RFC 8095 : Services Provided by IETF Transport Protocols and Congestion Control Mechanisms

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 10 mars 2017

Date de publication du RFC : Mars 2017

https://www.bortzmeyer.org/8095.html

Les protocoles de **transport** (couche 4 dans le modèle en couches traditionnel), comme le fameux TCP, fournissent certains services aux applications situées au-dessus d'eux. Mais quels services exactement ? Qu'attend-on de la couche de transport ? Le but de ce RFC de synthèse est de lister tous les services possibles de la couche 4, et d'analyser ensuite tous les protocoles existants pour voir lesquels de ces services sont offerts. Ce document ne normalise donc pas un nouveau protocole, il classe et organise les protocoles existants. (L'idée est de pouvoir ensuite développer une interface abstraite permettant aux applications d'indiquer quels services elles attendent de la couche transport au lieu de devoir, comme c'est le cas actuellement, choisir un protocole donné. Une telle interface abstraite permettrait au système d'exploitation de choisir le protocole le plus adapté à chaque environnement.)

C'est d'autant plus important qu'il n'y a pas que TCP mais aussi des protocoles comme SCTP, UDP, DCCP, les moins connus FLUTE ou NORM, et même HTTP, qui est devenu une **couche de transport de fait**. Toute évolution ultérieure de l'architecture de l'Internet, des "*middleboxes*", des API offertes par le système d'exploitation, implique une compréhension détaillée de ce que fait exactement la couche transport.

Pour TCP, tout le monde connait (ou croit connaitre) : il fournit un service de transport de données fiable (les données qui n'arrivent pas sont retransmises automatiquement, l'application n'a pas à s'en soucier, la non-modification est - insuffisamment - contrôlée via une somme de contrôle), et ordonné (les octets arrivent dans l'ordre d'envoi même si, dans le réseau sous-jacent, un datagramme en a doublé un autre). TCP ne fournit pas par contre de service de confidentialité, ce qui facilite le travail de la NSA ou de la DGSI. Tout le monde sait également qu'UDP ne fournit aucun des deux services de fiabilité et d'ordre : si l'application en a besoin, elle doit le faire elle-même (et il est donc logique que la plupart des applications utilisent TCP).

Parfois, le service de transport offert aux applications est lui-même bâti sur un autre service de transport. C'est la raison pour laquelle ce RFC présente des protocoles qui ne sont pas « officiellement » dans

la couche 4 (mais, de toute façon, le modèle en couches n'a toujours été qu'une vague indication ; en faire une classification rigide n'a aucun intérêt, et a été une des raisons de l'échec du projet l'OSI). Un exemple est TLS. Une application qui s'en sert ne voit pas directement le TCP sous-jacent, elle confie ses données à TLS qui, à son tour, fait appel à TCP. Le service de transport vu par l'application offre ainsi les fonctions de TCP (remise fiable et ordonnée des données) plus celles de TLS (confidentialité, authentification et intégrité). Il faudrait être particulièrement pédant pour s'obstiner à classer TLS dans les applications comme on le voit parfois.

Le même phénomène se produit pour UDP : comme ce protocole n'offre quasiment aucun service par lui-même, on le complète souvent avec des services comme TFRC (RFC 5348 [1]) ou LEDBAT (RFC 6817) qui créent ainsi un nouveau protocole de transport au-dessus d'UDP.

La section 1 de notre RFC liste les services **possibles** d'une couche de transport :

— Envoi des messages à un destinataire ("*unicast*") ou à plusieurs ("*multicast*" ou "*anycast*"),
— Unidirectionnel (ce qui est toujours le cas avec le "*multicast*") ou bidirectionnel,
— Nécessite un établissement de la connexion avant d'envoyer des données, ou pas,
— Fiabilité de l'envoi (par un mécanisme d'accusé de réception et de réémission) ou bien "*fire and forget*" (notez que cette fiabilité peut être partielle, ce que permet par exemple SCTP),
— Intégrité des données (par exemple via une somme de contrôle),
— Ordre des données (avec certains protocoles de transport comme UDP, le maintien de l'ordre des octets n'est pas garanti, un paquet pouvant en doubler un autre),
— Structuration des données ("*framing*"), certains protocoles découpent en effet les données en messages successifs (ce que ne fait pas TCP),
— Gestion de la congestion,
— Confidentialité,
— Authentification (TLS fournit ces deux derniers services).

La section 3 du RFC est le gros morceau. Elle liste tous les protocoles de transport possibles (au moins ceux normalisés par l'IETF), en donnant à chaque fois une description générale du protocole, l'interface avec les applications, et enfin les services effectivement offerts par ce protocole.

À tout seigneur, tout honneur, commençons par l'archétype des protocoles de transport, TCP. Normalisé dans le RFC 793, très largement répandu (il est difficile d'imaginer une mise en œuvre d'IP qui ne soit pas accompagnée de TCP), utilisé quotidiennement par des milliards d'utilisateurs. Le RFC originel a connu pas mal de mises à jour et, aujourd'hui, apprendre TCP nécessite de lire beaucoup de RFC (le RFC 7414 en donne la liste). Ainsi, la notion de données urgentes, qui était dans le RFC originel, a été supprimée par le RFC 6093.

TCP multiplexe les connexions en utilisant les numéros de port, comme beaucoup de protocoles de transport. Une connexion est identifiée par un tuple {adresse IP source, port source, adresse IP destination, port destination}. Le port de destination identifie souvent le service utilisé (c'est moins vrai aujourd'hui, où la prolifération de "*middleboxes*" stupides oblige à tout faire passer sur les ports 80 et 443). TCP fournit un service de données non-structurées, un flot d'octets, mais, en interne, il découpe ces octets en **segments**, dont la taille est négociée au début (en général, TCP essaie de faire que cette taille soit la MTU du chemin, en utilisant les RFC 1191, RFC 1981 et de plus en plus le RFC 4821). Chaque octet envoyé a un numéro, le numéro de séquence, et c'est ainsi que TCP met en œuvre la fiabilité et l'ordre. (Contrairement à ce que croient certaines personnes, c'est bien l'octet qui a un numéro, pas le segment.) Autrefois, si deux segments non contigus étaient perdus, il fallait attendre la réémission du premier pour demander celle du second, mais les accusés de réception sélectifs du RFC 2018 ont changé cela.

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc5348.txt`

Quant au contrôle de congestion de TCP, il est décrit en détail dans le RFC 5681. TCP réagit à la perte de paquets (ou bien à leur marquage avec l'ECN du RFC 3168) en réduisant la quantité de données envoyées.

Les données envoyées par l'application ne sont pas forcément transmises immédiatement au réseau. TCP peut attendre un peu pour remplir davantage ses segments (RFC 896). Comme certaines applications (par exemple celles qui sont fortement interactives comme SSH) n'aiment pas les délais que cela entraine, ce mécanisme est typiquement débrayable.

Enfin, pour préserver l'intégrité des données envoyées, TCP utilise une somme de contrôle (RFC 793, section 3.1, et RFC 1071). Elle ne protège pas contre toutes les modifications possibles et il est recommandé aux applications d'ajouter leur propre contrôle d'intégrité (par exemple, si on transfère un fichier, via un condensat du fichier).

Et l'interface avec les applications, cruciale, puisque le rôle de la couche transport est justement d'offrir des services aux applications ? Celle de TCP est décrite de manière relativement abstraite dans le RFC 793 (six commandes, "*Open*", "*Close*", "*Send*", "*Receive*", etc). Des points comme les options TCP n'y sont pas spécifiés. Le RFC 1122 est un peu plus détaillé, mentionnant par exemple l'accès aux messages ICMP qui peuvent indiquer une erreur TCP. Enfin, une interface concrète est celle des prises, normalisées par POSIX (pas de RFC à ce sujet). Vous créez une prise avec l'option `SOCK_STREAM` et hop, vous utilisez TCP et tous ses services.

Quels services, justement ? TCP fournit :
— Établissement d'une connexion, et démultiplexage en utilisant les numéros de port,
— Transport "*unicast*" (l'"*anycast*" est possible, si on accepte le risque qu'un changement de routes casse subitement une connexion),
— Communication dans les deux sens,
— Données envoyées sous forme d'un flot d'octets, sans séparation (pas de notion de message, c'est à l'application de le faire, si elle le souhaite, par exemple en indiquant la taille du message avant le message, comme le font EPP et DNS), c'est aussi cela qui permet l'accumulation de données avant envoi (algorithme de Nagle),
— Transport fiable, les données arriveront toutes, et dans l'ordre,
— Détection d'erreurs (mais pas très robuste),
— Contrôle de la congestion, via les changements de taille de la fenêtre d'envoi (la fenêtre est l'ensemble des octets qui peuvent être envoyés avant qu'on ait reçu l'accusé de réception des données en cours), voir le RFC 5681.

Par contre, TCP ne fournit **pas** de confidentialité, et l'authentification se limite à une protection de l'adresse IP <https://www.bortzmeyer.org/usurpation-adresse-ip.html> contre les attaquants situés hors du chemin (RFC 5961).

Après TCP, regardons le deuxième protocole de transport étudié, MPTCP ("*Multipath TCP*", RFC 6824). C'est une extension de TCP qui permet d'exploiter le "*multi-homing*". Pour échapper aux "*middleboxes*" intrusives, MPTCP fonctionne en créant plusieurs connexions TCP ordinaires depuis/vers toutes les adresses IP utilisées, et en multiplexant les données sur ces connexions (cela peut augmenter le débit, et cela augmente la résistance aux pannes, mais cela peut aussi poser des problèmes si les différents chemins ont des caractéristiques très différentes). La signalisation se fait par des options TCP.

L'interface de base est la même que celle de TCP, mais il existe des extensions (RFC 6897) pour tirer profit des particularités de MPTCP.

Les services sont les mêmes que ceux de TCP avec, en prime le "*multi-homing*" (il peut même y avoir des adresses IPv4 **et** IPv6 dans la même session MPTCP), et ses avantages notamment de résilience.

Après TCP, UDP est certainement le protocole de transport le plus connu. Il est notamment très utilisé par le DNS. Le RFC 8085 explique comment les applications peuvent l'utiliser au mieux. La section 3.3 de notre RFC lui est consacrée, pour décrire son interface et ses services.

Contrairement à TCP, UDP n'a pas la notion de connexion (on envoie directement les données, sans négociation préalable), UDP découpe les données en messages (voilà pourquoi les messages DNS en UDP ne sont pas précédés d'une longueur : UDP lui-même fait le découpage), n'a pas de contrôle de congestion, et ne garantit pas le bon acheminement. UDP dispose d'un contrôle d'intégrité, mais il est facultatif (quoique **très** recommandé) en IPv4, où on peut se contenter du contrôle d'intégrité d'IP. IPv6 n'ayant pas ce contrôle, UDP sur IPv6 doit activer son propre contrôle, sauf dans certains cas très précis (RFC 6936).

En l'absence de contrôle de congestion, l'application doit être prudente, veiller à ne pas surcharger le réseau, et ne pas s'étonner si l'émetteur envoie plus que ce que le récepteur peut traiter. D'une façon générale, il faut penser à lire le RFC 8085, qui explique en détail tout ce qu'une application doit faire si elle tourne sur UDP.

Il est d'ailleurs recommandé de bien se poser la question de l'utilité d'UDP, dans beaucoup de cas. Un certain nombre de développeurs se disent au début d'un projet « j'ai besoin de vitesse [sans qu'ils fassent bien la différence entre latence `<https://www.bortzmeyer.org/latence.html>` et capacité `<https://www.bortzmeyer.org/capacite.html>`], je vais utiliser UDP ». Puis ils découvrent qu'ils ont besoin de contrôle de flux, d'ordre des données, de bonne réception des données, ils ajoutent à chaque fois des mécanismes ad hoc, spécifiques à leur application et, au bout du compte, ils ont souvent réinventé un truc aussi lourd que TCP, mais bien plus bogué. Attention donc à ne pas réinventer la roue pour rien.

L'interface d'UDP, maintenant. Le RFC 768 donne quelques indications de base, que le RFC 8085 complète. Bien qu'UDP n'ait pas le concept de connexion, il est fréquent que les API aient une opération `connect()` ou analogue. Mais il ne faut pas la confondre avec l'opération du même nom sur TCP : ce `connect()` UDP est purement local, associant la structure de données locale à une machine distante (c'est ainsi que cela se passe avec les prises Berkeley).

Et les services d'UDP ? La liste est évidemment bien plus courte que pour TCP. Elle comprend :
— Transport des données, "*unicast*", "*multicast*" , "*anycast*" et "*broadcast*" (c'est le seul point où UDP en fournit davantage que TCP),
— Démultiplexage en utilisant les numéros de port,
— Unidirectionnel (ce qui est toujours le cas avec le "*multicast*") ou bidirectionnel,
— Données structurées en messages,
— Aucune garantie, ou signalement, des pertes de message,
— Aucune garantie sur l'ordre de délivrance des messages.

Nettement moins connu qu'UDP est UDP-Lite, normalisé dans le RFC 3828. C'est une version très légèrement modifiée d'UDP, où la seule différence est que les données corrompues (détectées par la somme de contrôle) sont quand même données à l'application réceptrice, au lieu d'être jetées comme avec UDP. Cela peut être utile pour certains applications, notamment dans les domaines audio et vidéo.

Avec UDP-Lite, le champ Longueur de l'en-tête UDP change de sémantique : il n'indique plus la longueur totale des données mais la longueur de la partie qui est effectivement couverte par la somme de contrôle. Typiquement, on ne couvre que l'en-tête applicatif. Le reste est... laissé à la bienveillance des dieux (ou des démons). Pour tout le reste, voyez la section sur UDP.

Notez qu'il n'existe pas d'API spécifique pour UDP-Lite. Si quelqu'un parmi mes lecteurs a des exemples de code bien clairs...

Bien plus original est SCTP (RFC 4960). C'est un protocole à connexion et garantie d'acheminement et d'ordre des données, comme TCP. Mais il s'en distingue par sa gestion du "*multi-homing*". Avec SCTP, une connexion peut utiliser plusieurs adresses IP source et destination, et passer de l'une à l'autre pendant la session, assurant ainsi une bonne résistance aux pannes. Plus drôle, cet ensemble d'adresses peut mêler des adresses IPv4 et IPv6.

Notez aussi qu'une connexion SCTP (on dit une association) comporte plusieurs flux de données, afin de minimiser le problème connu sous le nom de "*head of line blocking*" (un paquet perdu empêche la délivrance de toutes les données qui suivent tant qu'il n'a pas été réémis).

SCTP avait surtout été conçu pour la signalisation dans les réseaux téléphoniques. Mais on le trouve dans d'autres cas, comme ForCES (cf. RFC 5811) ou comme la signalisation WebRTC (RFC 8825).

Contrairement à TCP, SCTP utilise une quadruple poignée de mains pour établir la connexion, ce qui permet de ne négocier les options qu'une fois certain de l'identité du partenaire (les techniques anti-DoS de TCP sont incompatible avec l'utilisation des options, cf. RFC 4987, section 3.6). La somme de contrôle fait 32 bits (au lieu des 16 bits de TCP et UDP) et est donc normalement plus robuste.

SCTP est très extensible et plusieurs extensions ont déjà été définies comme l'ajout ou le retrait d'adresses IP pendant l'association (RFC 5061), ou bien la possibilité de n'accepter qu'une fiabilité partielle (RFC 3758). Pour la sécurité, on peut faire tourner TLS sur SCTP (RFC 3436) au prix de la perte de quelques fonctions, ou bien utiliser DTLS (RFC 6083), qui préserve quasiment toutes les fonctions de SCTP.

Victime fréquente des "*middleboxes*" stupides qui ne connaissent qu'UDP et TCP, SCTP peut tourner sur UDP (RFC 6951), au lieu de directement reposer sur IP, afin de réussir à passer ces "*middleboxes*".

Contrairement à des protocoles de transport plus anciens, SCTP a une interface bien spécifiée. Le RFC 4960 définit l'interface abstraite, et une extension aux prises Berkeley, spécifiée dans le RFC 6458, lui donne une forme concrète. Cette API prévoit également certaines extensions, comme celle des reconfigurations dynamiques d'adresses du RFC 5061.

Les services fournis par SCTP sont très proches de ceux fournis par TCP, avec deux ajouts (la gestion du "*multi-homing*" et le multi-flux), et un changement (données structurées en messages, au lieu d'être un flot d'octets continu comme TCP).

Un autre protocole de transport peu connu, et ne fournissant pas, lui, de fiabilité de l'envoi des données, est DCCP (RFC 4340). DCCP est une sorte d'UDP amélioré, qui peut fournir des services supplémentaires à ceux d'UDP, tout en restant plus léger que TCP (la description du besoin figure dans le RFC 4336). DCCP est bien adapté aux applications multimédia ou aux jeux en ligne, où une faible latence <https://www.bortzmeyer.org/latence.html> est cruciale, mais où peut aimer avoir des services en plus. Sans DCCP, chaque application qui veut de l'« UDP amélioré » devrait tout réinventer (et ferait sans doute des erreurs).

DCCP a des connexions, comme TCP, qu'on établit avant de communiquer et qu'on ferme à la fin. Il offre une grande souplesse dans le choix des services fournis, choix qui peuvent être unilatéraux (seulement l'envoyeur, ou bien seulement le récepteur) ou négociés lors de l'ouverture de la connexion. Le paquet d'ouverture de connexion indique l'application souhaitée (RFC 5595), ce qui peut être une information utile aux équipements intermédiaires. S'il faut faire passer DCCP à travers des "*middleboxes*" ignorantes, qui n'acceptent qu'UDP et TCP, on peut, comme avec SCTP, encapsuler dans UDP (RFC 6773).

L'interface avec DCCP permet d'ouvrir, de fermer et de gérer une connexion. Il n'y a pas d'API standard. Les services fournis sont :

— Transport des données, uniquement "*unicast*",
— Protocole à connexion, et démultiplexage fondé sur les numéros de port,
— Structuration des données en messages,
— Les messages peuvent être perdus (mais, contrairement à UDP, l'application est informée des pertes), et ils peuvent être transmis dans le désordre,
— Contrôle de la congestion (le gros avantage par rapport à UDP), et avec certains choix (optimiser la latence ou au contraire la gigue, par exemple) laissés à l'application.

Autre exemple de protocole de transport, même s'ils ne sont en général pas décrits comme tels, TLS (RFC 5246) et son copain DTLS (RFC 6347). Si on est un fanatique du modèle en couches, on ne met pas ces protocoles de sécurité en couche 4 mais, selon l'humeur, en couche 5 ou en couche 6. Mais si on est moins fanatique, on reconnait que, du point de vue de l'application, ce sont bien des protocoles de transport : c'est à eux que l'application confie ses données, comptant sur les services qu'ils promettent.

TLS tourne sur TCP et DTLS sur UDP. Du point de vue de l'application, TLS fournit les services de base de TCP (transport fiable d'un flot d'octets) et DTLS ceux d'UDP (envoi de messages qui arriveront peut-être). Mais ils ajoutent à ces services de base leurs services de sécurité :

— Confidentialité, éventuellement avec sécurité future,
— Authentification,
— Intégrité (souvent citée à part mais, à mon avis, c'est un composant indispensable de l'authentification).

Le RFC rappelle qu'il est important de se souvenir que TLS ne spécifie pas un mécanisme d'authentification unique, ni même qu'il doit y avoir authentification. On peut n'authentifier que le serveur (c'est actuellement l'usage le plus courant), le client et le serveur, ou bien aucun des deux. La méthode la plus courante pour authentifier est le certificat PKIX (X.509), appelé parfois par une double erreur « certificat SSL ».

DTLS ajoute également au service de base quelques trucs qui n'existent pas dans UDP, comme une aide pour la recherche de PMTU ou un mécanisme de "*cookie*" contre certaines attaques.

Il n'y a pas d'API standard de TLS. Si on a écrit une application avec l'API d'OpenSSL, il faudra refaire les appels TLS si on passe à WolfSSL ou GnuTLS. C'est d'autant plus embêtant que les programmeurs d'application ne sont pas forcément des experts en cryptographie et qu'une API mal conçue peut les entrainer dans des erreurs qui auront des conséquences pour la sécurité (l'article « "*The most dangerous code in the world : validating SSL certificates in non-browser software*" `<http://doi.acm.org/10.1145/2382196.2382204>` » en donne plusieurs exemples).

Passons maintenant à RTP (RFC 3550). Ce protocole est surtout utilisé pour les applications multimédia, où on accepte certaines pertes de paquet, et où le format permet de récupérer après cette perte. Comme TLS, RTP fonctionne au-dessus du « vrai » protocole de transport, et peut exploiter ses services (comme la protection de l'intégrité d'une partie du contenu, que fournissent DCCP et UDP-Lite).

RTP comprend en fait deux protocoles, RTP lui-même pour les données et RTCP pour le contrôle. Par exemple, c'est via RTCP qu'un émetteur apprend que le récepteur ne reçoit pas vite et donc qu'il faudrait, par exemple, diminuer la qualité de la vidéo.

RTP n'a pas d'interface standardisée offerte aux programmeurs. Il faut dire que RTP est souvent mis en œuvre, non pas dans un noyau mais directement dans l'application (comme avec libortp `<http://www.linphone.org/>` sur Unix). Ces mises en œuvre sont donc en général optimisées pour une utilisation particulière, au lieu d'être généralistes comme c'est le cas avec les implémentations de TCP ou UDP.

Autre cas d'un protocole de transport qui fonctionne au-dessus d'un autre protocole de transport, HTTP (RFC 7230 et suivants <https://www.bortzmeyer.org/http-11-reecrit.html>). Il n'était normalement pas conçu pour cela mais, dans l'Internet d'aujourd'hui, où il est rare d'avoir un accès neutre <https://www.bortzmeyer.org/neutralite.html>, où les ports autres que 80 et 443 sont souvent bloqués, et où hôtels, aéroports et écoles prétendent fournir un « accès Internet » qui n'est en fait qu'un accès HTTP, bien des applications qui n'ont rien à voir avec le Web en viennent à utiliser HTTP comme protocole de transport. (Même si le RFC 3205 n'encourage pas vraiment cette pratique puisque HTTP peut ne pas être adapté à tout. Mais, souvent, on n'a pas le choix.)

Outre cette nécessité de contourner blocages et limitations, l'utilisation de HTTP comme transport a quelques avantages : protocole bien connu, disposant d'un grand nombre de mises en œuvre, que ce soit pour les clients ou pour les serveurs, et des mécanismes de sécurité existants (RFC 2617, RFC 2817…). L'un des grands succès de HTTP est le style REST : de nombreuses applications sont conçues selon ce style <https://www.bortzmeyer.org/programmation-rest.html>.

Les applications qui utilisent HTTP peuvent se servir des méthodes existantes (`GET`, `PUT`, etc) ou bien en créer de nouvelles (qui risquent de moins bien passer partout).

Je ne vais pas refaire ici la description de HTTP que contient le RFC (suivant le même plan que pour les autres protocoles de transport), je suppose que vous connaissez déjà HTTP. Notez quand même quelques points parfois oubliés : HTTP a un mécanisme de négociation du contenu, qui permet, par exemple, de choisir le format lorsque la ressource existe en plusieurs formats, HTTP a des connexions persistentes donc on n'est pas obligé de se taper un établissement de connexion TCP par requête, et HTTP a des mécanismes de sécurité bien établis, à commencer par HTTPS.

Il y a plein de bibliothèques qui permettent de faire de l'HTTP facilement (libcurl et neon en C, Requests en Python, etc). Chacune a une API différente. Le W3C a normalisé une API nommée XMLHttpRequest <http://www.w3.org/TR/XMLHttpRequest>, très utilisée par les programmeurs JavaScript.

Les services que fournit HTTP quand on l'utilise comme protocole de transport sont :
— Transport "*unicast*", bi-directionnel, fiable (grâce à TCP en dessous), et avec contrôle de congestion (idem),
— Négociation du format, possibilité de ne transférer qu'une partie d'une ressource,
— Authentification et confidentialité si on utilise HTTPS.

Beaucoup moins connus que les protocoles précédents sont deux des derniers de notre liste, FLUTE et NORM.

FLUTE ("*File Delivery over Unidirectional Transport/ Asynchronous Layered Coding Reliable Multicast*") est normalisé dans le RFC 6726. Il est conçu pour un usage très spécifique, la distribution de fichiers à des groupes "*multicast*" de grande taille, où on ne peut pas demander à chaque récepteur d'accuser réception. Il est surtout utilisé dans le monde de la téléphonie mobile (par exemple dans la spécification 3GPP TS 26.346 <http://www.3gpp.org/DynaReport/26346.htm>).

FLUTE fonctionne sur UDP, et le protocole ALC du RFC 5775. Il est souvent utilisé sur des réseaux avec une capacité garantie, et où on peut donc relativiser les problèmes de congestion. Il n'y a pas d'interface de programmation spécifiée.

Les services de FLUTE sont donc :
— Transport de fichiers (que FLUTE appelle « objets ») plutôt que d'octets,
— Fiable (heureusement, pour des fichiers).

Et NORM ("*NACK-Oriented Reliable Multicast*" ? Normalisé dans le RFC 5740, il rend à peu près les mêmes services que FLUTE (distribution massive de fichiers). À noter qu'il en existe une mise en œuvre en logiciel libre <https://www.nrl.navy.mil/itd/ncs/products/norm>.

Reste un cas amusant, ICMP. Bien sûr, ICMP n'est pas du tout conçu pour être un protocole de transport, c'est le protocole de signalisation d'IP (RFC 792 pour ICMP sur IPv4 et RFC 4443 pour ICMP sur IPv6). Mais, bon, comme il est situé au-dessus de la couche 3, on peut le voir comme un protocole de transport.

Donc, ICMP est sans connexion, sans fiabilité, et unidirectionnel. Évidemment pas de contrôle de congestion. Pas vraiment d'interface standard, les messages ICMP ne sont signalés qu'indirectement aux applications (dans certains cas, une application peut demander à recevoir les messages ICMP). On ne peut pas tellement s'en servir comme protocole de transport, bien que des programmes comme ptunnel <http://www.cs.uit.no/~daniels/PingTunnel/> s'en servent presque ainsi.

Après cette longue section 3 qui faisait le tour de tous les protocoles de transport ou assimilés, la section 4 de notre RFC revient sur la question cruciale de la congestion. Sans contrôle de congestion, si chacun émettait comme ça lui chante, l'Internet s'écroulerait vite sous la charge. C'est donc une des tâches essentielles d'un protocole de transport que de fournir ce contrôle de congestion. Pour ceux qui ne le font pas, l'application doit le faire (et c'est très difficile à faire correctement).

À noter que la plupart des protocoles de transport tendent à ce que chaque flot de données utilise autant de capacité disponible que les autres flots. Au contraire, il existe des protocoles « décroissants » comme LEDBAT (RFC 6817) qui cèdent la place aux autres et n'utilise la capacité que lorsque personne n'est en concurrence avec eux.

La section 5 de notre RFC revient sur la notion de fonctions fournies par le protocole de transport, et classe sur un autre axe que la section 3. La section 3 était organisée par protocole et, pour chaque protocole, indiquait quelles étaient ses fonctions. La section 5, au contraire, est organisée par fonction et indique, pour chaque fonction, les valeurs qu'elle peut prendre, et les protocoles qui correspondent. Première catégorie de fonctions, celle du contrôle. Ainsi, une des fonctions de base d'un protocole de transport est l'adressage, celui-ci peut être "*unicast*" (TCP, UDP, SCTP, TLS, HTTP), "*multicast*" (UDP encore, FLUTE, NORM), "*broadcast*" (UDP toujours), "*anycast*" (UDP, quoique TCP puisse l'utiliser si on accepte le risque de connexions coupées lorsque le routage change).

Autre fonction, la façon dont se fait l'association entre les deux machines, et elle peut être avec connexion (TCP, SCTP, TLS) ou sans connexion (UDP). La gestion du "*multi-homing*" peut être présente (MPTCP, SCTP) ou pas. La signalisation peut être faite avec ICMP ou bien dans le protocole d'application (RTP).

Seconde catégorie de fonctions, la délivrance de données. Première fonction dans cette catégorie, la fiabilité, qui peut être complète (TCP, SCTP, TLS), partielle (RTP, FLUTE, NORM) ou inexistante (UDP, DCCP). Deuxième fonction, la détection d'erreurs, par une somme de contrôle qui couvre toutes les données (TCP, UDP, SCTP, TLS), une partie (UDP-Lite), et qui peut même être optionnelle (UDP en IPv4). Troisième fonction de délivrance, l'ordre des données, qui peut être maintenu (TCP, SCTP, TLS, HTTP, RTP) ou pas (UDP, DCCP, DTLS). Quatrième fonction, le découpage des données : flot sans découpage (TCP, TLS) ou découpage en messages (UDP, DTLS).

Troisième catégorie de fonctions, celles liées au contrôle de la transmission et notamment de la lutte contre la congestion.

Enfin, quatrième et dernière catégorie de fonctions, celles liées à la sécurité : authentification (TLS, DTLS) et confidentialité (les mêmes) notamment.

Voilà, armé de ce RFC, si vous êtes développeurs d'un nouveau protocole applicatif sur Internet, vous pouvez choisir votre protocole de transport sans vous tromper.

Une lecture recommandée est « *"De-Ossifying the Internet Transport Layer : A Survey and Future Perspectives"* <https://ieeexplore.ieee.org/abstract/document/7738442/> », de Giorgos Papastergiou, Gorry Fairhurst, David Ros et Anna Brunstr[Caractère Unicode non montré [2]]m.

2. Car trop difficile à faire afficher par LaTeX