RFC 6709 : Design Considerations for Protocol Extensions

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 15 septembre 2012

Date de publication du RFC : Septembre 2012

https://www.bortzmeyer.org/6709.html

Concevoir des protocoles de communication monolithiques qui n'évoluent jamais est une chose. Concevoir des protocoles qui soient **extensibles**, qui puissent, après leur conception initiale, évoluer par le biais d'**extensions**, est un exercice plus difficile. Cet excellent RFC de l'IAB décrit comment faire un protocole extensible et comment assurer ainsi une évolution heureuse. C'est un document avant tout destiné aux auteurs de protocoles, mais qui intéressera aussi les étudiants en réseaux informatiques et tous les techniciens curieux, qui veulent savoir comment ça marche.

Parmi les innombrables exemples de protocoles IETF qui ont été étendus, on peut citer les extensions d'internationalisation de SMTP dans le RFC 6531 [1], la recherche floue dans IMAP avec le RFC 6203, ou les capacités BGP du RFC 5492.

Le RFC 5218 (section 2.2.1), qui décrivait les raisons du succès pour un protocole réseau, notait parmi celles-ci l'extensibilité. Beaucoup de protocoles IETF ont des mécanismes pour cette extensibilité, qui est largement reconnue comme un bon principe. Mais, si elle est mal faite, des ennuis peuvent survenir, et cela peut menacer l'interopérabilité, voire la sécurité. Le RFC sur SMTP, le RFC 5321, note ainsi (section 2.2.1) que les protocoles avec peu d'options sont partout et que ceux avec beaucoup d'options sont nulle part. Un RFC comme le RFC 1263 (consacré aux extensions de TCP) critiquait également l'abus des extensions, et ce dès 1991. Malgré cela, d'innombrables RFC normalisent des extensions à tel ou tel protocole, mais il n'existait pas encore de document d'architecture sur le principe des extensions, rôle que ce RFC ambitionne de jouer.

D'abord, une distinction importante (section 2) entre extension « majeure » et « de routine ». Les secondes sont celles qui ne modifient pas le protocole de manière substantielle (et peuvent donc être introduites avec un minimum d'examen), les premières changent le protocole sérieusement et doivent

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc6531.txt

donc être étudiées de près. Le problème est particulièrement aigu si l'extension est faite par d'autres personnes que le protocole originel : les « extensionneurs » risquent de ne pas connaître certaines règles non écrites que les auteurs originaux avaient en tête. Pour éviter cela, la spécification originale d'un protocole devrait toujours expliciter les règles, et décrire comment étendre le protocole proprement.

Comment distinguer une extension majeure d'une extension de routine ? Les extensions majeures sont celles qui peuvent empêcher l'interopérabilité (la capacité de deux pairs à communiquer), perturber le fonctionnement de l'Internet, ou mettre en péril la sécurité. Cela se produit notamment :

— Si l'extension nécessite que les mises en œuvre existantes soient modifiées. Par exemple, si un protocole a un champ de taille fixe `Foobar` sur un octet et que, les 256 valeurs étant occupés, on passe à un champ de deux octets, les nouvelles mises en œuvre ne pourront pas parler aux anciennes, tout l'en-tête étant décalé à partir de ce champ (uniquement pour un octet ajouté). Il existe des méthodes pour éviter ces problèmes (par exemple utiliser des TLV au lieu des champs de taille fixe) mais elles ont aussi des inconvénients (de performance, par exemple) et, de toute façon, nécessitent d'avoir été mises dès le début dans le protocole. Un autre exemple est celui d'un protocole où les messages ont un type et où tout type inconnu fait couper la communication. Dans ce cas, ajouter un type de message, même optionnel, représente une extension majeure (là aussi, il existe des solutions, comme de spécifier dès le début que les messages de type inconnu doivent être silencieusement ignorés, cf. section 4.7).
— Si l'extension change fondamentalement le modèle sous-jacent. C'est d'autant plus difficile à détecter que ce modèle n'est pas toujours explicite. Bien des RFC décrivent au bit près le format des messages mais sont très courts sur la vision de haut niveau du protocole. Le RFC donne l'exemple d'une extension qui ajouterait un état à un protocole qui était précédemment sans état.
— Des nouveaux usages du protocole peuvent aussi représenter une extension majeure. Par exemple, si des changements purement quantitatifs se produisent (trafic accru, paquets plus gros comme dans le cas d'EDNS ou encore algorithmes qui ne supportent pas le passage à l'échelle d'une utilisation plus intense).
— Si on change la syntaxe du protocole (par exemple si on inverse deux champs).
— Si on change le modèle de sécurité.
— Et, bien sûr, si on crée une extension qui ne « rentre pas » dans le modèle d'extension originellement envisagé.

Et les extensions de routine, alors, elles ressemblent à quoi ?

— Elle n'ont aucune des caractéristiques citées plus haut, qui identifient les extensions majeures.
— Elle sont opaques au protocole (les messages avec la nouvelle extension sont traités par le protocole comme ceux d'avant).

Une extension de routine ne nécessitera aucun changement sur les vieux programmes. Seuls ceux qui voudront utiliser la nouvelle extension auront à être modifiés.

Quelques exemples d'extensions de routine : les options spécifiques à un fournisseur de DHCP (RFC 2132), les options spécifiques à un fournisseur de Radius (RFC 2865), les types MIME de l'arbre `vnd` (*"vendor"*), etc. Ce n'est que lorsqu'on est sûr que l'ajout d'une extension ne sera que de la routine qu'on peut avoir des politiques du genre « premier arrivé, premier servi » (RFC 5226), c'est-à-dire peu ou pas d'examen préalable. Et encore, l'expérience a montré que même les extensions de routine pouvaient poser des problèmes inattendus et qu'un examen de toutes les extensions par un expert est souvent une bonne chose.

Une fois posée cette importante distinction entre extensions de routines et extensions majeures, la section 3 de notre RFC s'attache des bons principes architecturaux pour assurer une extensibilité heureuse :

— Limiter l'extensibilité à ce qui est « raisonnable »,
— Penser avant tout à l'interopérabilité,
— Garder les extensions compatibles avec le protocole de base,
— Ne pas permettre des extensions qui créent un nouveau protocole, incompatible avec le précédent,
— Permettre de tester les extensions,

— Bien coordonner les allocations de paramètres (noms, numéros, etc),
— Faire **davantage** attention aux extensions aux composants critiques de l'Internet comme le DNS ou BGP.

Premier principe, limiter l'extensibilité. Il faut essayer de ne traiter que les cas les plus courants et ne pas chercher l'extensibilité à tout prix. C'est évidemment plus facile à dire qu'à faire. Comme le note le RFC 5218, un protocole qui rencontre un « succès fou » est utilisé dans des contextes que ses concepteurs n'imaginaient pas. Alors, essayer de déterminer à l'avance les extensions qui vont être vraiment utiles... On peut donc être amené à réviser le mécanisme d'extension en cours de route.

Deuxième principe, l'interopérabilité, c'est-à-dire la capacité de deux mises en œuvre complètement différentes du même protocole à communiquer. Le RFC 4775 (section 3.1) insiste sur ce concept, qui est au cœur de la mission de l'IETF. Une extension qui empêcherait les nouveaux programmes de communiquer avec les anciens casserait cette interopérabilité et serait donc inacceptable.

D'autres familles de protocole que la famille TCP/IP avaient une approche différente : le protocole était tellement bourré d'options très larges que deux mises en œuvre toutes les deux conformes à la norme pouvaient néanmoins ne pas pouvoir communiquer car elles n'implémentaient pas les mêmes options. C'est ce qui est arrivé aux protocoles OSI et la solution adoptée avait été de créer des profils (un ensemble d'options cohérent), créant ainsi autant de protocoles que de profils (les profils différents n'interopéraient pas même si, sur le papier, c'était le même protocole). Notre RFC dit clairement que ces profils ne sont pas une solution acceptable. Le but est d'avoir des protocoles qui fonctionnent sur tout l'Internet (alors qu'OSI avait une vision beaucoup plus "*corporate*" où il n'était pas prévu de communications entre groupes d'utilisateurs différents).

En pratique, pour assurer cette interopérabilité, il faut que le mécanisme d'extension soit bien conçu, avec notamment une sémantique claire (par exemple, précisant ce que doit faire un programme qui rencontre un message d'un type inconnu, ou bien un message d'un type qui ne correspond pas à ce qu'il devrait recevoir dans son état actuel).

Le RFC note aussi qu'écrire les plus belles normes ne suffira pas si elles sont mal mises en œuvre. On constate que les problèmes liés aux extensions sont souvent dus à des implémentations qui ne le gèrent pas correctement. Un exemple est celui des options d'IP qui, bien que clairement spécifiées dès le début (RFC 791), ne sont toujours pas partout acceptées en pratique <https://www.bortzmeyer.org/options-interdites.html>.

Et puisqu'on parle d'interopérabilité, une idée qui semble bonne mais qui a créé beaucoup de problèmes : les extensions privées. Pas mal de programmeurs se sont dit « de toute façon, ce système ne sera utilisé que dans des réseaux fermés, tant pis pour l'interopérabilité sur le grand Internet, je peux, sur mon réseau, ajouter les extensions que je veux ». Mais les prédictions ne sont que rarement 100 % vraies. Les ordinateurs portables, les "*smartphones*" et autres engins se déplacent, et les implémentations ayant des extensions privées se retrouvent tôt ou tard à parler aux autres et... paf. Même chose lorsque deux extensions privées rentrent en collision, par exemple à la suite de la fusion de deux entreprises. Imaginons que chacune d'elles ait choisi de mettre du contenu local dans un attribut DHCP de numéro 62, en pensant que, DHCP n'étant utilisé que sur le réseau local, cela ne gênerait jamais. Des années après, une des deux entreprises rachète l'autre, les réseaux fusionnent et... repaf.

Pour résoudre ce problème, on a parfois fait appel à des espaces réservés comme le fameux préfixe « X- » devant le nom d'un paramètre (RFC 822). Le RFC 6648 a mis fin à cette pratique (qui était déjà retirée de plusieurs RFC), notamment parce que les paramètres « privés », tôt ou tard, deviennent publics, et qu'alors les ennuis commencent.

Un problème proche existe pour les paramètres considérés comme « expérimentaux ». Ils ont leur utilité, notamment pour que les programmeurs puissent tester leurs créations. C'est ainsi que le RFC 4727 propose des valeurs pour les expérimentations avec de nouveaux protocoles de transport, des nouveaux ports, etc. Le RFC 3692 décrit plus en détail les bonnes pratiques pour l'utilisation de valeurs expérimentales. Elles doivent être strictement limitées au système en cours de test, et jamais distribuées sur l'Internet. Si l'expérience est multi-site, c'est une bonne raison pour ne pas utiliser de valeurs expérimentales. C'est ainsi que des expériences comme HIP et LISP ont obtenu des paramètres enregistrés, non expérimentaux, pour pouvoir être testés en multi-sites.

À noter que ces valeurs expérimentales ne sont nécessaires que lorsque l'espace disponible est limité. Le champ `Next header` d'IPv6 ne fait qu'un octet et il est donc logique de bloquer deux valeurs réutilisables, 253 et 254, pour expérimenter avec des valeurs non officiellement enregistrés. Mais ce n'est pas le cas, par exemple, des champs de l'en-tête HTTP, composés d'un grand nombre de caractères et permettant donc des combinaisons quasi-infinies. Si on expérimente avec HTTP, il y a donc peu de raisons de réserver des champs expérimentaux.

Troisième principe architectural, la compatibilité avec le protocole de base. Elle implique un double effort : les auteurs du protocole original doivent avoir bien spécifié le mécanisme d'extensions. Et les auteurs d'extensions doivent bien suivre ce mécanisme. Quelques exemples de protocole où le mécanisme d'extensions est documenté en détail : EPP (RFC 3735), les MIB (RFC 4181), SIP (RFC 4485), le DNS (RFC 2671 et RFC 3597), LDAP (RFC 4521) et Radius (RFC 6158).

Quatrième principe d'architecture, faire en sorte que les extensions ne mènent pas à une scission, avec apparition d'un nouveau protocole, incompatible avec l'original. Des changements qui semblent très modérés peuvent empêcher l'interopérabilité, ce qui serait de facto un nouveau protocole. Un protocole trop extensible peut donc ne pas être une si bonne idée que cela, si cette extensibilité va jusqu'à la scission.

Comme le note le RFC 5704, écrit dans le contexte d'une grosse dispute entre l'IETF et l'UIT, la scission ("*fork*") peut venir de l'action non coordonnée de deux SDO différentes. Mais il n'y a pas toujours besoin d'être deux pour faire des bêtises : le RFC 5421 réutilisait un code EAP, rendant ainsi le protocole incompatible avec la précédente version. L'IESG avait noté le problème (documenté dans la section "*IESG Note*" du RFC 5421) mais avait laissé faire.

Cette notion de scission est l'occasion de revenir sur un concept vu plus haut, celui de **profil**. Il y a deux façons de profiler un protocole :

— Retirer des obligations (ce qui peut créer de gros problèmes d'interopérabilité),
— Ajouter des obligations (rendre obligatoire un paramètre qui était facultatif).

Contrairement à la première, la seconde approche peut être une bonne idée. En réduisant le nombre de choix, elle peut améliorer l'interopérabilité. Par contre, retirer des obligations est presque toujours une mauvaise idée. Ces obligations, dans un RFC (les fameux "*MUST*", "*SHOULD*" et "*MAY*"), sont décrites avec les conventions du RFC 2119 et ce RFC insiste bien sur le fait qu'elles doivent être soigneusement choisies. Par exemple, un "*MUST*" ne doit être utilisé que si la violation de cette obligation est effectivement dangereuse, par exemple pour l'interopérabilité. Si ce conseil a été suivi, changer un "*MUST*" en "*SHOULD*" dans un profil va donc être négatif à coup presque sûr. Et, le RFC rappelle aussi qu'un "*MAY*" veut dire « vous n'êtes pas obligés de l'implémenter » mais certainement pas « vous avez le droit de planter si l'autre machine l'utilise ».

Place au cinquième principe, les tests. On a vu que ce n'est pas tout de bien normaliser, il faut aussi bien programmer. Pour s'assurer que tout a été bien programmé, il faut tester. Cela implique d'avoir développé des jeux de tests, essayant les différents cas vicieux (par exemple, essayer toutes les possibilités du mécanisme d'extension, pour vérifier qu'aucune ne plante les programmes ne les mettant pas

en œuvre). Sinon, le test sera fait dans la nature, comme cela avait été le cas du fameux attribut 99 de BGP <https://www.bortzmeyer.org/bgp-attribut-99.html>. La norme spécifiait très clairement ce qu'un pair BGP devait faire d'un attribut inconnu, mais IOS n'en tenait pas compte et, lorsque quelqu'un a essayé d'utiliser réellement cette possibilité de la norme, il a planté une bonne partie de l'Internet européen... TLS, étudié plus en détail plus loin, fournit un autre exemple d'implémentations déplorables, bloquant le déploiement de nombreuses extensions.

Sixième grande question d'architecture, l'enregistrement des paramètres. Beaucoup de protocoles ont besoin de registres pour noter les valeurs que peuvent prendre tel ou tel champ (par exemple, il y a un registre des numéros de protocole de transport <https://www.iana.org/assignments/protocol-numbers/protocol-numbers.xml> et un registre des options DHCP <https://www.iana.org/assignments/bootp-dhcp-parameters/bootp-dhcp-parameters.xml#options>). Ces registres et leur bonne utilisation sont une question souvent sous-estimée de l'interopérabilté. Si un champ du paquet identifie une extension et que deux extensions incompatibles utilisent le même numéro, l'accident est inévitable. Cette collision peut être due à l'absence de registre mais on rencontre régulièrement des développeurs qui ne vérifient pas le registre et qui s'attribuent tout simplement un numéro qui leur semble libre. (Le RFC note, de manière politiquement correcte, qu'il ne peut pas « publier les noms sans embarrasser des gens ». Mais cela semble particulièrement fréquent pour les numéros de port <https://www.iana.org/assignments/service-names-port-numbers/service-names-po
xml> et pour les plans d'URI <https://www.iana.org/assignments/uri-schemes.html>.)

Les concepteurs de protocole devraient vérifier, d'abord si un des registres existants <https://www.iana.org/protocols/> ne suffit pas et, sinon en créer un. Dans ce cas, il faut soigneusement choisir la politique d'allocation des valeurs dans ce registre. Le RFC 5226 décrit les différentes politiques possibles, de la plus laxiste à la plus stricte. Attention : choisir une politique très stricte peut avoir des effets de bord désagréables, par exemple encourager les gens à « s'auto-enregistrer », c'est-à-dire à prendre un numéro au hasard (et tant pis pour l'interopérabilité) parce que la complexité et la lenteur des procédures les ont découragés. Il peut être préférable d'avoir une approche plus ouverte du registre, qui devrait documenter ce qui existe plutôt que d'essayer de contrôler tous les enregistrements.

Enfin, septième et dernier principe architectural à garder en tête pour faire des protocoles extensibles, l'importance particulière des services critiques. Si on arrête BGP, l'Internet ne fonctionne plus. Si on arrête le DNS, il n'y a quasiment plus aucune activité possible (même si vous ne voyez pas de noms de domaine, ils sont présents dans la communication). Même chose pour certains algorithmes comme le contrôle de congestion : si on modifie cet algorithme et qu'on se trompe, on peut mettre l'Internet à genoux. Il faut donc faire dix fois plus attention lorsqu'on étend ces protocoles critiques.

Le RFC cite un exemple BGP. Il existait depuis longtemps dans les IGP comme OSPF la notion de LSA opaque (LSA = "*Link State Attribute*", la structure de données qui contient les informations que s'échangent les routeurs). « Opaque » signifie que les routeurs n'ont pas forcément besoin de comprendre son contenu. Ils peuvent le propager sans l'analyser et cela permet des extensions intéressantes du protocole, en introduisant de nouveaux types de LSA opaques. L'expérience montre que cela marche bien.

Mais l'introduction de nouveaux attributs dans BGP ne s'est pas aussi bien passé puisque des routeurs ont réinitialisé les sessions locales si un attribut, transmis par un routeur lointain, était inconnu ou mal formé. (Cas du fameux attribut 99 <https://www.bortzmeyer.org/bgp-attribut-99.html>.)

Une fois ces sept principes en tête, la section 4 se penche sur le protocole de base. Une bonne extensibilité implique un bon protocole de base, conçu pour l'extensibilité. Cela implique :

— Une spécification bien écrite,

— Prenant en compte l'interaction avec des versions plus récentes ou plus anciennes du protocole. Cela inclut la détermination des capacités du pair, la négociation des paramètres, le traitement des extensions inconnues (nécessaire pour que les nouvelles implémentations ne plantent pas les anciennes), etc.
— Décrivant bien les principes architecturaux du protocole, ainsi que sa sécurité (pour éviter qu'une extension ne casse un modèle de sécurité bien fait mais pas assez explicite).
— Précisant clairement le modèle d'extensions et notamment quelles extensions sont majeures et lesquelles sont de routine.
— Interagissant le moins possible avec les composants critiques de l'Internet. Par exemple, si un nouveau protocole nécessite des modifications sérieuses au DNS, cela va faire hésiter.

Écrire un protocole bien extensible nécessite aussi de prêter attention à ce qui peut sembler des petits détails. Ainsi (section 4.1), le problème des numéros de versions. Il est courant que le protocole porte un tel numéro, permettant au pair de savoir tout de suite s'il a affaire à une implémentation équivalente (même numéro de version) ou pas (numéro de version plus récent ou plus ancien). Par exemple, voici une session TLS avec l'outil `gnutls-cli` de GnuTLS :

```
% gnutls-cli  --verbose  --port 443 svn.example.net
...
- Version: TLS1.2
```

où on voit que le serveur accepte TLS version 1.2. Un tel versionnement n'est utile que si le protocole de base précise bien ce qu'il faut faire lorsqu'on rencontre une version différente. Par exemple, si un logiciel en version 1.1 rencontre un 2.0 ? Il y a plusieurs sémantiques possibles, par exemple le protocole de base peut préciser que toute version doit être un sur-ensemble de la version antérieure (le logiciel en 2.0 doit accepter toutes les commandes du 1.1). Une autre sémantique pourrait être qu'il n'y a compatibilité qu'au sein d'une même version majeure (entre 1.0, 1.1, 1.2, etc) et que donc les deux logiciels de l'exemple (1.1 et 2.0) doivent renoncer à se parler. Dans les deux cas, cela doit être clairement spécifié.

Un contre-exemple est fourni par la norme MIME. Le RFC 1341 décrivait un en-tête `MIME-Version:` mais sans dire ce que devait faire une implémentation correcte de MIME si elle rencontrait une version inférieure ou supérieure. En fait, ce RFC 1341 ne précisait même pas le format de la version ! Le RFC 1521 a un peu précisé les choses mais pas au point qu'on puisse prédire le comportement d'une mise en œuvre de MIME face à des versions nouvelles. Notre RFC 6709 estime donc que le `MIME-Version:` n'a guère d'utilité pratique.

Pour un exemple plus positif, on peut regarder ROHC. Le RFC 3095 décrivait un certain nombre de jeux de paramètres pour la compression. À l'usage, on leur a découvert des limitations, d'où le développement de ROHCv2 (RFC 5225). Ce dernier n'est pas compatible mais ne plante pas les ROHC précédents, ils ont simplement des jeux de paramètres différents et la négociation initiale (qui n'a pas changé en v2) permet à un ROHC ayant les jeux v1 de se comprendre facilement avec un plus récent.

Quelles sont les stratégies utilisées par les protocoles pour la gestion des versions ? Il y en a plein, toutes n'étant pas bonnes. On trouve :

— Pas de gestion de versions. Le cas le plus célèbre est EAP (RFC 3748) mais c'est aussi le cas de Radius (RFC 2865). Cela les a protégé contre l'apparition de multiples versions à gérer mais, comme la demande d'extensibilité est toujours là, elle s'est manifestée de manière non officielle (voir l'étude de cas de Radius en annexe A.2).
— HMSV ("*Highest mutually supported version*") : les deux pairs échangent les numéros de version du protocole qu'ils savent gérer et on choisit la plus élevée qui soit commune aux deux. Cela implique que la version plus élevée est toujours « meilleure » (alors que, parfois, une version supérieure retire des fonctions). Cela implique aussi que gérer une version implique aussi de gérer toutes les versions inférieures (si un pair annonce 3.5 et un autre 1.8, HMSV va choisir 1.8 alors que le pair en 3.5 ne sait pas forcément gérer cette vieille version).

— Min/Max : une variante de HMSV où on annonce non pas un numéro de version (la plus élevée qu'on gère) mais deux (la version la plus élevée qu'on gère, et la moins élevée). Ainsi, il n'y a pas besoin de gérer éternellement les vieilles versions. Dans certains cas, cela peut mener à l'échec de la négociation. Si Alice annonce 2.0-¿3.5 et que Bob répond avec 1.0-¿1.8, ils n'auront aucune version compatible (Bob est trop vieux). Au moins, cela permettra d'envoyer un message d'erreur clair.
— Considérer que les vieilles versions sont compatibles avec les récentes. L'idée est que l'implémentation récente peut envoyer ce qu'elle veut, l'ancienne devant réagir proprement (typiquement en ignorant les nouveaux messages et types). Cela a pour conséquence qu'une nouvelle version du protocole ne peut pas rendre obligatoire un message qui n'existait pas avant (puisque les vieilles implémentations ne le connaissent pas). Avec cette stratégie, qui est par exemple utilisée dans 802.1X, un logiciel n'est jamais dépassé, il peut toujours travailler avec ses successeurs.
— Majeur/mineur : pour les numéros de versions à deux champs (comme 3.5 et 1.8 plus tôt), on considère qu'il y a compatibilité tant que le champ majeur ne change pas. 3.5 peut communiquer sans problème avec 3.0 mais aussi avec 3.9. Mais 3.5 ne peut pas parler avec 2.4 ou 5.2. Les contraintes de la stratégie précédente (ignorer les types inconnus, ne pas rendre un nouveau type obligatoire) ne s'appliquent donc qu'au sein d'un même numéro majeur.

Et c'est à peu près tout. Le RFC considère qu'il ne faut pas chercher d'autres stratégies de versionnement, qui ont de fortes chances d'être trop complexes pour être mises en œuvre sans erreur.

Autre chose importante à considérer lorsqu'on normalise le protocole de base (la première version) : les champs réservés. On trouve souvent dans les RFC des phrases comme (celle-ci est extraite du RFC 1035) « "*Reserved for future use. Must be zero in all queries and responses.*" ». (Dans le cas du DNS, ils ont été utilisés par la suite pour DNSSEC <https://www.iana.org/assignments/dns-parameters>, bits AD et CD. Le bit 9, selon le RFC 6195, section 2.1, avait été utilisé de manière non standard autrefois et semble donc inutilisable désormais.) Certains champs dans l'en-tête sont donc gardés pour un usage futur et, presque toujours, le RFC demande à l'envoyeur de les mettre tous à zéro et au récepteur de les ignorer, ce qui est très important : autrement, on ne pourra jamais déployer d'implémentation d'une nouvelle version du protocole, qui utiliserait ces champs qui avaient été réservés. Dans les diagrammes montrant les paquets, ces champs sont en général marqués Z (pour "*Zero*") ou MBZ ("*Must Be Zero*", voir par exemple le RFC 4656). Notons que bien des pare-feux, fidéles jusqu'à l'excès au principe « tout ce qui est inconnu est interdit », au lieu d'ignorer ces champs, comme l'impose le RFC, jettent les paquets où ce champ ne vaut pas zéro. Ces engins sont une des plus grosses sources de problème lorsqu'on déploie un nouveau protocole. Donc, attention, programmeurs, votre travail n'est pas de vérifier ces champs mais de les **ignorer**, afin de permettre les futures extensions.

Autre problème douloureux, la taille à réserver aux champs (section 4.4). Certains protocoles utilisent des champs de taille illimitée (paramètres de type texte) ou suffisamment grands pour qu'on ne rencontre jamais la limite, même après plusieurs extensions du protocole. Mais ce n'est pas toujours le cas. Tout le monde se souvient évidemment de l'espace d'adressage trop petit <https://www.bortzmeyer.org/epuisement-adresses-ipv4.html> d'IPv4 et de la difficulté qu'il y a eu à passer aux adresses plus grandes d'IPv6 (le champ Adresse étant de taille fixe, les deux versions ne pouvaient pas être compatibles). Dans la réalité, on rencontre :

— Des champs Version trop petits, par exemple de seulement deux bits (après la version de développement et la version de production, on n'a plus que deux versions possibles).
— Des champs stockant un paramètre pour lesquels la taille était trop petite pour la politique d'allocation (cf. RFC 5226) qui avait été adoptée. Par exemple, la politique FCFS ("*First-come, First-served*", ou « Premier arrivé, premier servi »), la plus libérale, peut vite épuiser un champ de seulement huit bits (le problème s'est posé pour le champ "*Method Type*" d'EAP dans le RFC 2284).

Si un tel problème se pose, si on voit les valeurs possibles pour un champ arriver à un tel rythme que l'épuisement est une possibilité réelle, il y a plusieurs solutions :

— Changer la politique d'allocation en la durcissant. Mais attention, une politique trop dure peut mener à des comportements comme l'auto-allocation (« je prends la valeur qui me plait et tant pis pour le registre »). C'est ce qui avait été fait pour EAP (RFC 3748, section 6.2) en passant de « Premier arrivé, premier servi » à « Norme nécessaire ». Pour éviter l'auto-allocation, diverses solutions alternatives avaient également été normalisées par ce RFC, comme la suivante :
— Permettre des valeurs spécifiques à un fournisseur ("*vendor*"). Ainsi, chaque fournisseur qui crée une extension privée, spécifique à son logiciel, peut créer ses propres types/messages.
— Agrandir le champ. La plupart des protocoles IETF ont des champs de taille fixe, pour des raisons de performance (il faut se rappeler qu'il peut y avoir **beaucoup** de paquets à traiter par seconde, ce n'est donc pas une optimisation de détail). Agrandir le champ veut donc en général dire passer à une nouvelle version, incompatible, du protocole.
— Récupérer des valeurs inutilisées. C'est très tentant sur le principe (il est rageant d'épuiser un espace de numérotation alors qu'on sait que certaines valeurs ont été réservées mais jamais utilisées) mais cela marche mal en pratique car il est en général très difficile de dire si une valeur est vraiment inutilisée (peut-être qu'une obscure implémentation utilisée uniquement dans un lointain pays s'en sert ?)

Un usage particulier de l'extensibilité concerne la cryptographie (section 4.5). Celle-ci demande une **agilité**, la capacité à changer les algorithmes de cryptographie utilisés, pour faire face aux progrès constants de la cryptanalyse (c'est bien expliqué dans la section 3 du RFC 4962). Presque tous les protocoles de cryptographie sérieux fournissent cette agilité (le RFC ne donne pas d'exemple mais je peux citer deux contre-exemples, un à l'IETF, le TCP-MD5 du RFC 2385 depuis remplacé par le meilleur RFC 5925 qui est encore très peu déployé et, en dehors de l'IETF, le protocole DNScurve <`https://www.bortzmeyer.org/dnscurve.html`>). Le problème est lorsqu'un algorithme noté comme obligatoire (avant d'assurer l'interopérabilité, un ou plusieurs algorithmes sont obligatoires à implémenter, pour être sûr que deux pairs trouvent toujours un algorithme commun) est compromis au point qu'il n'est plus raisonnable de s'en servir.

Soit un algorithme non compromis est déjà très répandu dans les mises en œuvre existantes du protocole et on peut alors le déclarer obligatoire, et noter le vieil algorithme comme dangereux à utiliser. Soit il n'existe pas encore un tel algorithme répandu et on a peu de solutions : déployer du code dans la nature prend beaucoup de temps. Bref, le RFC recommande que, lorsque les signes de faiblesse d'un algorithme face à la cryptanalyse apparaissent (le RFC ne cite pas de noms mais, en 2012, on peut dire que c'est le cas de SHA-1 et RSA), on normalise, puis on pousse le déploiement de solutions de rechange (par exemple SHA-2 et ECDSA dans ces deux cas, algorithmes qui sont aujourd'hui utilisables dans la plupart des protocoles IETF). Il faut bien se rappeler que le déploiement effectif d'un algorithme normalisé peut prendre des années, pendant lesquelles la cryptanalyse continuera à progresser.

Un aspect peu connu des protocoles applicatifs de la famille IP est qu'ils peuvent parfois tourner sur plusieurs transports, par exemple TCP **et** SCTP (section 4.6). Cela a parfois été fait dès le début, pour avoir le maximum de possibilités mais, parfois, le protocole de couche 7 n'était prévu que pour un seul protocole de couche 4, puis a été étendu pour un autre protocole. Par exemple, utilisant TCP, il ne passait pas à l'échelle, alors on l'a rendu utilisable sur UDP. (Dans ce cas, les concepteurs du protocole devraient lire le RFC 8085 car il y a plein de pièges.)

Cela peut évidemment casser l'interopérabilité. Si le protocole Foobar est normalisé pour tourner sur TCP ou SCTP, deux mises en œuvre de Foobar, une qui ne gère que TCP et une qui ne gère que SCTP (pensez aux systèmes embarqués, avec peu de ressources, ce qui impose de limiter les choix), ne pourront pas se parler. Notre RFC conseille donc de bien réflechir à ce problème. Davantage de choix n'est pas forcément une bonne chose. D'un autre côté, si le premier protocole de transport choisi a de sérieuses limites, on pourra en ajouter un autre comme option mais il sera quasi-impossible de supprimer le premier sans casser l'interopérabilité.

Enfin, dernier casse-tête pour le concepteur de protocoles, le cas des extensions inconnues (section 4.7). Je suis programmeur, je crée un programme pour la version 1.0 du protocole, ce programme sait

traiter un certain nombre de messages. Le temps passe, une version 1.1 du protocole est créé, des programmeurs la mettent en œuvre et, soudain, les copies de mon programme reçoivent des messages d'un type inconnu, car introduit dans la 1.1. Que faire ? Première approche, une règle comme quoi les extensions inconnues doivent être ignorées ("*silently discarded*"). Je reçois un tel message, je le jette. Un champ qui était à zéro acquiert une signification ? J'ignore ce champ. L'avantage de cette approche est qu'elle permet le maximum d'interopérabilité. On est sûr que les vieilles versions pourront parler avec les nouvelles, même si seules les nouvelles pourront tirer profit des extensions récentes. L'inconvénient de cette approche est qu'on ne sait même pas si les nouveaux messages sont traités ou pas. Dans le cas de la sécurité, on voudrait bien pouvoir savoir si une extension qu'on considère comme indispensable a bien été acceptée par le pair (voir par exemple la section 2.5 du RFC 5080).

Autre approche, mettre un bit "*Mandatory*" (aussi appelé "*Must Understand*") dans les messages. Si ce bit est à zéro, une vieille implémentation peut ignorer le message. Sinon, elle doit prévenir qu'elle ne sait pas le gérer (par exemple via un message `ICMP`). C'est ce que font L2TP (RFC 2661, section 4.1) ou SIP (RFC 3261, section 8.1.1.9). Cette méthode permet d'éviter qu'une extension cruciale soit ignorée par le pair. Mais elle diminue l'interopérabilité : si un programme récent l'utilise, il ne pourra pas parler avec les vieux qui ne savent pas du tout gérer cette extension.

On peut aussi négocier les extensions gérées au début de la session. Cela ralentit son établissement mais cela permet aux deux pairs qui dialoguent de savoir ce que fait l'autre. Typiquement, l'initiateur de la connexion indique ce qu'il sait gérer et le répondeur choisit parmi ces extensions acceptées. IKE (RFC 5996) fait cela, HTTP et SIP aussi, via leurs en-têtes `Accept*:` et `Supported:`.

Enfin, le concepteur d'un protocole de base, ou d'une extension à ce protocole doit lire la section 5 sur la sécurité. Elle rappelle qu'une extension à l'apparence inoffensive peut facilement introduire un nouveau risque de sécurité, ou parfois désactiver une mesure de sécurité qui était dans le protocole de base (l'ajout d'une poignée de mains dans un protocole qui était avant sans état peut créer une nouvelle occasion de DoS). L'analyse de sécurité de la nouvelle extension ne peut donc pas se contenter de reprendre celle du protocole de base, il faut s'assurer qu'il n'y ait pas de régression.

L'annexe A de notre RFC est très instructive car elle comprend trois études de cas de protocoles qui ont réussi, ou raté, leur extensibilité. La première concerne Radius (RFC 2865) grand succès mais qui, pour cette raison, a connu de fortes pressions pour son extension, pressions qui ont sérieusement secoué un protocole qui n'était pas vraiment conçu pour cela. L'idée dans le protocole de base était que les extensions se feraient en ajoutant des attributs (le protocole venait avec un certain nombre d'attributs standards dans les messages) dont la valeur pouvait avoir un nombre limité de types.

En pratique, on a constaté que, non seulement beaucoup de fournisseurs ne pouvaient pas s'empêcher de s'auto-allouer des noms d'attributs, mais aussi que bien des attributs ajoutés sortaient du modèle original (comme l'idée d'un dictionnaire, sur lesquels les correspondants se mettent d'accord et qui permet le déploiement de nouveaux attributs sans changer le code : idée astucieuse mais qui n'est pas dans le RFC 2865). La section 1 du RFC 2882, tirant un premier bilan, note qu'une des raisons de la tension est que le monde des NAS, pour lequel Radius avait été conçu, s'était beaucoup diversifié et complexifié, au delà du modèle initialement prévu. Le RFC 2882 notait aussi que certaines mises en œuvre ressemblaient à Radius (même format des messages) mais ne l'étaient pas vraiment (sémantique différente).

Un des cas compliqué est celui des types de données pour les attributs. Il était prévu d'ajouter de nouveaux attributs mais pas de nouveaux types. Il n'existe pas de registre des types définis, ni de moyen pour un client ou serveur Radius que savoir quels types gèrent son correspondant. L'utilisation d'attributs définis avec un type nouveau est donc problématique. Le RFC 6158 (section 2.1) a tenté de mettre de l'ordre dans ce zoo des types mais, publié plus de quatorze ans après la première norme Radius, il n'a pas eu un grand succès.

Radius dispose d'un mécanisme d'extensions spécifiques à un fournisseur (RFC 2865, section 6.2). Ce mécanisme a été utilisé à tort et à travers, notamment par des SDO différentes de l'IETF qui voulaient leurs propres extensions (alors que ces extensions n'avaient jamais été prévues pour être compatibles d'un fournisseur à l'autre). Autre problème, le cas des extensions « fournisseur » inconnues n'avait pas été traité. Or, certaines ont des conséquences pour la sécurité (définition d'une ACL par exemple) et il serait dommage qu'elles soient ignorées silencieusement. Le RFC 5080, section 2.5, estime que la seule solution est que les programmes mettant en œuvre Radius n'utilisent les extensions que lorsqu'ils savent (par exemple par configuration explicite) que l'autre programme les gère.

Autre protocole qui a connu bien des malheurs avec les extensions, TLS (annexe A.3). Son histoire remonte au protocole SSL, v2, puis v3, celle-ci remplacée par TLS. SSL, et TLS 1.0, n'avaient aucun mécanisme d'extension défini. Il a fallu attendre le RFC 4366 pour avoir un mécanisme normalisé permettant de nouveaux types d'enregistrements, de nouveaux algorithmes de chiffrement, de nouveaux messages lors de la poignée de mains initiale, etc. Il définit aussi ce que les mises en œuvre de TLS doivent faire des extensions inconnues (ignorer les nouveaux types, rejeter les nouveaux messages lors de la connexion). Mais, à ce moment, TLS était déjà très largement déployé et ces nouveaux mécanismes se heurtaient à la mauvaise volonté des vieilles implémentations. En pratique, les problèmes ont été nombreux, notamment pendant la négociation initiale. Ce n'est qu'en 2006 que les clients TLS ont pu considérer que SSLv2 était mort, et arrêter d'essayer d'interopérer avec lui. Encore aujourd'hui, de nombreux clients TLS restent délibérement à une version de TLS plus faible que ce qu'ils pourraient faire, pour éviter de planter lors de la connexion avec un vieux serveur (ces serveurs étant la majorité, y compris pour des sites Web très demandés). C'est ainsi que, lors de l'annonce de l'attaque BEAST <https://www.bortzmeyer.org/beast-tls.html> en 2011, on s'est aperçu que TLS 1.1 résolvait ce problème de sécurité depuis des années... mais n'était toujours pas utilisé, de peur de casser l'interopérabilité.

Dans le futur, cela pourra même poser des problèmes de sécurité sérieux : il n'existe aucun moyen pratique de retirer MD5 de la liste des algorithmes de hachage, par exemple. TLS 1.2 (RFC 5246) permet d'utiliser SHA-256 mais tenter d'ouvrir une connexion en TLS 1.2 avec un serveur a peu de chances de marcher, le mécanisme de repli des versions n'existant que sur le papier.

Morale de l'histoire, dit notre RFC : ce n'est pas tout de faire de bonnes spécifications, il faut aussi qu'elles soient programmées correctement.

Dernier cas intéressant étudié, L2TP (RFC 2661). Fournissant dès le début des types spécifiques à un fournisseur, disposant d'un bit "*Mandatory*" pour indiquer si les types doivent être compris par le récepteur, L2TP n'a guère eu de problèmes. On pourrait imaginer des ennuis (RFC 2661, section 4.2) si un programmeur mettait un type spécifique **et** le bit "*Mandatory*" (empêchant ainsi toute interaction avec un autre fournisseur qui, par définition, ne connaîtra pas ce type) mais cela ne s'est guère produit en pratique.