RFC 8303 : On the Usage of Transport Features Provided by IETF Transport Protocols

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 8 février 2018

Date de publication du RFC : Février 2018

https://www.bortzmeyer.org/8303.html

La famille de protocoles TCP/IP dispose d'un grand nombre de protocoles de transport. On connait bien sûr TCP mais il y en a d'autres comme UDP et SCTP et d'autres encore moins connus. Cette diversité peut compliquer la tâche des développeurs d'application, qui ne savent pas toujours bien lequel choisir. Et elle pose également problème pour les développements futurs à l'IETF, par exemple la réflexion en cours sur les API. On a besoin d'une description claire des services que chaque protocole fournit effectivement. C'est le but du groupe de travail TAPS <https://datatracker.ietf.org/wg/taps/about/> dont voici le deuxième RFC : une description de haut niveau des services que fournit la couche transport. À faire lire à tous les développeurs d'applications réseau, à tous les participants à la normalisation, et aux étudiants en réseaux informatiques.

Ce RFC parle de TCP, MPTCP, SCTP, UDP et UDP-Lite, sous l'angle « quels services rendent-ils aux applications ? » (QUIC <https://www.bortzmeyer.org/quic.html>, pas encore normalisé à l'époque, n'y figure pas. LEDBAT, qui n'est pas vraiment un protocole de transport, est par contre présent.) L'idée est de lister un ensemble d'opérations abstraites, qui pourront ensuite être exportées dans une API. J'en profite pour recommander la lecture du RFC 8095 [1], premier RFC du groupe TAPS.

La section 1 de notre RFC décrit la terminologie employée. Il faut notamment distinguer trois termes :
— La **fonction** ("*Transport Feature*") qui désigne une fonction particulière que le protocole de transport va effectuer, par exemple la confidentialité, la fiabilité de la distribution des données, le découpage des données en messages, etc.
— Le **service** ("*Transport Service*") qui est un ensemble cohérent de fonctions. C'est ce que demande l'application.
— Le **protocole** ("*Transport Protocol*") qui est une réalisation concrète d'un ou plusieurs services.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc8095.txt

Un exemple de programme qui permet d'essayer différents types de prises ("*sockets*") est (en ligne sur `https://www.bortzmeyer.org/files/socket-types.c`). Vous pouvez le compiler et le lancer en indiquant le type de service souhaité. `STREAM` va donner du TCP, `SEQPACKET` du SCTP, `DATAGRAM` de l'UDP :

```
% make socket-types
cc -Wall -Wextra     socket-types.c   -o socket-types

% ./socket-types -v -p 853 9.9.9.9 STREAM
9.9.9.9 is an IP address
"Connection" STREAM successful to 9.9.9.9

% ./socket-types -v -p 443 -h dns.bortzmeyer.org STREAM
Address(es) of dns.bortzmeyer.org is(are): 2605:4500:2:245b::42 204.62.14.153
"Connection" STREAM successful to 2605:4500:2:245b::42
256 response bytes read
```

À première vue, l'API sockets nous permet de sélectionner le service, indépendamment du protocole qui le fournira. Mais ce n'est pas vraiment le cas. Si on veut juste un flot de données, cela pourrait être fait avec TCP ou SCTP, mais seul TCP sera sélectionné si on demande `STREAM`. En fait, les noms de services dns l'appel à `socket()` sont toujours en correspondance univoque avec les protocoles de transport.

Autres termes importants :

— Adresse ("*Transport Address*") qui est un tuple regroupant adresse IP, nom de protocole et numéro de port. Le projet TAPS considère uniquement des protocoles ayant la notion de ports, et fournissant donc la fonction de démultiplexage (différencier les données destinées à telle ou telle application).
— Primitive ("*Primitive*"), un moyen pour l'application de demander quelque chose au protocole de transport.
— Événement ("*Event*"), généré par le protocole de transport, et transmis à l'application pour l'informer.

La section 2 décrit le problème général. Actuellement, les applications sont liées à un protocole, même quand elles ne le souhaitent pas. Si une application veut que ses octets soient distribués dans l'ordre, et sans pertes, plusieurs protocoles peuvent convenir, et il n'est pas forcément utile de se limiter à seulement TCP, par exemple. Pour avancer vers un moindre couplage entre application et protocole, ce RFC décrit les fonctions et les services, un futur RFC définira des services minimaux, un autre peut-être une API.

Le RFC essaie de s'en tenir au minimum, des options utiles mais pas indispensables (comme le fait pour l'application de demander à être prévenue de certains changements réseau, cf. RFC 6458) ne sont pas étudiées.

Le RFC procède en trois passes : la première (section 3 du RFC) analyse et explique ce que chaque protocole permet, la seconde (section 4) liste les primitives et événements possibles, la troisième (section 5) donne pour chaque fonction les protocoles qui la mettent en œuvre. Par exemple, la passe 1 note que TCP fournit à l'application un moyen de fermer une connexion en cours, la passe 2 liste une primitive `CLOSE.TCP`, la passe 3 note que TCP et SCTP (mais pas UDP) fournissent cette opération de fermeture (qui fait partie des fonctions de base d'un protocole orienté connexion, comme TCP). Bon, d'accord, cette méthode est un peu longue et elle rend le RFC pas amusant à lire mais cela permet de s'assurer que rien n'a été oublié.

Donc, section 3 du RFC, la passe 1. Si vous connaissez bien tous ces protocoles, vous pouvez sauter cette section. Commençons par le protocole le plus connu, TCP (RFC 793). Tout le monde sait que c'est

un protocole à connexion, fournissant un transport fiable et ordonné des données. Notez que la section 3.8 de la norme TCP décrit, sinon une API, du moins l'interface (les primitives) offertes aux applications. Vous verrez que les noms ne correspondent pas du tout à celle de l'API sockets (`OPEN` au lieu de `connect/listen`, `SEND` au lieu de `write`, etc).

TCP permet donc à l'application d'ouvrir une connexion (activement - TCP se connecte à un TCP distant - ou passivement - TCP attend les connexions), d'envoyer des données, d'en recevoir, de fermer la connexion (gentiment - TCP attend encore les données de l'autre - ou méchamment - on part en claquant la porte). Il y a aussi des événements, par exemple la fermeture d'une connexion déclenchée par le pair, ou bien une expiration d'un délai de garde. TCP permet également de spécifier la qualité de service attendue (cf. RFC 2474, et la section 4.2.4.2 du RFC 1123). Comme envoyer un paquet IP par caractère tapé serait peu efficace, TCP permet, pour les sessions interactives, genre SSH, d'attendre un peu pour voir si d'autres caractères n'arrivent pas (algorithme de Nagle, section 4.2.3.4 du RFC 1123). Et il y a encore d'autres possibilités moins connues, comme le "*User Timeout*" du RFC 5482. Je vous passe des détails, n'hésitez pas à lire le RFC pour approfondir, TCP a d'autres possibilités à offrir aux applications.

Après TCP, notre RFC se penche sur MPTCP. MPTCP crée plusieurs connexions TCP vers le pair, utilisant des adresses IP différentes, et répartit le trafic sur ces différents chemins (cf. RFC 6182). Le modèle qu'il présente aux applications est très proche de celui de TCP. MPTCP est décrit dans les RFC 6824 et son API est dans le RFC 6897. Ses primitives sont logiquement celles de TCP, plus quelques extensions. Parmi les variations par rapport à TCP, la primitive d'ouverture de connexion a évidemment une option pour activer ou pas MPTCP (sinon, on fait du TCP normal). Les extensions permettent de mieux contrôler quelles adresses IP on va utiliser pour la machine locale.

SCTP est le principal concurrent de TCP sur le créneau « protocole de transfert avec fiabilité de la distribution des données ». Il est normalisé dans le RFC 9260. Du point de vue de l'application, SCTP a trois différences importantes par rapport à TCP :

— Au lieu d'un flot continu d'octets, SCTP présent une suite de messages, avec des délimitations entre eux,
— Il a un mode où la distribution fiable n'est pas garantie,
— Et il permet de tirer profit du "*multi-homing*".

Il a aussi une terminologie différente, par exemple on parle d'associations et plus de connexions.

Certaines primitives sont donc différentes : par exemple, `Associate` au lieu du `Open` TCP. D'autres sont identiques, comme `Send` et `Receive`. SCTP fournit beaucoup de primitives, bien plus que TCP.

Et UDP ? Ce protocole sans connexion, et sans fiabilité, ainsi que sa variante UDP-Lite, est analysé dans un RFC séparé, le RFC 8304.

Nous passons ensuite à LEDBAT (RFC 6817). Ce n'est pas vraiment un protocole de transport, mais plutôt un système de contrôle de la congestion bâti sur l'idée « décroissante » de « on n'utilise le réseau que si personne d'autre ne s'en sert ». LEDBAT est pour les protocoles économes, qui veulent transférer des grandes quantités de données sans gêner personne. Un flot LEDBAT cédera donc toujours la place à un flot TCP ou compatible (RFC 5681). N'étant pas un protocole de transport, LEDBAT peut, en théorie, être utilisé sur les protocoles existants comme TCP ou SCTP (mais les extensions nécessaires n'ont hélas jamais été spécifiées). Pour l'instant, les applications n'ont pas vraiment de moyen propre d'activer ou de désactiver LEDBAT.

Bien, maintenant, passons à la deuxième passe (section 4 du RFC). Après la description des protocoles lors de la première passe, cette deuxième passe liste explicitement les primitives, rapidement

décrites en première passe, et les met dans différentes catégories (alors que la première passe classait par protocole). Notons que certaines primitives, ou certaines options sont exclues. Ainsi, TCP a un mécanisme pour les données urgentes (RFC 793, section 3.7). Il était utilisé pour telnet (cf. RFC 854) mais n'a jamais vraiment fonctionné de manière satisfaisante et le RFC 6093 l'a officiellement exclu pour les nouvelles applications de TCP. Ce paramètre n'apparait donc pas dans notre RFC, puisque celui-ci vise le futur.

La première catégorie de primitives concerne la gestion de connexion. On y trouve d'abord les primitives d'établissement de connexion :

— `CONNECT.TCP` : établit une connexion TCP,
— `CONNECT.SCTP` : idem pour SCTP,
— `CONNECT.MPTCP` : établit une connexion MPTCP,
— `CONNECT.UDP` : établit une « connexion » UDP sauf, que, UDP étant sans connexion, il s'agit cette fois d'une opération purement locale, sans communication avec une autre machine.

Cette catégorie comprend aussi les primitives permettant de signaler qu'on est prêt à recevoir une connexion, `LISTEN.TCP`, celles de maintenance de la connexion, qui sont bien plus variées, et celles permettant de mettre fin à la connexion (comme `CLOSE.TCP` et `CLOSE.SCTP`, ou comme le signal qu'un délai de garde a été dépassé, `TIMEOUT.TCP`).

Dans les primitives de maintenance, on trouve aussi bien les changements du délai de garde (`CHANGE_TIMEOUT.TCP` et `SCTP`), le débrayage de l'algorithme de Nagle, pour TCP et SCTP, l'ajout d'une adresse IP locale à l'ensemble des adresses utilisables (`ADD_PATH.MPTCP` et `ADD_PATH.SCTP`), la notification d'erreurs, l'activation ou la désactivation de la somme de contrôle (UDP seulement, la somme de contrôle étant obligatoire pour les autres)...La plupart sont spécifiques à SCTP, le protocole le plus riche.

La deuxième catégorie rassemble les primitives liées à la transmission de données. On y trouve donc `SEND.TCP` (idem pour SCTP et UDP), `RECEIVE.TCP` (avec ses équivalents SCTP et UDP), mais aussi les signalements d'échec comme `SEND_FAILURE.UDP` par exemple quand on a reçu un message ICMP d'erreur.

Venons-en maintenant à la troisième passe (section 5 du RFC). On va cette fois lister les primitives sous forme d'un concept de haut niveau, en notant les protocoles auxquels elles s'appliquent. Ainsi, pour la gestion des connexions, on aura de quoi se connecter : une primitive "*Connect*", qui s'applique à plusieurs protocoles (TCP, SCTP et UDP, même si "*Connect*" ne se réalise pas de la même façon pour tous ces protocoles). Et de quoi écouter passivement : une primitive "*Listen*". Pour la maintenance, on aura une primitive "*Change timeout*", qui s'applique à TCP et SCTP, une "*Disable Nagle*" qui s'applique à TCP et SCTP, une "*Add path*" qui a un sens pour MPTCP et SCTP, une "*Disable checksum*" qui ne marche que pour UDP, etc.

Pour l'envoi de données, on ne fusionne pas les opérations d'envoi (`SEND.PROTOCOL`) de la passe précédente car elles ont une sémantique trop différente. Il y a "*Reliably transfer data*" pour TCP, et "*Reliably transfer a message*" pour SCTP. (Et bien sûr "*Unreliably transfer a message*" avec UDP.)

Voilà, tout ça a l'air un peu abstrait mais ça acquerra davantage de sens quand on passera aux futures descriptions d'API (voir le RFC 8923).

L'annexe B du RFC explique la méthodologie suivie pour l'élaboration de ce document : coller au texte des RFC, exclure les primitives optionnelles, puis les trois passes dont j'ai parlé plus tôt. C'est cette annexe, écrite avant le RFC, qui avait servi de guide aux auteurs des différentes parties (il fallait des spécialistes SCTP pour ne pas faire d'erreurs lors de la description de ce protocole complexe...)

Notez (surtout si vous avez lu le RFC 8095) que TLS n'a pas été inclus, pas parce que ce ne serait pas un vrai protocole de transport (ça se discute) mais car il aurait été plus difficile à traiter (il aurait fallu des experts en sécurité).