Protocoles Réseaux VII
BROUILLON

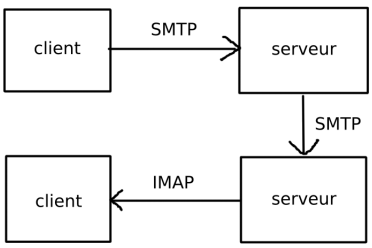
Juliusz Chroboczek

16 novembre 2021

1 Applications réseau

Le but du réseau est de servir de support à des *applications réseau*, des applications qui consistent de plusieurs processus qui communiquent entre eux à travers le réseau. Les protocoles qu'utilisent les applications réseau sont des *protocoles de couche application*, et sont construits sur des protocoles de couche transport.

Une application réseau complexe utilise plusieurs protocoles de couche application simultanément. Prenons par exemple le cas du courrier électronique, le « mail », *e-mail* en anglais). Un courrier est soumis par le client à son serveur local par le protocole SMTP ou sa variante LMTP; le courrier transite ensuite entre serveurs par SMTP, jusqu'à atteindre le serveur destinataire. Lorsque le récipient décide lire son courrier, il y accède à travers le protocole IMAP. (Dans le cas du *webmail*, il y a un protocole supplémentaire : le client *web* accède au client SMTP/IMAP à travers le protocole HTTP.)



2 La couche application

Les protocoles implémentés dans l'application sont à la *couche application*.

2.1 Messages

Aux couches 2 à 4 (lien, réseau et transport), chaque couche encapsule les données de la couche directement au dessus : le segment ou datagramme de couche 4 était encapsulé pour donner un paquet de couche 3 qui lui-même était encapsulé dans une trame de couche 2. La couche 1 (physique) est différente : la trame de couche 2 est transmise par la couche physique sous forme d'une suite de symboles.

La couche 7 (appication), elle aussi, ne fait pas de la simple encapsulation. L'application manipule des structures de données complexes, qui peuvent être des entiers, des chaînes de caractères, des tableaux, des enregistrements (des « structures » en C, ou des « objets » en Java), des tableaux d'enregistrements, ou même des structures chaînées telles que des arbres ou des graphes généraux. Le protocole de couche application sera chargé de coder (« sérialiser ») ces structures sous une forme acceptable par le protocole de couche transport : des flots d'octets si le protocole de couche transport est TCP, et des suites de taille bornée d'octets si c'est UDP.

2.2 Digression : la couche session

Une application réseau complexe manipule plusieurs connexions parallèles. Par exemple, une application de vidéoconférence manipule au moins trois connexions :

- une connexion TCP pour le contrôle (établissement de session, négociation des paramètres audio et vidéo, négociation des clés cryptographiques) ;
- un flot UDP pour le traffic audio ; et
- un autre flot UDP pour le traffic vidéo.

Manipuler ces nombreuses connexions est complexe, et mène souvent à des bugs. (En termes techniques, même si chaque connexion obéit à un automate simple, la composition parallèle de ces automates est complexe.)

Dans le modèle OSI d'origine, celui qui avait sept couches, la couche 5 s'appelait la *couche session* et était chargée de manipuler plusieurs connexions parallèles comme une seule entité. L'application aurait pu demander d'établir une « session » consistant de plusieurs connexions, et la couche session aurait retourné soit un succès (si toutes les connexions avaient pu être établies), soit une indication d'erreur (si au moins une avait échoué).

La notion de couche session a été abandonnée à la fin des années 1980, et depuis trente ans les applications se limitent à une seule connexion, ou alors gèrent elles même toute la complexité de la gestion des connexions multiples. Cependant, le protocole ICE (RFC 8445), utilisé par les applications de vidéoconférence récentes, ressemble fort à un protocole de couche session.

3 Structure des applications réseau

3.1 Relations entre pairs

client-serveur, pair-à-pair, structures hybrides

3.2 Structure de la communication

Requête-réponse (synchrone, asynchrone, agrégation).
Notifications asynchrones.

3.3 Extensibilité

Un protocole de couche application évolue au cours du temps. Par exemple, le protocole SMTP a été défini en 1982, et ne supportait alors que les messages textuels limités au jeu de caractères

ASCII. En 1996, il a été étendu pour supporter le texte international (les accents français, les alphabets autres que l'alphabet latin) et les attachements binaires. Le codage des attachements binaires était peu-efficace, ce qui n'était pas un problème jusqu'à ce que les commerciaux découvrent le mail, et commencent à s'envoyer entre eux des documents *Powerpoint* gigantesques ; en 2000, le protocole a de nouveau été étendu pour coder les attachements binaires de façon efficace.

Il existe trois techniques principales pour rendre un protocole extensible, qu'il est possible de toutes employer en même temps.

3.3.1 Protocole versionné

La technique la plus simple consiste à donner au protocole un numéro de version. Lorsqu'il se connecte au serveur, le client annonce la liste de protocoles qu'il supporte ; le serveur répond en indiquant la version qu'il a choisie. C'est plus ou moins ainsi que fonctionne par exemple HTTP.

Le versionnage du protocole est simple et facile à implémenter ; cependant, il requiert un ordre total sur les versions du protocole, ce qui ne permet pas naturellement à deux entités indépendantes (par exemple concurrentes) d'étendre le protocole dans deux directions différentes.

3.3.2 Négociation d'options

Une technique plus flexible consiste à définir un protocole minimal et des fonctionnalités optionnelles. Lorsqu'il se connecte au serveur, le client annonce l'ensemble des fonctionnalités qu'il implémente ; le serveur répond avec l'ensemble des fonctionnalités qu'il a sélectionnées. Le protocole SMTP fonctionne plus ou moins ainsi, sauf que c'est le serveur qui offre des fonctionnalités et le client qui les sélectionne.

3.3.3 Protocole extensible

Un protocole est extensible s'il est possible pour un pair d'ignorer des parties des messages. Par exemple, en HTTP, le serveur ignore les entêtes d'une requête qu'il ne comprend pas ; cela permet au client d'insérer des entêtes qui demandent des fonctionnalités supplémentaires sans se préoccupper de savoir a priori si le serveur les implémente.

4 Syntaxe des messages

À la couche application, une connexion échange une suite de messages. Pour être transportés par une connexion TCP ou un flot UDP, ces messages doivent être *sérialisés* par l'émetteur, c'est-à-dire codés comme une suite d'octets, et *désérialisés* par le récepteur. L'algorithme de sérialisation/désérialisation doit résoudre deux problèmes : comment séparer les messages et comment coder les données.

4.1 Séparation entre messages

4.1.1 Un message par datagramme

La technique la plus simple pour séparer les messages est d'envoyer un message par entité de couche transport. Avec UDP, c'est facile, il suffit d'envoyer un message par datagramme (attention, cependant, aux limites de taille des datagrammes). Avec TCP, c'est généralement impossible : TCP ne préserve pas les frontières entre `write`, et la seule structure préservée par TCP est la suite d'octets ; or, il est rare qu'un message puisse être codé dans un seul octets.

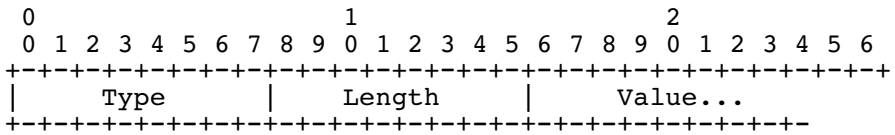
Il existe des protocoles de couche transport fiables qui préservent les frontières entre messages ; le plus connu est SCTP, initialement conçu pour les applications de télécommunication. Malheureusement, SCTP ne traverse pas les NAT, et n'est donc pas utilisable dans l'Internet actuel. Il existe une variante de SCTP encapsulée dans UDP, qui est utilisée notamment par certaines applications de vidéoconférence.

4.1.2 Séparateur explicite

On peut aussi séparer les messages par un séparateur explicite. C'est souvent le cas dans les protocoles textuels : SMTP utilise un caractère de fin de ligne « CR-LF » pour terminer les commandes, et HTTP/1.0 utilise une ligne vide « CR-LF-CR-LF ». C'est plus difficile dans les protocoles binaires, car le séparateur peut apparaître dans le contenu d'un message, ce qui force l'utilisation de techniques de « citation » complexes et fragiles. (Nous en avons vu un exemple à la couche lien, lorsque nous avons étudié le protocole SLIP.)

4.1.3 TLV

Un *TLV* est un triplet *Type-Longueur-Valeur*. Une application basée sur les TLV code une suite de message par une suite de TLV, un par message. Un TLV peut par exemple avoir la structure suivante :



Le champ *Type* indique le type du message, il a un sens qui est spécifique à l'application. Le champ *Length* indique la longueur du message, et permet donc d'en déterminer la fin ; il y a deux variantes : *Length* peut coder la taille en octets du champ *Value*, ou alors il peut coder la taille totale du TLV. Enfin, le champ *Value* contient le message lui-même, sous un format spécifique à l'application.

Il y a bien sûr plusieurs variations possibles sur ce thème : les champs *Type* et *Length* peuvent avoir une longueur de 2 ou même 4 octets, et on peut même omettre le champ *Type* s'il n'est pas utile.

La structure en TLV est facile à implémenter de manière efficace, et elle est extensible : pour ignorer un TLV de type inconnu, il suffit d'incrémenter un pointeur de la valeur du champ *Length* sans nul besoin d'en parser le contenu (comparez cela au format XML, qui n'est pas extensible :

pour ignorer un morceau de XML, il faut savoir le parser afin de trouver où se trouve la balise fermante).

5 Représentation des données

Que les messages soient des datagrammes, des TLV ou qu'ils soient séparés par un séparateur explicite, ils contiennent une représentation des données de l'application. Ces dernières doivent donc être codées sous forme d'une suite d'octets.

Il existe deux catégories de représentations : les représentations textuelles, qui représentent les données par du texte, et les représentations binaires.

5.1 Représentation textuelle

L'avantage principal des représentations textuelles est qu'elles peuvent être facilement affichées, ce qui rend les applicacations plus faciles à développer et à débogguer. Elles sont cependant moins efficaces que les représentations binaires.

5.1.1 Représentation textuelle *ad hoc*

Il n'est généralement pas trop difficile de définir une représentation textuelle spécifique à l'application. La technique la plus simple consiste à définir une grammaire LL(1) pour coder les données (un parcours préfixe d'un arbre, s'il est bien fait, mène à une grammaire LL(1)). Il est alors facile d'écrire un parseur par descente récursive.

5.1.2 Représentation textuelle générique

Comme la plupart des ingénieurs n'aiment pas écrire des parseurs, la communeauté a défini un certain nombre de représentations textuelles génériques, capables en principe de stocker n'importe quelle structure de données.

XML XML est un format textuel générique qui était à la mode au début des années 2000. XML provient de la communeauté de la représentation du texte enrichi, et n'est donc pas adapté à la représentation des structures de données. Par exemple, il existe de nombreuses manières de stocker une liste d'entiers, dont chacune à des avantages :

```
<list><int>1</int><int>2</int><int>3</int></list>
<list><int value="1"/><int value="2"/><int value="3"/></list>
<list value1="1" value2="2" value3="3"/>
<list values="1 2 3"/>
```

L'inadaptation de XML cause beaucoup de complexité inutile, ce qui permet aux consultants spécialisés en XML de gagner beaucoup d'argent. Même si XML n'est plus à la mode, il est utilisé par beaucoup de protocoles développés au début des années 2000, et il est nécessaire de le connaître.

Il est important de comprendre que XML code des messages (des « document » en langage de la communeauté XML), et pas des flots. Il est donc nécessaire de stocker les documents XML à l'intérieur de datagrammes ou de TLV afin de pouvoir détecter les frontières entre les messages [1].

JSON Le format JSON est un format textuel générique développé par la communeauté des applications *web*. JSON est adapté à la représentation des « objets » Javascript : il code les nombres, les chaînes de caractères, les tableaux et les dictionnaires indexés par des chaînes. Une liste d'entiers se code naturellement en JSON comme un tableau de nombres :

```
[1, 2, 3]
```

Le principal défaut de JSON est que, tout comme Javascript, il ne distingue pas entre les entiers et les nombres à virgule flottante. Cependant, JSON garantit que tout entier de 53 bits ou moins peut être représenté sans erreur d'arrondi.

5.2 Représentations binaire

5.2.1 Représentation des entiers

5.2.2 Codage des nombres en virgule flottante

5.2.3 Codage des chaînes

5.2.4 Codage des données composites : sous-TLV

5.2.5 Codages binaires génériques

1. Le protocole XMPP, notamment, commet l'erreur de vouloir définir une notion de « flot XML ».