RFC 9170 : Long-term Viability of Protocol Extension Mechanisms

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 1 janvier 2022

Date de publication du RFC : Décembre 2021

https://www.bortzmeyer.org/9170.html

Après des années de déploiement d'un protocole sur l'Internet, lorsqu'on essaie d'utiliser des fonctions du protocole parfaitement standards et légales, mais qui avaient été inutilisées jusqu'à présent, on découvre souvent que cela ne passe pas : des programmes bogués, notamment dans les "*middleboxes*", plantent de manière malpropre lorsqu'ils rencontrent ces (toutes relatives) nouveautés. C'est ce qu'on nomme l'**ossification** de l'Internet. Ce RFC de l'IAB fait le point sur le problème et sur les solutions proposées, par exemple le **graissage**, l'utilisation délibérée et précoce de variations, pour ne pas laisser d'options inutilisées.

Lorsqu'un protocole a du succès (cf. RFC 5218 [1], sur cette notion de succès), on va vouloir le modifier pour traiter des cas nouveaux. Cela n'est pas toujours facile, comme le note le RFC 8170. Tout protocole a des degrés de liberté ("*extension points*") où on pourra l'étendre. Par exemple, le DNS permet de définir de nouveaux types de données (contrairement à ce qu'on lit souvent, le DNS ne sert pas qu'à « trouver des adresses IP à partir de noms ») et IPv6 permet de définir de nouvelles options pour la destination du paquet, voire de nouveaux en-têtes d'extension. En théorie, cela permet d'étendre le protocole. Mais en pratique, utiliser ces degrés de liberté peut amener des résultats imprévus, par exemple, pour le cas du DNS, un pare-feu programmé avec les pieds qui bloque les paquets utilisant un type de données que le pare-feu ne connait pas. Ce RFC se focalise sur des couches relativement hautes, où tout fonctionne de bout en bout (et où, en théorie, les intermédiaires doivent laisser passer ce qu'ils ne comprennent pas). Les couches basses impliquent davantage de participants et sont donc plus problématiques. Ainsi, pour IPv6, l'en-tête « options pour la destination » doit normalement être ignoré et relayé aveuglément par les routeurs alors que l'en-tête « options pour chaque saut » doit (enfin, devait, avant le RFC 8200) être compris et analysé par tous les routeurs du chemin, ce qui complique sérieusement son utilisation.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc5218.txt

Notre RFC a été développé par l'IAB dans le cadre du programme « "*Evolvability, Deployability, & Maintainability (EDM)*" <https://www.iab.org/activities/programs/evolvability-deployability-m
> ».

Déployer une nouvelle version d'un protocole, ou simplement utiliser des options qui n'avaient pas été pratiquées avant, peut donc être très frustrant. Prenons le cas imaginaire mais très proche de cas réels d'un protocole qui a un champ de huit bits inutilisé et donc le RFC d'origine dit que l'émetteur du paquet doit mettre tous ces bits à zéro, et le récepteur du paquet, en application du principe de robustesse <https://www.bortzmeyer.org/principe-robustesse.html>, doit ignorer ces bits. Un jour, un nouveau RFC sort, avec une description du protocole où ces bits ont désormais une utilité. Les premiers logiciels qui vont mettre en œuvre cette nouvelle version, et mettre certains bits à un, vont fonctionner lors de tests puis, une fois le déploiement sur l'Internet fait, vont avoir des problèmes dans certains cas. Un pare-feu programmé par des incompétents va jeter ces paquets. Dans son code, il y a un test que le champ vaut zéro partout, car le programmeur n'a jamais lu le RFC et il a juste observé que ce champ était toujours nul. Cette observation a été mise dans le code et bonne chance pour la corriger. (Les DSI qui déploient ce pare-feu n'ont pas regardé s'il était programmé correctement, ils ont juste lu la brochure commerciale.) À partir de là, les programmeurs qui mettent en œuvre le protocole en question ont le choix de foncer et d'ignorer les mines, acceptant que leur programme ne marche pas si les paquets ont le malheur de passer par un des pare-feux bogués (et seront donc perdus), ou bien de revenir à l'ancienne version, ce second choix étant l'**ossification** : on n'ose plus rien changer de peur que ça casse quelque chose quelque part. Comme les utilisateurs ignorants attribueraient sans doute les problèmes de "*timeout*" à l'application, et pas au pare-feu mal écrit, le choix des programmeurs est vite fait : on ne déploie pas la nouvelle version. C'est d'autant plus vrai si le protocole est particulièrement critique pour le bon fonctionnement de l'Internet (BGP, DNS…) ou bien s'il y a de nombreux acteurs non coordonnés (cf. section 2.3). Bien sûr, le blocage de certains paquets peut être volontaire mais, bien souvent, il résulte de la négligence et de l'incompétence des auteurs de "*middleboxes*" (cf. section 5).

Les exemples de tels problèmes sont innombrables (l'annexe A du RFC en fournit plusieurs). Ainsi, TLS a eu bien des ennuis avec des nouvelles valeurs de l'extension `signature_algorithms`.

Si le protocole en question était conçu de nos jours, il est probable que ses concepteurs prendraient la précaution de réserver quelques valeurs non nulles pour le champ en question, et de demander aux programmes d'utiliser de temps en temps ces valeurs, pour être sûr qu'elles sont effectivement ignorées. C'est ce qu'on nomme le **graissage** et c'est une technique puissante (mais pas parfaite) pour éviter la rouille, l'ossification. (Notez que les valeurs utilisées pour le graissage ne doivent pas être consécutives, pour limiter les risques qu'un programmeur de "*middlebox*" paresseux ne les teste facilement. Et qu'il faut les réserver pour que, plus tard, l'utilisation de vraies valeurs pour ce champ ne soit pas empêchée par les valeurs de graissage.) Le graissage ne résout pas tous les problèmes puisqu'il y a toujours le risque que les valeurs utilisées pour graisser finissent par bénéficier d'un traitement de faveur, et soient acceptées, alors que les valeurs réelles poseront des problèmes.

On pourrait penser qu'une meilleure conception des protocoles éviterait l'ossification. (C'est le discours des inventeurs géniaux et méconnus qui prétendent que leur solution magique n'a aucun inconvénient et devrait remplacer tout l'Internet demain.) Après tout, le RFC 6709 contient beaucoup d'excellents conseils sur la meilleure façon de concevoir des protocoles pour qu'ils puissent évoluer. Par exemple, il insiste sur le fait que le mécanisme de négociation qui permet aux deux parties de se mettre d'accord sur une option ou une version doit être simple, pour qu'il y ait davantage de chances qu'il soit mis en œuvre correctement dans tous les programmes. En effet, en l'absence de graissage, ce mécanisme ne sera testé en vrai que lorsqu'on introduira une nouvelle version ou une nouvelle option et, alors, il sera trop tard pour modifier les programmes qui n'arrivent pas à négocier ces nouveaux cas, car leur code de négociation est bogué. Le RFC 6709 reconnait ce problème (tant qu'on n'a pas utilisé un mécanisme, on ne sait pas vraiment s'il marche) et reconnait que le conseil d'un mécanisme simple n'est pas suffisant.

Notre RFC 9170 contient d'ailleurs une petite pique contre QUIC <https://www.bortzmeyer.org/quic.html> en regrettant qu'on repousse parfois à plus tard le mécanisme de négociation de version, pour arriver à publier la norme décrivant le protocole (exactement ce qui est arrivé à QUIC, qui n'a pas de mécanisme de négociation des futures versions). Le problème est que, une fois la norme publiée et le protocole déployé, il sera trop tard…

Bref, l'analyse du RFC (section 3) est qu'il faut utiliser tôt et souvent les mécanismes d'extension. Une option ou un moyen de négocier de nouvelles options qui n'a jamais été utilisé depuis des années est probablement ossifié et ne peut plus être utilisé. C'est très joli de dire dans le premier RFC d'un protocole « cet octet est toujours à zéro dans cette version mais les récepteurs doivent ignorer son contenu car il pourra servir dans une future version » mais l'expérience prouve largement qu'une telle phrase est souvent ignorée et que bien des logiciels planteront, parfois de façon brutale, le jour où on commencer à utiliser des valeurs non nulles. Il faut donc utiliser les mécanismes ou bien ils rouillent. Par exemple, si vous concevez un mécanisme d'extension dans votre protocole, il est bon que, **dès le premier jour**, au moins une fonction soit mise en œuvre via ce mécanisme, pour forcer les programmeurs à en tenir compte, au lieu de simplement sauter cette section du RFC. Les protocoles qui ajoutent fréquemment des options ou des extensions ont moins de problèmes que ceux qui attendent des années avant d'exploiter leurs mécanismes d'extension. Et plus on attend, plus c'est pire. C'est pour cela que, par exemple, il faut féliciter le RIPE NCC d'avoir tenté l'annonce de l'attribut BGP 99 <https://www.bortzmeyer.org/bgp-attribut-99.html>, même si cela a cassé des choses, car si on ne l'avait pas fait (comme l'avaient demandé certaines personnes qui ne comprenaient pas les enjeux), le déploiement de nouveaux attributs dans BGP serait resté quasi-impossible.

Tester tôt est d'autant plus important qu'il peut être crucial, pour la sécurité, qu'on puisse étendre un protocole (par exemple pour l'agilité cryptographique, cf. RFC 7696). L'article de S. Bellovin et E. Rescorla, « "*Deploying a New Hash Algorithm*" <https://www.cs.columbia.edu/~smb/papers/new-hash.pdf> » montre clairement que les choses ne se passent pas aussi bien qu'elles le devraient.

Comme dit plus haut, la meilleure façon de s'assurer qu'un mécanisme est utilisable est de l'utiliser effectivement. Et pour cela que ce mécanisme soit indispensable au fonctionnement normal du protocole, qu'on ne puisse pas l'ignorer. Le RFC cite l'exemple de SMTP : le principal mécanisme d'extension de SMTP est d'ajouter de nouveaux en-têtes (comme le `Archived-At:` dans le RFC 5064) or, SMTP dépend d'un traitement de ces en-têtes pour des fonctions mêmes élémentaires. Un MTA ne peut pas se permettre d'ignorer les en-têtes. Ainsi, on est sûr que toute mise en œuvre de SMTP sait analyser les en-têtes. Et, comme des en-têtes nouveaux sont assez fréquemment ajoutés, on sait que des en-têtes inconnus des vieux logiciels ne perturbent pas SMTP. Ce cas est idéal : au lieu d'un mécanisme d'extension qui serait certes spécifié dans le RFC mais pas encore utilisé, on a un mécanisme d'extension dont dépendent des fonctions de base du protocole.

Le RFC cite également le cas de SIP, qui est moins favorable : les relais ne transmettent en général pas les en-têtes inconnus, ce qui ne casse pas la communication, mais empêche de déployer de nouveaux services tant que tous les relais n'ont pas été mis à jour.

Bien sûr, aucune solution n'est parfaite. Si SMTP n'avait pas ajouté de nombreux en-têtes depuis sa création, on aurait peut-être des programmes qui certes savent analyser les en-têtes mais plantent lorsque ces en-têtes ne sont pas dans une liste limitée. D'où l'importance de changer souvent (ici, en ajoutant des en-têtes).

Souvent, les protocoles prévoient un mécanisme de négociation de la version utilisée. On parle de version différente lorsque le protocole a suffisamment changé pour qu'on ne puisse pas interagir avec un vieux logiciel. Un client SMTP récent peut toujours parler à un vieux serveur (au pire, le serveur ignorera les en-têtes trop récents) mais une machine TLS 1.2 ne peut pas parler à une machine TLS 1.3.

Dans le cas le plus fréquent, la machine récente doit pouvoir parler les deux versions, et la négociation de version sert justement à savoir quelle version utiliser. Le problème de cette approche est que, quand la version 1 est publiée en RFC, avec son beau mécanisme de négociation de version, il n'y a pas encore de version 2 pour tester que cette négociation se passera bien. Celle-ci ne sortira parfois que des années plus tard et on s'apercevra peut-être à ce moment que certains programmes ont mal mis en œuvre la négociation de version, voire ont tout simplement négligé cette section du RFC…

Première solution à ce problème, utiliser un mécanisme de négociation de version située dans une couche plus basse. Ainsi, IP a un mécanisme de négociation de version dans l'en-tête IP lui-même (les quatre premiers bits indiquent le numéro de version, cf. RFC 8200, section 3) mais ce mécanisme n'a jamais marché en pratique. Ce qui marche, et qui est utilisé, c'est de se servir du mécanisme de la couche 2. Par exemple, pour Ethernet (RFC 2464), c'est l'utilisation du type de protocole ("*EtherType*"), 0x800 pour IPv4 et 0x86DD pour IPv6. Un autre exemple est l'utilisation d'ALPN (RFC 7301) pour les protocoles au-dessus de TLS. (Le RFC cite aussi la négociation de contenu de HTTP.)

Une solution récente au problème de l'ossification est le **graissage**, présenté en section 3.3. Décrit à l'origine pour TLS, dans le RFC 8701, il est désormais utilisé dans d'autres protocoles comme QUIC <https://www.bortzmeyer.org/quic.html>. Son principe est de réserver un certain nombre de valeurs utilisant des extensions du protocole et de s'en servir, de façon à forcer les différents logiciels, intermédiaires ou terminaux, à lire tout le RFC et à gérer tous les cas. Dans l'exemple cité plus haut d'un protocole qui aurait un champ de huit bits « cet octet est toujours à zéro dans cette version mais les récepteurs doivent ignorer son contenu car il pourra servir dans une future version », on peut réserver les valeurs 3, 21, 90 et 174 comme valeurs de graissage et l'émetteur les mettra de temps en temps dans le champ en question. Dans le cas de TLS, où le client propose des options et le serveur accepte celles qu'il choisit dans ce lot (oui, je sais, TLS est plus compliqué que cela, par exemple lorsque le serveur demande une authentification), le client annonce au hasard des valeurs de graissage. Ainsi, une "*middlebox*" qui couperait les connexions TLS serait vite détectée et, on peut l'espérer, rejetée par le marché. Le principe du graissage est donc « les extensions à un protocole s'usent quand on ne s'en sert pas ». Cela évoque les tests de "*fuzzing*" qu'on fait en sécurité, où on va essayer plein de valeurs inhabituelles prises au hasard, pour s'assurer que le logiciel ne se laisse pas surprendre. On voit aussi un risque du graissage : si les programmes bogués sont nombreux, le premier qui déploie un mécanisme de graissage va essuyer les plâtres et se faire parfois rejeter. Il est donc préférable que le graissage soit utilisé dès le début, par exemple dans un nouveau protocole.

Évidemment, la solution n'est pas parfaite. On peut imaginer un logiciel mal fait qui reconnait les valeurs utilisées pour le graissage (et les ignore) mais rejette quand même les autres valeurs pourtant légitimes. (La plupart du temps, les valeurs réservées pour le graissage ne sont pas continues, comme dans les valeurs 3, 21, 90 et 174 citées plus haut, pour rendre plus difficile un traitement spécifique au graissage.)

Bien sûr, même sans valeurs réservées au graissage, un programme pourrait toujours faire à peu près l'équivalent, en utilisant les valeurs réservées pour des expérimentations ou bien des usages privés. Mais le risque est alors qu'elles soient acceptées par l'autre partie, ce qui n'est pas le but.

Tous les protocoles n'ont pas le même style d'interaction que TLS, et le graissage n'est donc pas systématiquement possible. Et puis il ne teste qu'une partie des capacités du protocole, donc il ne couvre pas tous les cas possibles. Le graissage est donc utile mais ne résout pas complètement le problème.

Parfois, des protocoles qui ne permettaient pas facilement l'extensibilité ont été assez sérieusement modifiés pour la rendre possible. Ce fut le cas du DNS avec EDNS (RFC 6891). Son déploiement n'a pas été un long fleuve tranquille et, pendant longtemps, les réactions erronées de certains serveurs aux requêtes utilisant EDNS nécessitaient un mécanisme de repli (re-essayer sans EDNS). Il a ensuite fallu

supprimer ce mécanisme de repli <https://dnsflagday.net/2019/> pour être sûr que les derniers systèmes erronés soient retirés du service, le tout s'étalant sur de nombreuses années. Un gros effort collectif a été nécessaire pour parvenir à ce résultat, facilité, dans le cas du DNS, par le relativement petit nombre d'acteurs et leur étroite collaboration.

La section 4 du RFC cite d'autres techniques qui peuvent être utilisées pour lutter contre l'ossification. D'abord, ne pas avoir trop de possibilités d'étendre le protocole car, dans ce cas, certaines possibilités seront fatalement moins testées que d'autres et donc plus fragiles si on veut s'en servir un jour.

Le RFC suggère aussi l'utilisation d'**invariants**. Un invariant est une promesse des auteurs du protocole, assurant que cette partie du protocole ne bougera pas (et, au contraire, que tout le reste peut bouger et qu'il ne faut pas compter dessus). Si les auteurs de logiciels lisent les RFC (une supposition audacieuse, notamment pour les auteurs de "*middleboxes*"), cela devrait éviter les problèmes avec les évolutions futures. (Il est donc toujours utile de graisser les parties du protocole qui ne sont pas des invariants.) Le RFC 5704, dans sa section 2.2, définit plus rigoureusement ce que sont les invariants. Deux exemples d'utilisation de ce concept sont le RFC 8999 (décrivant les invariants de QUIC <https://www.bortzmeyer.org/quic.html>) et la section 9.3 du RFC 8446 sur TLS. Notre RFC conseille aussi de préciser explicitement ce qui n'est **pas** invariant (comme le fait l'annexe A du RFC 8999), ce que je trouve contestable (il y a un risque que cette liste d'exemples de non-invariants soit interprétée comme limitative, alors que ce ne sont que des exemples).

Une autre bonne technique, que recommande notre RFC, est de prendre des mesures techniques pour empêcher les intermédiaires de tripoter la communication. Moins il y a d'entités qui analysent et interprètent les paquets, moins il y aura de programmes à vérifier et éventuellement à modifier en cas d'extension du protocole. Un bon exemple de ces mesures techniques est évidemment la cryptographie. Ainsi, chiffrer le fonctionnement du protocole, comme le fait QUIC <https://www.bortzmeyer.org/quic.html>, ne sert pas qu'à préserver la vie privée de l'utilisateur, cela sert aussi à tenir à l'écart les intermédiaires, ce qui limite l'ossification. Le RFC 8558 est une bonne lecture à ce sujet.

Bien sûr, si bien conçu que soit le protocole, il y aura des problèmes. Le RFC suggère donc aussi qu'on crée des mécanismes de retour, permettant d'être informés des problèmes. Prenons par exemple un serveur TLS qui refuserait des clients corrects, mais qui utilisent des extensions que le serveur ne gère pas correctement. Si le serveur ne journalise pas ces problèmes, ou bien que l'administrateur système ne lit pas ces journaux, le problème pourra trainer très longtemps. Idéalement, ces retours seront récoltés et traités automatiquement, et envoyés à celles et ceux qui sont en mesure d'agir. C'est plus facile à dire qu'à faire, et cela dépend évidemment du protocole utilisé. Des exemples de protocoles ayant un tel mécanisme sont DMARC (RFC 7489, avec son étiquette `rua`, section 7), et SMTP (avec le `TLSRPT` du RFC 8460).

Enfin, pour terminer, l'annexe A du RFC présente quelques exemples de protocoles et comment ils ont géré ce problème. Elle commence par le DNS (RFC 1034 et RFC 1035). Le DNS a une mauvaise expérience avec le déploiement de nouveaux types d'enregistrement qui, par exemple, provoquaient des rejets violents des requêtes DNS par certains équipements. C'est l'une des raisons pour lesquelles SPF (RFC 7208) a finalement renoncé à utiliser son propre type d'enregistrement, `SPF` (cf. RFC 6686) et pour lesquelles tout le monde se sert de `TXT`. Le problème s'est heureusement amélioré depuis la parution du RFC 3597, qui normalise le traitement des types inconnus. (Mais il reste d'autres obstacles au déploiement de nouveaux types, comme la mise à jour des interfaces « conviviales » d'avitaillement d'une zone DNS, interfaces qui mettent de nombreuses années à intégrer les nouveaux types.)

HTTP, lui, a plutôt bien marché, question extensibilité mais il a certains mécanismes d'extension que personne n'ose utiliser et qui ne marcheraient probablement pas, comme les extensions des "*chunks*" (RFC 7230, section 4.1.1) ou bien comme l'utilisation d'autres unités que les octets <https://www.

iana.org/assignments/http-parameters/http-parameters.xml#range-units> dans les requêtes avec intervalles (RFC 7233, section 2.2).

Et IP lui-même ? Par exemple, IP avait un mécanisme pour permettre aux équipements réseau de reconnaitre la version utilisée, permettant donc de faire coexister plusieurs versions d'IP, le champ Version (les quatre premiers bits du paquet). Mais il n'a jamais réellement fonctionné, les équipements utilisant à la place les indications données par la couche inférieure (l'« "*ethertype*" », dans le cas d'Ethernet, cf. RFC 2464). D'autres problèmes sont arrivés avec IP, par exemple l'ancienne « classe E ». Le préfixe 224.0.0.0/3 avait été réservé par le RFC 791, section 3.2, mais sans précisions (« "*The extended addressing mode is undefined. Both of these features are reserved for future use.*" »). Le RFC 988 avait ensuite pris 224.0.0.0/4 pour le "*multicast*" (qui n'a jamais été réellement déployé sur l'Internet), créant la « classe D », le reste devenant la « classe E », 240.0.0.0/4 dont on ne sait plus quoi faire aujourd'hui, même si certains voudraient la récupérer pour prolonger l'agonie d'IPv4. Son traitement spécial par de nombreux logiciels empêche de l'affecter réellement. Une solution souvent utilisée pour changer la signification de tel ou tel champ dans l'en-tête est la négociation entre les deux parties qui communiquent, mais cela ne marche pas pour les adresses (la communication peut être unidirectionnelle).

SNMP n'a pas eu trop de problèmes avec le déploiement de ses versions 2 et 3. La norme de la version 1 précisait clairement (RFC 1157) que les paquets des versions supérieures à ce qu'on savait gérer devaient être ignorés silencieusement, et cela était vraiment fait en pratique. Il a donc été possible de commencer à envoyer des paquets des versions 2, puis 3, sans casser les vieux logiciels qui écoutaient sur le même port.

TCP, par contre, a eu bien des problèmes avec son mécanisme d'extension. (Lire l'article de Honda, M., Nishida, Y., Raiciu, C., Greenhalgh, A., Handley, M., et H. Tokuda, « "*Is it still possible to extend TCP ?*" <https://conferences.sigcomm.org/imc/2011/docs/p181.pdf> ».) En effet, de nombreuses "*middleboxes*" se permettaient de regarder l'en-tête TCP sans avoir lu et compris le RFC, jetant des paquets qui leur semblaient anormaux, alors qu'ils avaient juste des options nouvelles. Ainsi, le "*multipath TCP*" (RFC 6824) a été difficile à déployer (cf. l'article de Raiciu, C., Paasch, C., Barre, S., Ford, A., Honda, M., Duchêne, F., Bonaventure, O., et M. Handley, « " *How Hard Can It Be ? Designing and Implementing a Deployable Multipath TCP*" <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/raiciu> »). "*TCP Fast Open*" (RFC 7413) a eu moins de problèmes sur le chemin, mais davantage avec les machines terminales, qui ne comprenaient pas ce qu'on leur demandait. Comme le but de "*TCP Fast Open*" était d'ouvrir une connexion rapidement, ce problème était fatal : on ne peut pas entamer une négociation lorsqu'on veut aller vite.

Et enfin, dernier protocole étudié dans cette annexe A, TLS. Là, la conclusion du RFC est que le mécanisme de négociation et donc d'extension de TLS était correct, mais que la quantité de programmes mal écrits et qui réagissaient mal à ce mécanisme l'a, en pratique, rendu inutilisable. Pour choisir une version de TLS commune aux deux parties qui veulent communiquer de manière sécurisée, la solution était de chercher la plus haute valeur de version commune (HMSV pour "*highest mutually supported version*", cf. RFC 6709, section 4.1). Mais les innombrables bogues dans les terminaux, et dans les "*middleboxes*" (qui peuvent accéder à la négociation TLS puisqu'elle est en clair, avant tout chiffrement) ont fait qu'il était difficile d'annoncer une version supérieure sans que les paquets soient rejetés. (Voir l'expérience racontée dans ce message <https://mailarchive.ietf.org/arch/msg/tls/bOJ2JQc3HjAHFFWCiNTIb0JuMZc>.)

TLS 1.3 (RFC 8446) a donc dû abandonner complètement le mécanisme HMSV et se présenter comme étant du 1.2, pour calmer les "*middleboxes*" intolérantes, dissimulant ensuite dans le ClientHello des informations qui permettent aux serveurs 1.3 de reconnaitre un client qui veut faire du 1.3 (ici, vu avec tshark) :

```
    TLSv1 Record Layer: Handshake Protocol: Client Hello
        Content Type: Handshake (22)
        Version: TLS 1.0 (0x0301)
        Length: 665
        Handshake Protocol: Client Hello
            Handshake Type: Client Hello (1)
            Length: 661
            Version: TLS 1.2 (0x0303)

[Ce champ Version n'est là que pour satisfaire les middleboxes.]

            Extension: supported_versions (len=9)
                Type: supported_versions (43)
                Length: 9
                Supported Versions length: 8
                Supported Version: TLS 1.3 (0x0304)
                Supported Version: TLS 1.2 (0x0303)
                Supported Version: TLS 1.1 (0x0302)
                Supported Version: TLS 1.0 (0x0301)

[La vraie liste des versions acceptées était dans cette extension.]
```

Toujours à propos de TLS, SNI ("*Server Name Indication*", cf. RFC 6066, section 3) est un autre exemple où la conception était bonne mais le manque d'utilisation de certains options a mené à l'ossification et ces options ne sont plus, en pratique, utilisables. Ainsi, SNI permettait de désigner le serveur par son nom de domaine (`host_name` dans SNI) mais aussi en utilisant d'autres types d'identificateurs (champ `name_type`) sauf que, en pratique, ces autres types n'ont jamais été utilisés et ne marcheraient probablement pas. (Voir l'article de A. Langley, « "*Accepting that other SNI name types will never work*" <https://mailarchive.ietf.org/arch/msg/tls/1t79gzNItZd71DwwoaqcQQ_4Yxc> ».)