RFC 9621 : Architecture and Requirements for Transport Services

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 23 janvier 2025

Date de publication du RFC : Janvier 2025

https://www.bortzmeyer.org/9621.html

Ce RFC s'inscrit dans un projet IETF datant de quelques années : formaliser davantage les services que rend la couche Transport aux applications. Le but est de permettre de développer des applications en indiquant ce qu'on veut de la couche Transport, en rentrant le moins possible dans les détails. Ce RFC décrit l'architecture générale, le RFC 9622 [1] spécifiant quant à lui une API plus concrète.

Des API pour les applications parlant à la couche Transport, il y en a beaucoup. La plus connue est bien sûr l'interface « socket », qui a pour principal avantage d'être ancienne et très documentée, dans des livres (comme le génial Stevens <https://www.bortzmeyer.org/unix-network-programming.html>) et dans d'innombrables logiciels libres à étudier (mais, par contre, la norme POSIX qui la spécifie officiellement, n'est toujours pas disponible publiquement). C'est d'ailleurs cette multiplicité d'API, donc beaucoup sont solidement installées dans le paysage, qui fait que je doute du succès du projet et de sa TSAPI ("*Transport Services API*") : la base installée à déplacer est énorme.

Pourtant, ces API traditionnelles ont de nombreux défauts. Le RFC cite par exemple l'incohérence : pour envoyer des données sur une prise qui utilise du TCP en clair, on utilise `send()` mais quand la communication est chiffrée avec TLS, c'est une autre fonction (non normalisée). Pourtant, pour l'application, c'est la même chose.

Toujours question cohérence, la terminologie n'est pas fixée et des termes comme "*flow*", "*stream*", "*connection*" et "*message*" ont des sens très différents selon les cas.

1. Pour voir le RFC de numéro NNN, https://www.ietf.org/rfc/rfcNNN.txt, par exemple https://www.ietf.org/rfc/rfc9622.txt

Le but de la nouvelle architecture est donc de permettre le développement d'une API (normalisée dans le RFC 9622) qui unifiera tout cela. (Notez que l'IETF, habituellement, normalise des protocoles et pas des API. Mais il y a déjà eu des exceptions à cette règle.) Un autre objectif est de permettre de mettre dans les bibliothèques du code qui autrefois était souvent dupliqué d'une application à l'autre, comme la mise en œuvre de l'algorithme des « globes oculaires heureux » (RFC 8305).

J'avais déjà présenté le projet TAPS, qui était alors nettement moins avancé, dans un exposé à la conférence MiXiT <https://mixitconf.org/2018/le-reseau-vu-du-langage-de-programmation-que (dont voici le support (en ligne sur https://www.bortzmeyer.org/files/mixit-api-reseau.pdf)).

L'architecture présentée ici est dérivée de l'analyse des protocoles de transport du RFC 8095 et de leur synthèse dans le RFC 8923. Pour les questions de sécurité, il est également conseillé de lire le RFC 8922.

L'interface pour les applications (TSAPI : "*Transport Services Application Programming Interface*") sera fondée sur l'asynchronicité, en mode programmation par évènements. (Opinion personnelle : c'est une mauvaise idée, excessivement favorable aux langages sans vrai parallélisme, comme C. J'aurais préféré un modèle synchrone, fondé sur des fils d'exécution parallèle, plus simple pour la·e programmeur·se.)

Une des idées de base du projet est de développer une API qui soit indépendante du langage de programmation. C'est à mon avis fichu dès le départ. Outre cette question du parallélisme, la gestion des erreurs est très variable d'un langage à l'autre : codes de retour (en C), exceptions, valeurs mixtes comme en Haskell ou en Zig…L'API du RFC 9622 ne plaira donc pas à tout le monde.

La section 2 du RFC expose le modèle utilisé par l'API. D'abord, un résumé de la traditionnelle API « socket » :

— TCP et UDP sont mis en œuvre dans le noyau,
— les applications utilisent l'API socket pour parler au noyau, choisissant TCP ou UDP (même si l'API socket se prétend plus abstraite que cela, avec des termes comme `SOCK_STREAM` mais, en pratique, son utilisation nécessite de connaitre le protocole de transport utilisé),
— la résolution de noms en adresses se fait par un mécanisme distinct (typiquement, en appelant le résolveur DNS <https://www.bortzmeyer.org/resolveur-dns.html>, directement ou indirectement).

La nouvelle API généralisera ce modèle : l'application utilise l'API "*Transport Services*" (TSAPI), en indiquant les propriétés souhaitées pour la connexion, et la mise en œuvre de l'API parlera à tous les protocoles de transport (et de chiffrement) possibles, qu'ils soient placés dans le noyau ou pas. La résolution de noms est intégrée dans ce cadre, entre autres parce que certains services sous-jacents, notamment TLS, nécessitent de connaitre le nom du service auquel on accède (ce qui n'est pas le cas avec l'API socket, qui ne manipule que des adresses IP).

Pendant qu'on en parle, cette idée d'intégrer la résolution de noms est également dans des projets comme Connect by Name <https://nlnet.nl/project/ConnectByName/>, qui vise à permettre aux applications de juste demander `connect(domain-name, port, tls=true)` et que tout soit fait proprement par l'API, notamment les fonctions de sécurité (vérifier le nom dans le certificat, par exemple, ce qui n'est pas trivial à faire proprement).

On l'a dit plus haut, TSAPI est asynchrone. Le RFC estime que ce mode d'interaction avec le réseau, fondé sur des évènements, est plus fréquent et même « plus naturel ». C'est un des reproches que je ferais au projet. Je préfère le modèle d'interactions synchrone, notamment pour des langages de programmation ayant la notion de fils d'exécution séparés (les goroutines en Go, les processus en Erlang

ou Elixir, les tâches d'Ada...), dans l'esprit des "*Communicating sequential processes*" (et j'en profite pour recommander le génial livre de Hoare <http://www.usingcsp.com/>, bien qu'il soit assez abstrait et d'un abord austère). Rien n'est naturel en programmation (ou ailleurs : « la seule interface utilisateur naturelle est le téton ; tout le reste est appris ») et le RFC se débarrasse un peu vite du choix effectué et ne cherche pas vraiment à le justifier.

Donc, l'application qui veut, par exemple, recevoir des données, va appeler l'API pour demander une lecture et l'API préviendra l'application quand des données seront prêtes. Au contraire, dans un modèle synchrone, le fil qui veut lire sera bloqué tant qu'il n'y aura rien à lire, les autres fils continuant leur travail.

Dans l'API socket, on peut avoir des messages séparés lorsqu'on utilise UDP mais, avec TCP, c'est forcément une suite d'octets, sans séparation des éventuels messages. Chaque protocole applicatif doit donc, s'il en a besoin, inventer sa méthode de découpage en messages : DNS et EPP indiquent la longueur du message avant d'envoyer le message, HTTP/1 l'indique dans l'en-tête mais a un mécanisme de découpage fondé sur des délimiteurs (cf. RFC 9112, sections 6.1 et 7), HTTP/2 et HTTP/3 ont leurs ruisseaux (une requête par ruisseau), TLS utilise également une indication de longueur. Au contraire, avec TSAPI, il y a toujours un découpage en messages, qui seront encodés en utilisant les capacités du protocole de transport sous-jacent (par un mécanisme nommé cadreur - "*framer*").

TSAPI permet des choses qui n'étaient pas possibles avec les API classiques, comme d'avoir plusieurs paires d'adresses IP entre les machines qui communiquent, avec changement de la paire utilisée pendant une connexion.

Un des intérêts de cette nouvelle API est de permettre aux applications de fonctionner de manière plus déclarative et moins impérative. Au lieu de sélectionner tel ou tel protocole précis, l'application déclare ses exigences, et TSAPI se débrouillera pour les lui fournir (section 3.1).

Bon, c'est bien joli, de vouloir que les applications ne connaissent pas les protocoles (ce qui permet de les faire évoluer plus facilement) mais, quand même, parfois, il existe des fonctions très spécifiques d'un protocole particulier qu'il serait bien sympa d'utiliser. La section 3.2 de notre RFC couvre donc cette question. Par exemple, l'option "*User Timeout*" de TCP (RFC 5482) est bien pratique, même si elle est spécifique à TCP. Donner accès à ces spécificités soulève toutefois un problème : parfois, l'application souhaite une certaine fonction, et peut donc s'en passer, et parfois elle l'exige et la connexion doit échouer si elle n'est pas disponible pour un certain protocole. L'API devra donc distinguer les deux cas.

Une fois que l'application a indiqué ses préférences et ses exigences, l'API devra sélectionner un protocole de transport. Parfois, plusieurs protocoles conviendront. Si l'application demande juste un transport fiable d'une suite d'octets, TCP, SCTP et QUIC conviendront tous les trois (alors qu'UDP serait exclu).

Comme rien n'est gratuit en ce bas monde, le progrès de l'abstraction, et une relative indépendance de l'application par rapport aux particularités des protocoles de transport, ont un inconvénient : le système devient plus complexe. Les développeureuses d'application lorsqu'il faut déboguer, et les administrateurices système qui vont installer (et essuyer les plâtres !) les applications vont avoir des difficultés nouvelles. Des fonctions de déboguage devront donc être intégrées. (Personnellement, je pense aussi à la nécessité que la bibliothèque qui mette en œuvre TSAPI coopère au déboguage pour les protocoles chiffrés, qui ne permettront pas tellement d'utiliser Wireshark. Voir mon exposé à Capitole du Libre <https://www.bortzmeyer.org/capitole-du-libre-2022.html>.)

Et enfin, après ces généralités, passons aux concepts qui devront être visibles par l'application, rapprochons-nous de la future API (section 4 du RFC). D'abord, la Connexion. Non, ce n'est pas une erreur d'avoir capitalisé le mot. Le RFC utilise la même méthode pour montrer qu'il s'agit d'un concept abstrait, qui ne correspond pas forcément parfaitement aux connexions du protocole de transport (pour ceux qui ont ce concept). La Connexion est donc au cœur de l'API "*Transport Services*". Elle se fait entre deux points, le local et le distant, chacun identifié par un "*Endpoint Identifier*". Elle a un certain nombre de propriétés, qui peuvent s'appliquer avant (et influenceront la sélection du protocole de transport, et/ou des machines auxquelles ont se connecte), juste avant ou pendant la Connexion, par exemple l'utilisation du 0-RTT (RFC 9622, section 6.2.5) ou l'exigence du chiffrement (cf. RFC 8922), mais il y a aussi des propriétés pour chaque message individuel. Ensuite, on va :

— Pré-établir : ce sont les opérations qu'on fait avant de se connecter, comme de définir des paramètres de la Pré-Connexion, la future Connexion.
— Établir : là, ça y est, on se connecte.
— Transférer des données (lire et écrire).
— Gérer des évènements qui surviennent comme « la Connexion est prête » ou « le chemin a changé ». Rappelez-vous que TSAPI est asynchrone, pilotée par des évènements.
— Et, comme les meilleures choses ont une fin, on peut terminer la Connexion (proprement avec `Close` ou salement avec `Abort`).

Un écoutant va créer une Pré-Connexion et attendre les demandes, ce n'est pas lui qui va établir les connexions. L'entité active, elle, va se connecter (fonctions `Listen` et `Initiate`). Et le pair-à-pair ? Il sera également possible via la fonction `Rendezvous`. Ce sera à l'API de gérer les services comme STUN (RFC 8489), souvent indispensables en pair-à-pair.

Le transfert de données a les classiques fonctions `Send` et `Receive` (qui n'est pas synchrone, rappelons-le, donc qui ne bloque pas). Mais il y a aussi un nouveau concept, le cadreur (traduction peu imaginative de "*framer*"). C'est la couche logicielle qui va ajouter et/ou modifier les données du message pour mettre en œuvre des exigences du protocole, par exemple indiquer les frontières entre messages dans un protocole de transport qui n'a pas ce concept (comme TCP). Ainsi, les protocoles EPP (RFC 5734) et DNS (RFC 7766) nécessitent que chaque message soit préfixé d'un seizet indiquant sa longueur. Un cadreur peut faire cela automatiquement pour chaque message, simplifiant la tâche de l'application (par exemple la bibliothèque standard d'Erlang a cette option <`https://www.erlang.org/doc/apps/kernel/inet.html#setopts/2`> - cf. `packet`).

Enfin, un peu de sécurité et de vie privée (section 6). D'abord, les implémentations devront faire attention aux risques pour la vie privée si on réutilise l'état des connexions passées. C'est le cas des sessions TLS (RFC 8446, section 2.2) ou des gâteaux HTTP (RFC 6265), qui peuvent permettre de relier deux connexions entre elles, lien qu'on ne souhaite peut-être pas.

Et puis, en sécurité, le RFC rappelle qu'on ne doit pas se replier sur des solutions non sécurisées si la technique sécurisée échoue. Ainsi, si une application demande du chiffrement, mais que la connexion chiffrée échoue, il ne faut pas se rabattre sur une connexion en clair.

Notez que l'ensemble du projet TAPS <`https://datatracker.ietf.org/wg/taps/`> a bénéficié de divers financements publics, par exemple de l'UE.

Une opinion personnelle sur ce projet ? Outre des critiques sur tel ou tel choix, comme l'asynchronisme, je suis malheureusement sceptique quant aux chances de réussite. Les API actuelles, avec tous leurs défauts, sont solidement installées, et dans le meilleur des cas, il faudra de nombreuses années pour que les formations s'adaptent et enseignent une API comme TSAPI.