ne bénéficiera pas de Fast Open. Mais toutes les suivantes, oui (sur Linux, le noyau se souviendra du gâteau). Si le serveur ne gère pas cette option, il répond par un SYN + ACK sans l'option, informant ainsi le client qu'il ne doit pas compter sur Fast Open.

Voici, vu avec tcpdump, un exemple d'une session TCP Fast Open d'une machine Debian/Linux (version `jessie`) vers Google. Le gâteau (`32a8b7612cb5ea57`) est en mémoire chez le client (les options TCP sont affichées entre crochets, la « nôtre » est `exp-tfo`) :

```
20:10:39.920892 IP (tos 0x0, ttl 64, id 6594, offset 0, flags [DF], proto TCP (6), length 158)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [S], cksum 0x5c7d (incorrect -> 0x9bd5), seq 1779163941:177916
20:10:39.923005 IP (tos 0x0, ttl 57, id 3023, offset 0, flags [none], proto TCP (6), length 52)
    173.194.38.98.80 > 106.186.29.14.53598: Flags [S.], cksum 0xae4c (correct), seq 1775907905, ack 1779164028,
20:10:39.923034 IP (tos 0x0, ttl 64, id 6595, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [.], cksum 0x5c07 (incorrect -> 0x95af), ack 1, win 229, lengt
20:10:39.923462 IP (tos 0x0, ttl 57, id 3024, offset 0, flags [none], proto TCP (6), length 589)
    173.194.38.98.80 > 106.186.29.14.53598: Flags [P.], cksum 0xcdd1 (correct), seq 1:550, ack 1, win 670, lengt
20:10:39.923475 IP (tos 0x0, ttl 57, id 3025, offset 0, flags [none], proto TCP (6), length 40)
    173.194.38.98.80 > 106.186.29.14.53598: Flags [F.], cksum 0x91d0 (correct), seq 550, ack 1, win 670, length
20:10:39.923492 IP (tos 0x0, ttl 64, id 6596, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [.], cksum 0x5c07 (incorrect -> 0x9382), ack 550, win 237, len
20:10:39.923690 IP (tos 0x0, ttl 64, id 6597, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.53598 > 173.194.38.98.80: Flags [R.], cksum 0x5c07 (incorrect -> 0x937d), seq 1, ack 551, win
```

Notez la longueur du premier paquet, 86 octets (une requête HTTP), alors qu'elle est normalement nulle, sans Fast Open.

Il y a en tout sept paquets. Sans Fast Open, la même requête HTTP aurait pris deux paquets de plus :

```
20:11:13.403762 IP (tos 0x0, ttl 64, id 55763, offset 0, flags [DF], proto TCP (6), length 60)
    106.186.29.14.42067 > 173.194.38.96.80: Flags [S], cksum 0x5c19 (incorrect -> 0x858c), seq 720239607, win 29
20:11:13.405827 IP (tos 0x0, ttl 57, id 7042, offset 0, flags [none], proto TCP (6), length 52)
    173.194.38.96.80 > 106.186.29.14.42067: Flags [S.], cksum 0x5792 (correct), seq 687808390, ack 720239608, wi
20:11:13.405857 IP (tos 0x0, ttl 64, id 55764, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.42067 > 173.194.38.96.80: Flags [.], cksum 0x5c05 (incorrect -> 0x3ef5), ack 1, win 229, lengt
20:11:13.405915 IP (tos 0x0, ttl 64, id 55765, offset 0, flags [DF], proto TCP (6), length 126)
    106.186.29.14.42067 > 173.194.38.96.80: Flags [P.], cksum 0x5c5b (incorrect -> 0xaa0c), seq 1:87, ack 1, win
20:11:13.407979 IP (tos 0x0, ttl 57, id 7043, offset 0, flags [none], proto TCP (6), length 40)
    173.194.38.96.80 > 106.186.29.14.42067: Flags [.], cksum 0x3ce6 (correct), ack 87, win 670, length 0
20:11:13.408456 IP (tos 0x0, ttl 57, id 7044, offset 0, flags [none], proto TCP (6), length 589)
    173.194.38.96.80 > 106.186.29.14.42067: Flags [P.], cksum 0x9cce (correct), seq 1:550, ack 87, win 670, leng
20:11:13.408469 IP (tos 0x0, ttl 57, id 7045, offset 0, flags [none], proto TCP (6), length 40)
    173.194.38.96.80 > 106.186.29.14.42067: Flags [F.], cksum 0x3ac0 (correct), seq 550, ack 87, win 670, length
20:11:13.408498 IP (tos 0x0, ttl 64, id 55766, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.42067 > 173.194.38.96.80: Flags [.], cksum 0x5c05 (incorrect -> 0x3c72), ack 550, win 237, len
20:11:13.408720 IP (tos 0x0, ttl 64, id 55767, offset 0, flags [DF], proto TCP (6), length 40)
    106.186.29.14.42067 > 173.194.38.96.80: Flags [R.], cksum 0x5c05 (incorrect -> 0x3c6d), seq 87, ack 551, win
```

Dans cet exemple, le gâteau envoyé par Google était en mémoire. Si ce n'est pas le cas (la machine vient de redémarrer, par exemple), la première requête Fast Open va être une triple poignée de mains classique, les bénéfices de Fast Open ne venant qu'après :

```
16:53:26.120293 IP (tos 0x0, ttl 64, id 55402, offset 0, flags [DF], proto TCP (6), length 64)
    106.186.29.14.57657 > 74.125.226.86.80: Flags [S], cksum 0x07de (incorrect -> 0xc67b), seq 3854071484, win 2
16:53:26.121734 IP (tos 0x0, ttl 57, id 16732, offset 0, flags [none], proto TCP (6), length 72)
    74.125.226.86.80 > 106.186.29.14.57657: Flags [S.], cksum 0xb913 (correct), seq 2213928284, ack 3854071485,
```

Ici, le client, n'ayant pas de gâteau pour `74.125.226.86`, a dû envoyer une option Fast Open vide (et donc pas de données : la longueur de son paquet `SYN` est nulle). À la deuxième connexion, on a un gâteau, on s'en sert :

```
16:54:30.200055 IP (tos 0x0, ttl 64, id 351, offset 0, flags [DF], proto TCP (6), length 161)
    106.186.29.14.57659 > 74.125.226.86.80: Flags [S], cksum 0x083f (incorrect -> 0x651d), seq 1662839861:16
16:54:30.201529 IP (tos 0x0, ttl 57, id 52873, offset 0, flags [none], proto TCP (6), length 60)
    74.125.226.86.80 > 106.186.29.14.57659: Flags [S.], cksum 0x67e3 (correct), seq 2010131453, ack 1662839
```

La section 4 de notre RFC plonge ensuite dans les détails compliqués de TCP Fast Open. Le gâteau est un MAC généré par le serveur et est donc opaque au client. Ce dernier ne fait que le stocker et le renvoyer. L'option Fast Open est simple : juste le code 34, une longueur (qui peut être nulle, par exemple pour un client qui ne s'est pas encore connecté à ce serveur, et qui demande donc un gâteau), et éventuellement le gâteau. Le serveur, lors de la génération du gâteau, va typiquement devoir suivre ces règles :

— Lier le gâteau à l'adresse IP source (pour éviter qu'un attaquant ayant espionné le réseau n'utilise le gâteau d'un autre),
— Utiliser un algorithme de génération imprévisible de l'extérieur (par exemple un générateur aléatoire),
— Aller vite (le but de Fast Open est de diminuer la latence : pas question de faire des heures de calcul cryptographiques compliqués),
— Imposer une date d'expiration au gâteau (soit en changer la clé privée utilisée lors de la génération, soit en incluant une estampille temporelle dans les données qui servent à générer le gâteau).

Un exemple d'algorithme valable (mais rappelez-vous que le gâteau est opaque, le serveur peut donc utiliser l'algorithme qu'il veut) est de chiffrer avec AES l'adresse IP et de changer la clé AES de temps en temps (invalidant ainsi automatiquement les vieux gâteaux). AES étant très rapide, cet algorithme a toutes les propriétés listées plus haut. Pour vérifier un gâteau entrant, le serveur a tout simplement à refaire tourner l'algorithme et voir s'il obtient le même résultat.

Et côté client ? Comme indiqué plus haut, le client doit stocker les gâteaux reçus (sans les comprendre : ils sont opaques pour lui) et les renvoyer lors des connexions suivantes vers le même serveur. Puisqu'on mémorise le gâteau de chaque serveur, on peut en profiter pour mémoriser également le MSS, ce qui indiquera la taille des données qu'on pourra envoyer dans le prochain paquet `SYN`. (Rappelez-vous que le serveur indique normalement sa MSS dans le `SYN + ACK`, donc trop tard pour Fast Open.) Mais attention : même si le MSS ainsi mémorisé est grand (supérieur à la MTU, par exemple), ce n'est pas forcément une bonne idée d'envoyer autant de données dans le paquet `SYN`. Des problèmes comme la fragmentation ou comme les "*middleboxes*" ne s'attendant pas à des `SYN` s'étalant sur plusieurs paquets IP, risquent de diminuer les performances, voire d'empêcher TCP de marcher. Ah, et si on ne connait pas le MSS, on doit se limiter à 536 octets en IPv4 et 1240 en IPv6.

Comme toujours sur l'Internet, lorsqu'on déploie une nouvelle technique, il faut aussi tenir compte des trucs bogués. Si le serveur ne répond pas aux `SYN` comportant l'option Fast Open, cela peut être parce qu'une stupide "*middlebox*" a décidé de jeter ces paquets, qui passeraient sans l'option. Même chose au cas où le serveur n'accuse pas réception des données qui étaient dans le `SYN` : le client Fast Open doit être prêt à réessayer sans cette option, et à mémoriser que le serveur ne doit pas être utilisé avec Fast Open. (Notez, car le RFC ne le fait pas, que ces incompatibilités, étant typiquement causées par une "*middlebox*" et pas par le serveur lui-même, peuvent changer dans le temps, si le routage fait soudain passer par un autre chemin.)

Autre point important lorsqu'on met en œuvre Fast Open : le serveur doit garder une trace en mémoire du nombre de connexions qui ont demandé Fast Open mais n'ont pas encore terminé la triple poignée de mains. Et, au delà d'une certaine limite, le serveur doit refuser de nouvelles connexions Fast Open (en ne renvoyant pas d'option Fast Open dans le `SYN + ACK`), n'acceptant que le TCP traditionnel. Cette précaution permet de résister à certaines attaques par déni de service.

En parlant d'attaques, la section 5 du RFC se concentre sur la sécurité. L'obligation d'envoyer un gâteau authentique arrête certaines attaques triviales (envoyer paquets `SYN` avec des données qui vont

faire travailler le serveur). Mais d'autres attaques restent possibles. Accrochez-vous, nous allons étudier ce qu'un méchant peut faire contre des serveurs TCP Fast Open.

D'abord, il peut tenter d'épuiser les ressources du serveur en utilisant des gâteaux valides. Où les obtient-il ? Cela peut être en utilisant plein de machines (un botnet). Bien sûr, vous allez me dire, on peut faire des tas d'attaques par déni de service avec un botnet mais, avec Fast Open, les zombies peuvent faire plus de dégâts pour moins cher (ils ne sont pas obligés d'écouter les réponses ni même de les attendre). D'où l'importance de la variable « nombre de connexions Fast Open pas complètement ouvertes » citée plus haut.

On ne peut pas normalement voler des gâteaux à une machine et les utiliser ensuite soi-même puisque le gâteau est (si le serveur a bien fait son boulot) lié à l'adresse IP. Mais ce vol reste possible si plusieurs machines partagent une adresse IP publique (cas du CGN par exemple). Une solution possible <http://www.ietf.org/mail-archive/web/tcpm/current/msg07192.html> serait d'inclure dans le calcul du gâteau, non seulement l'adresse IP mais aussi la valeur d'une option TCP "*Timestamp*".

Fast Open peut aussi en théorie être utilisé dans des attaques par réflexion. Par exemple (mais le RFC indique aussi d'autres méthodes), si l'attaquant contrôle une machine dans le réseau de sa victime, il peut obtenir des gâteaux valables et ensuite, lancer depuis un botnet des tas de connexions Fast Open en usurpant l'adresse IP source de sa victime. Les serveurs Fast Open vont alors renvoyer des données (potentiellement plus grosses que les requêtes, donc fournissant une amplification, chose bien utile pour une attaque par déni de service) à la victime. C'est idiot de la part de l'attaquant de s'en prendre à une machine qu'il contrôle déjà ? Non, car sa vraie victime peut être le réseau qui héberge la machine compromise. Les réponses des serveurs Fast Open arriveront peut-être à saturer la liaison utilisée par ce réseau, et cela en contrôlant juste une machine (soit par piratage, soit par location normale d'une machine chez l'hébergeur qu'on veut attaquer). La seule protection envisagée pour l'instant est de décourager les serveurs d'envoyer les réponses au-delà d'une certaine taille, tant que la triple poignée de mains n'a pas été terminée. Mais cela diminue une partie de l'intérêt de TCP Fast Open.

Bon, fini avec ces tristes questions de sécurité, revenons à la question de l'applicabilité de Fast Open. On a bien dit que Fast Open ne convient pas à tous les cas. Si je suis développeur, dans quels cas mon application a-t-elle raison d'utiliser Fast Open ? D'abord, on a vu que Fast Open fait courir le risque d'une duplication du `SYN` si le paquet est dupliqué **et** que le second arrive après que le serveur ait détruit le début de connexion. En pratique, la probabilité d'une telle malchance semble faible. Le RFC ne fournit pas de chiffres précis (voir Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., Towsley, D., « "*Measurement and classification of out-of-sequence packets in a tier-1 IP backbone*" <ftp://gaia.cs.umass.edu/pub/Jaiswal03_oos.pdf> » dans "*IEEE/ACM Transactions on Networking (TON)*"). Dans le doute, une application qui ne pourrait pas gérer le cas de données dupliquées ne doit donc **pas** activer Fast Open (rappelez-vous qu'il ne doit pas non plus l'être par défaut). Comme indiqué plus haut, pour HTTP, un `GET` ne pose pas de problèmes (les navigateurs Web impatients causent déjà souvent des `GET` dupliqués, qu'on retrouve dans ses journaux) mais un `POST` non protégé (par exemple par les requêtes conditionnelles du RFC 7232) a davantage de chances de créer des histoires.

Autre cas où il n'y a pas de problèmes à attendre, celui de TLS. Si le client met le `TLS_CLIENT_HELLO` dès le `SYN`, cela n'entraine pas de conséquences fâcheuses si le `SYN` est dupliqué, et cela fait gagner un RTT sur la poignée de mains de TLS.

Ensuite, même s'il n'a pas de conséquences néfastes, TCP Fast Open n'a pas non plus d'avantages si le temps d'établissement de la connexion est négligeable devant la durée totale de la connexion. Une requête HTTP pour un fichier de petite taille peut sans doute profiter de Fast Open, mais pas le transfert d'une énorme vidéo.

On a parlé plus haut des connexions HTTP persistantes (RFC 7230). TCP Fast Open est-il utile lorsqu'on a ces connexions persistantes ? Oui, répond notre RFC. L'étude de Radhakrishnan, S., Cheng, Y., Chu, J., Jain, A. et Raghavan, B. citée plus haut, ainsi que celle d'Al-Fares, M., Elmeleegy, K., Reed, B. et Gashinsky, I., « "*Overclocking the Yahoo ! CDN for Faster Web Page Loads*" <http://conferences.sigcomm.org/imc/2011/docs/p569.pdf> » (dans "*Proceedings of Internet Measurement Conference* ", novembre 2011), montrent que le nombre moyen de transactions HTTP par connexion TCP n'est que de 2 à 4, alors même que ces connexions restent ouvertes plusieurs minutes, ce qui dépasse normalement le temps de réflexion d'un être humain. Les mesures effectuées sur Chrome (qui garde les connexions de 5 à 10 minutes) ne voyaient que 3,3 requêtes HTTP par connexion. Faudrait-il allonger cette durée pendant laquelle les connexions persistent ? Cela entrainerait d'autres problèmes, par exemple avec les routeurs NAT qui, en violation du RFC 5382, coupent automatiquement les connexions bien avant la limite de deux heures demandée par le RFC (voir les études de Haetoenen, S., Nyrhinen, A., Eggert, L., Strowes, S., Sarolahti, P. et Kojo., M., « "*An Experimental Study of Home Gateway Characteristics*" <https://eggert.org/papers/2010-imc-hgw-study.pdf> » dans les "*Proceedings of Internet Measurement Conference*", octobre 2010 ainsi que de Wang, Z., Qian, Z., Xu, Q., Mao, Z. et Zhang, M., « "*An Untold Story of Middleboxes in Cellular Networks*" » dans "*Proceedings of SIGCOMM*", août 2011). Envoyer des "*keepalives*" TCP résoudrait ce problème mais serait une sérieuse source de consommation électrique pour les machines fonctionnant sur batteries. On voit même le phénomène inverse, les navigateurs Web conçus pour les équipements mobiles qui se mettent à couper les connexions HTTP persistantes plus tôt (Souders, S., « "*Making A Mobile Connection*" <http://www.stevesouders.com/blog/2011/09/21/making-a-mobile-connection/> »).

Ce RFC sur TCP Fast Open a le statut « expérimental ». Qu'est-ce qu'on doit encore étudier et mesurer pour être sûr que Fast Open marche bien ? D'abord, quel est le pourcentage exact de chemins sur l'Internet où les paquets TCP ayant des options inconnues sont jetés ? Pas mal de "*middleboxes*" stoppent stupidement tout ce qu'elles ne comprennent pas (Medina, A., Allman, M., and S. Floyd, « "*Measuring Interactions Between Transport Protocols and Middleboxes*" <http://conferences.sigcomm.org/imc/2004/papers/p336-medina.pdf> » dans "*Proceedings of Internet Measurement Conference*" en octobre 2004). Une option nouvelle, comme Fast Open, pourrait donc avoir du mal à percer. Des mesures semblent indiquer que 6 % des chemins Internet seraient dans ce cas (Langley, A, « "*Probing the viability of TCP extensions*" <https://www.imperialviolet.org/binary/ecntest.pdf> » ou bien Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M. et Tokuda, H., « "*Is it Still Possible to Extend TCP ?*" <http://conferences.sigcomm.org/imc/2011/docs/p181.pdf> » dans "*Proceedings of Internet Measurement Conference*" en novembre 2011). TCP Fast Open réagit à ce problème en réessayant sans l'option (comme le font les résolveurs DNS quand ils n'obtiennent pas de réponse lorsque les requêtes sont envoyées avec EDNS).

Autre sujet de recherche intéressant, les liens avec la congestion. Normalement, Fast Open ne modifie pas les algorithmes qui tentent d'éviter la congestion mais il peut y avoir des cas subtils où Fast Open ne peut pas suivre ces algorithmes (données déjà envoyées avant que les pertes ne soient détectées).

TCP Fast Open impose actuellement l'usage d'un gâteau pour détecter les méchants qui tricheraient sur leur adresse IP source. Mais pour certains serveurs qui n'assurent que des tâches simples et idempotentes (le RFC prend l'exemple d'un serveur HTTP qui ne ferait que des redirections), la protection fournie par le gâteau est peut-être inutile et on pourrait faire des économies en s'en passant (l'expérience du DNS, qui est aussi requête/réponse, me rend personnellement sceptique sur ce point). Ou bien le serveur pourrait s'en passer par défaut, et basculer en Fast Open avec gâteau s'il détecte une attaque par déni de service en cours ? Bref, il y a encore des sujets ouverts.

La section 8 rappelle les travaux qui avaient précédé Fast Open. Il y a bien sûr T/TCP, déjà cité, qui avait trébuché sur les problèmes de sécurité <http://www.phrack.com/issues.html?issue=53&id=6>. Une autre solution pour TCP est le TCPCT du RFC 6013 (désormais abandonné, cf. RFC

7805). Mais il y a aussi les solutions dans les applications comme « *"preconnect"* <http://www.belshe.com/2011/02/10/the-era-of-browser-preconnect/> ».

Sinon, si vous voulez de la lecture sur Fast Open, il y a une bonne explication dans Linux Weekly News <https://lwn.net/Articles/508865/>, avec des détails sur son utilisation dans les programmes.

À propos de programmes, et les mises en œuvre ? TCP Fast Open existe dans le navigateur Chrome, ainsi que dans le noyau Linux, depuis la version 3.7 (3.16 seulement <http://lwn.net/Articles/598280/> pour IPv6). Une Debian en version « jessie » permet donc de tester. Une API possible figure en annexe A du RFC. Du côté serveur, il faut, après avoir créé une *"socket"* : *" setsockopt(sfd, SOL_TCP, TCP_FASTOPEN, &qlen, sizeof(qlen));"* C'est plus compliqué sur le client (il faudrait envoyer des données dans le `connect()` ou bien utiliser `sendto()` ou encore une option de la *"socket"*). Le programme (en ligne sur `https://www.bortzmeyer.org/files/client-http-tcp-fastopen.c`) montre un simple client HTTP utilisant Fast Open. C'est avec lui qu'ont été obtenues les traces montrées plus haut (le pcap complet est sur pcapr <http://www.pcapr.net/view/bortzmeyer+pcapr/2014/10/0/8/tcp-fastopen-nocookie-gmail.pcap.html> ; il utilise une valeur expérimentale pour l'option et pas la valeur standard de 34).

Une lecture pour finir : la présentation « *"Network Support for TCP Fast Open (NANOG 67)"* <https://www.nanog.org/sites/default/files/Paasch_Network_Support.pdf> ».

Merci à Alexis La Goutte pour ses remarques.