RFC 7766 : DNS Transport over TCP - Implementation Requirements

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 4 mars 2016

Date de publication du RFC : Mars 2016

https://www.bortzmeyer.org/7766.html

Ce nouveau RFC décrit des exigences bien plus importantes pour l'utilisation du DNS au-dessus de TCP : non seulement les serveurs DNS doivent gérer TCP (ce qui a toujours été le cas), mais ils doivent le faire sur un pied d'égalité avec UDP. TCP n'est pas une solution de secours pour des cas rares, c'est désormais une alternative complète à UDP pour le service DNS. La règle est donc désormais plus radicale qu'avec le RFC précédent, le RFC 5966 [1], maintenant retiré.

Traditionnellement, surtout avant la sortie du RFC 5966, le DNS utilisait surtout UDP. Beaucoup de gens croyaient même, bien à tort <http://www.circleid.com/posts/afnic_dns_server_redelegation/>, que TCP ne servait que pour des cas très particuliers comme le transfert de zones <https://www.bortzmeyer.org/recuperer-zone-dns.html> ou comme la récupération de données plus grosses que 512 octets. Ces croyances étaient déjà fausses lors de la sortie des RFC originels sur le DNS, mais le RFC 5966 leur a tordu le cou complètement. UDP a en effet des limites :

— Les données deviennent plus grosses (clés cryptographiques, par exemple, avec DNSSEC ou DANE), rendant indispensables TCP **ou** EDNS (RFC 6891),
— UDP, contrairement à TCP <https://www.bortzmeyer.org/usurpation-adresse-ip.html>, n'offre aucune protection contre l'usurpation d'adresse IP, usurpation qui est utilisée dans les attaques par réflexion <https://www.bortzmeyer.org/attaques-reflexion.html>,
— TCP est utilisé par certaines solutions d'amélioration de la confidentialité des requêtes DNS, notamment « DNS sur TLS » (cf. le travail en cours au sein du groupe de travail DPRIVE <https://tools.ietf.org/wg/dprive>).

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc5966.txt

La formulation des sections 3.7 du RFC 1034 et 4.2 du RFC 1035 sur l'usage de TCP étaient assez claires (mais personne ne lit les RFC...) Par contre, la section 6.1.3.2 du RFC 1123 avait semé la confusion en laissant entendre que TCP était facultatif. C'est désormais corrigé, TCP est bien obligatoire pour le DNS.

Mais TCP, utilisé bêtement, souffre de plusieurs défauts : si on ouvre une connexion TCP à chaque requête DNS, la latence <https://www.bortzmeyer.org/latence.html> va être bien plus élevée qu'en UDP, et, si on fait passer toutes les requêtes séquentiellement sur une seule connexion TCP, et que le serveur les traite dans l'ordre, une requête lente peut faire attendre des requêtes qui auraient été rapides. Voici pourquoi notre RFC impose un certain nombre de comportements intelligents (section 3) :

— Connexion persistente et réutilisable (ne pas fermer la connexion TCP, mais la réutiliser pour les requêtes suivantes),
— "*Pipelining*" (pour un client DNS, ne pas attendre la réponse avant d'envoyer d'autres questions, mais en envoyer plusieurs dans le tuyau),
— Traitement non-séquentiel (pour un serveur, le fait de traiter en parallèle les requêtes, pour éviter qu'une requête lente ne retarde celles qui suivent). Cela concerne surtout les résolveurs (car le temps de répondre à une requête peut varier considérablement, en fonction du cache et du temps de réponse des serveurs faisant autorité.)

J'ai parlé plus haut de la taille des réponses DNS (section 4 du RFC). C'était autrefois une des bonnes raisons de passer en TCP, afin d'obtenir des réponses supérieures à 512 octets. Les clients DNS le font typiquement automatiquement. Par exemple, ici, dig tente d'obtenir une réponse de plus de 512 octets, et elle est donc tronquée ("*flag*" TC pour "*TrunCation*") :

```
% dig +bufsize=0 +noedns +nodnssec +ignore NS fox.com
...
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38102
;; flags: qr tc rd ra ad; QUERY: 1, ANSWER: 15, AUTHORITY: 0, ADDITIONAL: 0
...
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; MSG SIZE  rcvd: 494
```

J'ai utilisé l'option `+ignore` pour voir cette troncation. Si on ne l'utilise pas, dig passe automatiquement en TCP (le message « "*Truncated, retrying in TCP mode*" ») :

```
% dig +bufsize=0 +noedns +nodnssec NS fox.com
;; Truncated, retrying in TCP mode.
...
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 24693
;; flags: qr rd ra; QUERY: 1, ANSWER: 16, AUTHORITY: 0, ADDITIONAL: 0
...
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; MSG SIZE  rcvd: 522
```

Maintenant qu'on a EDNS (RFC 6891), TCP est moins utile pour ce problème de taille (il reste utile pour les autres raisons énumérées au début). Voici un exemple avec EDNS pour une donnée de grande taille (comme le note la section 4, une réponse négative, `NXDOMAIN` dans une zone signée avec DNSSEC/RSA et utilisant le NSEC3 du RFC 5155 va presque toujours faire plus de 512 octets) :

```
% dig +bufsize=4096 A nimportequoinexistepas.fr
...
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 27274
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 8, ADDITIONAL: 1
...
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; MSG SIZE  rcvd: 1012
```

Le `+bufsize=4096` active EDNS, avec une taille maximale des réponses de 4 096 octets.

(Notez que la troncation est aussi utilisée pour des raisons de sécurité, lorsqu'un serveur détecte une attaque par amplification <https://www.bortzmeyer.org/attaques-reflexion.html>. Avec la technique RRL <http://ss.vix.su/~vixie/isc-tn-2012-1.txt>, le serveur répond alors avec des paquets tronqués - paquets nommés SLIP, pour forcer le client à réessayer en TCP, où il n'y aura pas d'usurpation d'adresse IP possible.)

Reste la question de la MTU. La réponse précédente faisait 1 012 octets, inférieure à la MTU typique (en général de 1 500 octets, à cause d'Ethernet). Si elle est plus grande, la réponse va être fragmentée et, là, les ennuis commencent. On ne peut pas compter sur la bonne réception des paquets fragmentés : trop de pare-feux configurés avec les pieds sont sur le trajet, et ils jettent souvent bêtement les fragments. En parlant de "*middleboxes*", il faut aussi noter que certaines perturbent EDNS (RFC 5625). Les réponses supérieures à 1 500 octets, elles, auront peut-être besoin de TCP :

```
% dig DNSKEY cpsc.gov
...
;; MSG SIZE  rcvd: 1733
```

La section 5 du RFC est la partie normative. Elle contient la nouvelle règle : un serveur DNS **doit** parler TCP. Et un client DNS peut s'en servir, il n'est même pas forcé d'attendre la troncation.

Maintenant, les détails pratiques. D'abord (section 6), la gestion des connexions TCP. Compte-tenu de la « triple poignée de mains » qui ouvre les connexions TCP, et de son coût en latence, il est fortement recommandé d'éviter de faire une connexion TCP par requête DNS. L'idée est donc, pour le serveur, de ne pas fermer la connexion dès qu'elle est inactive mais de la garder ouverte (comme le font HTTP 1.1 - RFC 7230 et HTTP 2 - RFC 7540) et, pour les clients, de ne pas fermer après la première requête (ce qu'ils font actuellement, en général). Mais, contrairement à HTTP, le DNS n'a pas encore de mécanisme de fermeture explicite de ces connexions TCP persistentes (certains ont été proposés). Il faut donc utiliser des heuristiques pour savoir quand fermer une connexion.

Au fait, j'ai parlé de « connexion inactive » sans l'expliquer. La section 3 la définit comme « pour un client, lorsqu'il a reçu toutes les réponses à ses questions, pour un serveur, lorsqu'il a répondu à toutes les questions posées ».

Les serveurs actuels laissent en général les connexions assez longtemps (deux minutes par défaut, sur Unbound ou sur NSD), et ont un nombre de connexions TCP maximal assez faible. Cela permet des attaques par déni de service triviales (ouvrir N connexions TCP, attendre) ou même des simples accidents, où un petit nombre de clients, sans le faire exprès, épuisent complètement les maigres ressources allouées à TCP. Avec BIND, vous pouvez par exemple regarder combien de clients TCP sont actuellement actifs par rapport au maximum autorisé. Si le nombre de clients actifs se rapproche trop du maximum, il y a un risque :

```
% rndc status | grep tcp
tcp clients: 8/100
```

Notre RFC recommande donc (section 6.2) une série de bonnes pratiques qui vont permettre à TCP d'être utilisé massivement pour le DNS. D'abord, réutiliser les connexions, de façon à amortir le coût, relativement élevé, de l'établissement d'une connexion, sur plusieurs requêtes. Attention, le client DNS ne doit pas réutiliser l'identicateur de requête ("*Query ID*", ou `id:` dans la sortie de dig) sur la même connexion, tant qu'il reste une question non répondue utilisant cet identificateur (autrement, le client ne saurait pas à quelle question correspond une réponse).

Ensuite, une requête ne devrait pas avoir à attendre la réponse à la requête précédente. Dès que le client a une question, il l'envoie ("*pipelining*").

Au passage, un client DNS gourmand pourrait être tenté d'ouvrir plusieurs connexions TCP avec le serveur. Notre RFC le déconseille (sauf pour des cas ponctuels comme les longs transferts de zone) : cela use des ressources, et c'est inutile, si on a le "*pipelining*" et le traitement en parallèle. Un serveur a donc le droit de mettre des limites et, par exemple, de ne pas accepter plus de N connexions TCP par adresse IP (avec N probablement strictement supérieur à 1 car il faut tenir compter des clients différents situés derrière une même adresse IP, par exemple en cas de CGN).

Bon, et le quantitatif ? C'est bien de dire qu'un serveur ne devrait pas laisser les connexions TCP inactives ouvertes trop longtemps, mais combien, exactement ? Le RFC recommande quelques secondes, pas plus, après le traitement de la dernière requête, sauf si on a reçu une demande explicite (voir le RFC 7828). À comparer avec les 120 secondes actuelles de NSD ou Unbound... Cette durée peut évidemment être réglée dynamiquement, par exemple pour répondre à la disponibilité de ressources chez le serveur (nombre maximum de connexions TCP atteint, je coupe des connexions inactives...)

Dans tous les cas, outre la coupure d'une connexion par un client qui a fini son travail, le client doit aussi se rappeler qu'un serveur DNS a le droit de couper les connexions TCP quand ça lui chante, et sans prévenir à l'avance. Le client doit donc gérer ces coupures (détecter et reconnecter). C'est d'autant plus important qu'un serveur peut redémarrer (perdant ses sessions TCP) ou que tout simplement, le routage peut changer, amenant le client DNS vers une autre instance d'un nuage "*anycast*", instance qui n'a pas les sessions TCP en cours.

Et le traitement en parallèle (section 7) ? C'est un des points importants de DNS sur TCP, sans lequel TCP n'aurait guère d'intérêt. Lorsqu'un résolveur minimum (rappel, la terminologie du DNS est dans le RFC 8499) se connecte à un résolveur complet en TCP, le résolveur complet va mettre un temps très variable à répondre aux questions. De quelques dizaines de micro-secondes si la réponse est dans le cache à des centaines de milli-secondes s'il a fallu interroger trois ou quatre serveurs faisant autorité avant d'avoir sa réponse. Si le traitement des requêtes était fait de manière strictement séquentielle, la requête rapide qui aurait la malchance d'être située après une requête lente devrait patienter très longtemps. Il est donc crucial de traiter les requêtes en parallèle.

Conséquence : il est parfaitement normal que, lorsqu'on envoie les questions Q1 puis Q2, la réponse R2 arrive avant R1. Tous les clients DNS doivent être prêts à cela. Le démultiplexage des réponses doit se faire sur le "*Query ID*" du DNS (et voilà pourquoi il ne faut pas le dupliquer) et, dans le cas le plus fréquent, également sur le nom demandé (QNAME, pour "*Query NAME*").

L'encodage d'un message DNS en TCP est très légèrement différent de ce qu'il est en UDP. Un champ Longueur, de deux octets, précède en effet le message, pour faciliter la lecture. Cela permet au récepteur de savoir combien d'octets il va lire dans le flux TCP (qui, rappelons-le, n'a pas la notion de messages, c'est juste un flux d'octets sans séparateurs). Par exemple, en Go, on lirait ainsi :

```
// Lire le champ Longueur
smallbuf := make([]byte, 2)
n, error := connection.Read(smallbuf)
msglength := binary.BigEndian.Uint16(smallbuf) // RFC 1035, section 4.2.2 "TCP usage"

// Allouer un tampon pour le message
message := make([]byte, msglength)

// Lire le message DNS
n, error = connection.Read(message)
```

Le RFC recommande (section 8) que les émetteurs DNS passent ce champ Longueur et le message ensemble, dans un seul `write()` sur le réseau, de façon à maximiser les chances qu'ils se trouvent dans le même paquet IP, certaines mises en œuvre réagissant mal dans le cas contraire. Ce n'est pas une garantie absolue (TCP peut découper en paquets IP comme il veut). Les experts en Go noteront que le code ci-dessus ne tient pas compte de ce risque : `Read()` peut lire moins d'octets que la taille du tampon, il faudrait tester ce cas (ou bien utiliser `ReadFull()`.) Et, naturellement, il faudrait tester les cas d'erreur.

On a vu qu'un des problèmes potentiels dus à l'utilisation de TCP était la latence liée à l'ouverture des connexions. Outre la réutilisation des connexions pour plusieurs requêtes, un autre mécanisme peut aider à limiter ce problème (section 9) : "*TCP Fast Open*" (RFC 7413). En permettant aux données d'être transportées dès le paquet initial (le paquet `SYN`), il fait gagner du temps. Contrairement à d'autres tentatives précédentes de faire de l'« ouverture rapide », "*TCP Fast Open*" ne permet pas d'attaques par réflexion, grâce à un "*cookie*". C'est donc une solution à considérer, pour les clients et serveurs DNS utilisant TCP.

C'est un bon prétexte pour discuter de sécurité (section 10). Lors de la mise au point de ce RFC, il y a eu beaucoup de discussions concernant le risque d'attaque par déni de service pour les serveurs DNS. Le traitement d'une requête TCP (y compris l'établissement de la connexion) nécessite, après tout, davantage de ressources que celui d'une requête UDP. Mais on a aussi davantage d'expérience avec les attaques utilisant TCP (c'est le protocole sous HTTP, après tout, et de nombreuses attaques sont faites contre les serveurs Web). Outre la lecture du « "*Security Assessment of the Transmission Control Protocol (TCP)*" <http://www.gont.com.ar/papers/tn-03-09-security-assessment-TCP.pdf> », notre RFC recommande de permettre de configurer le serveur DNS sur des points comme le nombre total de connexions TCP (`tcp-clients` pour BIND, `tcp-count` pour NSD, `incoming-num-tcp` pour Unbound), le nombre maximal de connexions TCP par adresse IP de client, la durée de vie des connexions inactives (`tcp-timeout` pour NSD et, attention, il faut redémarrer le serveur si on le change), le nombre maximal de requêtes DNS par connexion (`tcp-query-count` pour NSD), etc. Sans compter les réglages permis par le système d'exploitation (comme les innombrables paramètres sysctl `net.ipv4.tcp_*` de Linux et, oui, c'est `ipv4` même pour IPv6).

En prime, le RFC rappelle que les résolveurs ne devraient pas accepter des connexions du monde entier (RFC 5358), mais, sauf cas rares, uniquement de certains préfixes IP, leurs clients. Cela contribue à limiter les risques.

Pour les amateurs de choix technologiques complexes, l'annexe A du RFC résume les avantages et les inconvénients de TCP pour le DNS. Avantages :

— Élimine l'usurpation d'adresse et donc les attaques par réflexion (cf. mon article JRES « Persée et la Gorgone » <https://2013.jres.org/archives/37/index.htm>),
— Moins d'ennuis avec la fragmentation, grâce à la PMTUD.

Inconvénients :

— Coût par requête plus élevé à cause de l'établissement de la connexion. On peut réduire ce coût en réutilisant les connexions,

— Davantage d'état à maintenir chez les clients et les serveurs,
— Risques de plantage des connexions TCP si on se connecte à un serveur "*anycast*" et que le routage change,
— Risque avec des "*middleboxes*" boguées (RFC 5625).

Le rapport « "*Connection-Oriented DNS to Improve Privacy and Security*" <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=7163025> » discute bien plus en détail ces choix.

Enfin, l'annexe B liste les principaux changements depuis le RFC 5966. Ils sont importants puisqu'on passe d'un modèle « TCP dans certains cas » à un modèle « pouvoir faire tout en TCP si on veut ». Par exemple, un client peut désormais utiliser TCP avant d'avoir testé en UDP. Les autres changements sont des conséquences de ce principe : les logiciels DNS doivent désormais permettre la réutilisation des connexions, le "*pipelining*" et le traitement en parallèle des requêtes parce que, sans cela, TCP ne serait pas vraiment utilisable pour le DNS.

Allez, pour finir, un example avec dig où on demande explicitement du TCP :

```
% dig +tcp @d.nic.fr NS gouvernement.fr

; <<>> DiG 9.9.5-9+deb8u3-Debian <<>> +tcp @d.nic.fr NS gouvernement.fr
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 14247
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 7, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;gouvernement.fr. IN NS

;; AUTHORITY SECTION:
gouvernement.fr. 172800 IN NS ns2.produhost.net.
gouvernement.fr. 172800 IN NS ns33.produhost.net.
gouvernement.fr. 172800 IN NS ns1-sgg.produhost.net.
98dsof44tab20a3p7gv0cgelh01vlv9j.fr. 5400 IN NSEC3 1 1 1 D0D3AC0C (
98ECUBNBQN7GCK4GQ1KQURRJTRUF6P83
NS SOA TXT NAPTR RRSIG DNSKEY NSEC3PARAM )
98dsof44tab20a3p7gv0cgelh01vlv9j.fr. 5400 IN RRSIG NSEC3 8 2 5400 (
20160213101014 20151215091014 1146 fr.
MDAHRaeLGVQiK6KEQvFn8PNSl608bd5iMup4cwkqgStB
uXEFWOpQLeSy37IattIDOt0AhwEirpNe7pzH+KMOdrM0
mEWGiGa3iSdsF9hc29JDUtVyfalNmJqPIoDplw+Q0lpI
Jxd8XlI/229PEH64Nua2jgLk5gxyZPCFt7yVvGg= )
ceast34o75ellj5eprb3j5i82re4g1ug.fr. 5400 IN NSEC3 1 1 1 D0D3AC0C (
CEB3B4VDS9MG7Q92A7FVCRQ5TTIVJ3LT
NS DS RRSIG )
ceast34o75ellj5eprb3j5i82re4g1ug.fr. 5400 IN RRSIG NSEC3 8 2 5400 (
20160213054416 20151215054416 1146 fr.
Eusw6LfnNPxmquf8JDHJnWZpBFJj8hrvJillx3bounyO
YW8ceJsrm0AkjMHm5Sh40TDfXcVrLeTK5U9PWJYqpUAG
7LABfh4nwwnlofvuyKtSx1yl44G6Gv/MSyQUnj0dIgQ5
6MB+b3tsUx2rHI2D0w9kiV50xJeCYVBiViQP2y8= )

;; Query time: 19 msec
;; SERVER: 2001:678:c::1#53(2001:678:c::1)
;; WHEN: Tue Dec 15 22:07:36 CET 2015
;; MSG SIZE  rcvd: 607
```

Et le tcpdump correspondant :

```
% sudo tcpdump -n -vvv host 2001:678:c::1
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
22:08:08.073682 IP6 (hlim 64, next-header TCP (6) payload length: 40) 2001:db8:cafe:1234:21e:8cff:fe76:29b6.6010
22:08:08.090127 IP6 (hlim 57, next-header TCP (6) payload length: 40) 2001:678:c::1.53 > 2001:db8:cafe:1234:21e:
22:08:08.090175 IP6 (hlim 64, next-header TCP (6) payload length: 32) 2001:db8:cafe:1234:21e:8cff:fe76:29b6.6010
22:08:08.090246 IP6 (hlim 64, next-header TCP (6) payload length: 78) 2001:db8:cafe:1234:21e:8cff:fe76:29b6.6010
22:08:08.107559 IP6 (hlim 57, next-header TCP (6) payload length: 32) 2001:678:c::1.53 > 2001:db8:cafe:1234:21e:
22:08:08.109126 IP6 (hlim 57, next-header TCP (6) payload length: 641) 2001:678:c::1.53 > 2001:db8:cafe:1234:21e
22:08:08.109137 IP6 (hlim 64, next-header TCP (6) payload length: 32) 2001:db8:cafe:1234:21e:8cff:fe76:29b6.6010
22:08:08.109743 IP6 (hlim 64, next-header TCP (6) payload length: 32) 2001:db8:cafe:1234:21e:8cff:fe76:29b6.6010
22:08:08.130087 IP6 (hlim 57, next-header TCP (6) payload length: 32) 2001:678:c::1.53 > 2001:db8:cafe:1234:21e:
22:08:08.130133 IP6 (hlim 64, next-header TCP (6) payload length: 32) 2001:db8:cafe:1234:21e:8cff:fe76:29b6.6010
```

Question mise en œuvre, remarquez que les dernières versions de BIND ont un outil nommé mdig qui permet d'envoyer plusieurs requêtes DNS sur une même connexion TCP (en "*pipeline*", sans attendre la réponse à la première).