Réseaux 7
La couche application

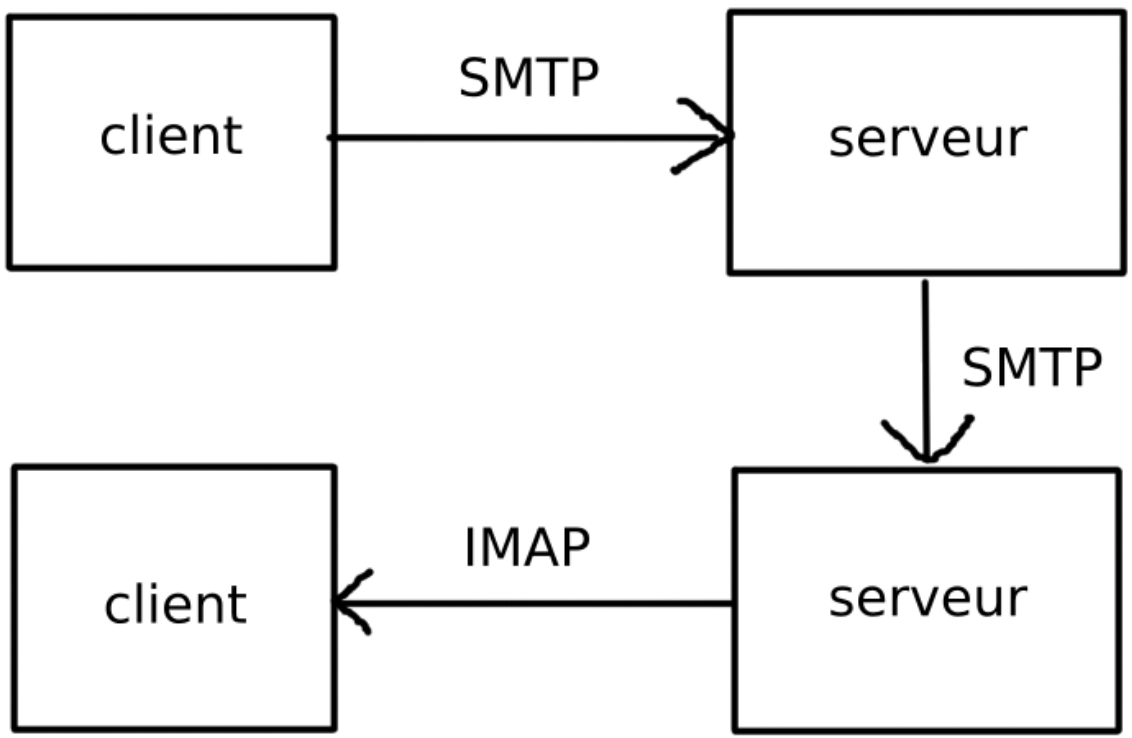
Juliusz Chroboczek

9 novembre 2020

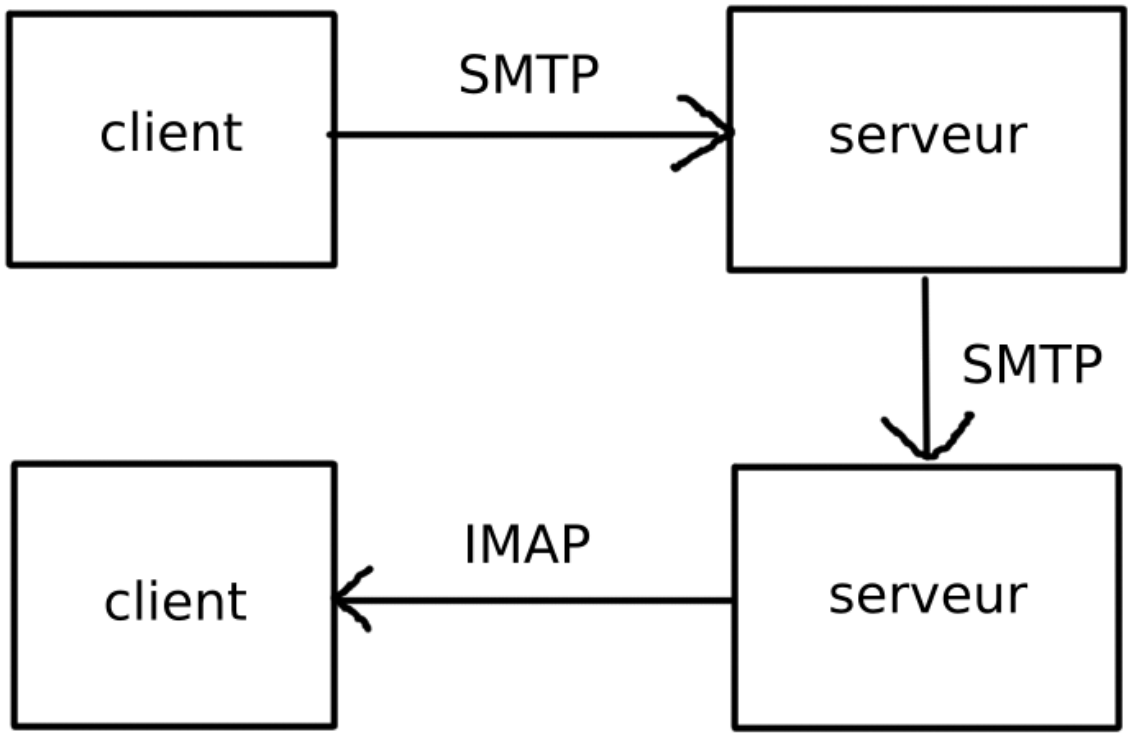
Application réseau

But : faire des applications réseau
(tout le reste est du bavardage).

Application réseau : application constituée de plusieurs processus communiquant à travers le réseau.



Application réseau



- À quoi sert le premier serveur ?
- que faut-il ajouter pour le *Webmail* ?

Message

Aux couches 2 à 4, on fait de l'encapsulation :

$$\text{segment} \longrightarrow \text{paquet} \longrightarrow \text{trame}$$

Les couches 1 et 7 sont différentes :

- couche 1 : symboles, pas d'encapsulation évidente ;
- couche 7 : message, dont la nature dépend de l'application.

Un message est l'unité d'information transmise par l'application.

La couche application doit coder ses messages par des suites d'octets.

Digression : la couche Session

Une application réseau complexe gère plusieurs connexions de couche transport simultanément :

- une connexion Websocket/TCP pour le chat ;
- un flot RTP/UDP pour l'audio ;
- un flot RTP/UDP pour la vidéo.

Que faut-il faire si l'une de ces connexions échoue ?

Le rôle de la couche Session est de combiner ces connexions en une seule session.

Pas de couche session : la gestion des sessions est à la charge de l'application !

(Peut-être ICE est-il un protocole de couche session ?)

Relations entre pairs

- Client-Serveur : un ou plusieurs clients communiquent avec un serveur (généralement aligné avec la couche transport, mais pas forcément) ;
- pair-à-pair : pas de structure, chacun parle à chacun.

La structure se construit sur ces deux relations :

- dans SMTP, un serveur devient client ;
- dans HTTP, un client parle à plusieurs serveurs simultanément ;
- dans IRC, les serveurs sont en relation pair-à-pair.

Structure de la communication

- Requête-réponse synchrone (SMTP)
  le client envoie une requête puis attend la réponse
  facile à implémenter
  problème de latence ;
- requête-réponse asynchrone (X11, HTTP/1.1)
  le client peut envoyer plusieurs requêtes avant d'avoir une réponse
  facile côté serveur, difficile côté client
  gestion des erreurs impossible ;
- requête-réponse par paquets *(batches)* (SMB)
  plusieurs requêtes concaténées,
  le serveur arrête le traitement à la première erreur.

Structure de la communication (2)

Événements asynchrones (X11, Spritely-NFS) :

- pas des requêtes (pas de réponse associée) ;
- pas des réponses (non sollicités).

Évitent les problèmes de latence,
peuvent être combinés avec requête-réponse,
difficiles à implémenter sans deadlock.

En pair-à-pair, plusieurs structures sont possibles :

- requête-réponse dans les deux sens ;
- événements asynchrones ;
- les deux.

Extensibilité

Un protocole évolue au cours du temps.

Exemple : SMTP :

- 1982 : texte ASCII ;
- 1996 : texte international, attachements binaires ;
- 2000 : attachements binaires efficaces.

Trois techniques :

- protocole versionné : on change tout
  ex : HTTP/2, HTTP/3 ;
- négociations d'options : on négocie les options
  ex : (E)SMTP ;
- protocole extensible : on ignore les messages inconnus
  ex : HTTP/1.1.

Syntaxe des messages

À la couche application, une connexion échange une suite de messages.
Codés dans un flot TCP ou une suite de datagrammes UDP (des suites d'octets).

- comment séparer les messages ?
  - un message par datagramme UDP ;
  - messages séparés par un séparateur ;
  - TLV.
- comment coder les messages ?
  - textuel ad-hoc (LL(1)) ;
  - textuel formalisé (XML, JSON) ;
  - binaire ad-hoc ;
  - binaire formalisé (sans ou avec schéma).

Un message par datagramme UDP

Applicable à UDP. Non-fiable.

Inapplicable à TCP ou QUIC.

(Vivement qu'on déploie SCTP, qui a une sémantique par messages fiables.)

Messages séparés par un séparateur

On peut séparer les messages par une chaîne distinguée :

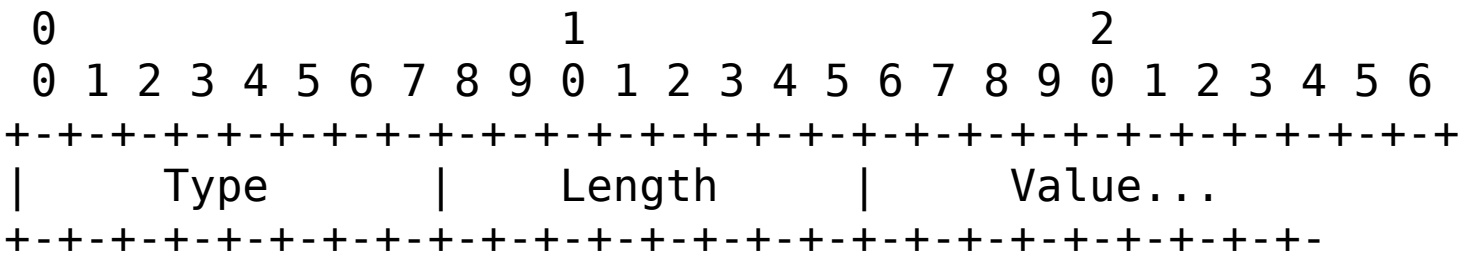
- dans SMTP, commandes sont terminées par `\n\r` ;
- en HTTP, entêtes sont terminés par `\n\r\n\r`.

Souvent utilisé dans les protocoles textuels.

Peu pratique dans les protocoles binaires : il faut citer le séparateur lorsqu'il apparaît dans les données.

TLV

Un TLV est un triplet Type-Longeur-Valeur.



Avantages :

- facile à implémenter ;
- efficace ;
- extensible (ignorer : incrémenter un pointeur).

Représentation textuelle

On aime bien : facile à débogguer.

Exemples : SMTP, NNTP, HTTP/1…

Technique ad hoc :

- concevoir un langage LL(1) ;
- écrire un parseur par descente récursive.

Représentation textuelle générique

Les gens ne savent pas écrire des parseurs.
Ils croient que c'est difficile.
(Ils ont séché compil L3 ?)

On a donc défini des formats textuels génériques :

- on utilise une bibliothèque toute faite ;
- c'est magique, il n'y a rien à faire ;
- on passe le reste de sa vie à débogguer une bibliothèque qu'on ne comprend pas.

XML

XML : format générique avec plein de « < » et « > ».

- que code XML ? des arbres ?
- complexe : quelle est la distinction entre

```
<user name="jch"/>
<user>jch</user>
<user><name>jch</name></user>
<user><name value="jch"/></user>
```

XML code des documents, pas des flots.

Il faut donc le combiner avec une technique de découpage en messages.
(REST, TLV, ou séparateur explicite.)

JSON

JSON : alternative plus simple à XML.

- au lieu de plein de « < », on a des « { » ;
- code clairement des arbres ;
- s'intègre bien à *Javascript* ;

JSON code des objets, par des flots.

Généralement stocké dans des TLV *(WebSockets)*.

Représentation des données binaires

TCP implémente des flots d'octets.
Le corps d'un datagramme UDP est une suite d'octets.

Il est donc nécessaire de coder les données comme des flots d'octets :

- entiers : boutisme, signe ;
- flottants : représenatation ;
- texte : codage.

Boutisme

Un entier d’un octet peut être envoyé tel quel.

Pour un entier de deux octets, il y a deux ordres possibles :

$$1026 = 4 \times 2^8 + 2$$

On peut envoyer

- 4, 2 (gros-boutiste, *big-endian*, *Motorola-endian*) ;
- 2, 4 (petit boutiste, *small-endian*, *VAX-endian*).

Solution : chaque protocole définit le boutisme des entiers.
La communauté du réseau préfère l’ordre gros-boutiste : « network-endian ».
(Certains protocoles font le contraire : BitTorrent.)

Codage des flottants

On évite d'utiliser les flottants, on préfère la virgule fixée.
Ex : temps en ms.
Si les flottants sont nécessaires, on utilise le format IEEE 754.
Attention aux non-nombres (∞, –0, NaN).

Codage des chaînes

1963 : ASCII, jeu de 95 caractères numérotés de 32 à 126.
Années 1980 : jeux de caractères spécifiques à une région :

- CP437 (IBM PC) : Europe de l'Ouest ;
- CP852 (IBM PC) : Europe de l'Est ;
- KOI-8 : Russie ;
- *Apple Roman* (Mac OS) : Europe de l'Ouest ;
- CP1252 (Windows) : Europe de l'Ouest ;
- GB2312 : Chine Populaire ;
- Big5 : Chine Nationaliste ;
- Shift JIS (IBM PC) : Japon ;
- ...

Unicode

Dans les années 1990, le consortium Unicode s'est réuni pour définir un jeu de caractères universel :

- jeu de caractères — ne définit pas le codage ;
- vise à couvrir tous les systèmes d'écriture
  (cf. l'Arabe — chaque lettre a jusq'à quatre formes)
  (et je ne vous parle même pas du lam-alif)
  (et si on commence à parler du *Nastaliq*…) ;

UTF-8 est un système de codage qui code les suites de caractères Unicode par des suites d'octets.
Tous les protocoles modernes utilisent UTF-8.

Formats binaires génériques

Comme pour les formats textuels, les gens ont défini des formats génériques :

- sans schéma : CBOR, BSON, MessagePack ;
- avec schéma : ProtoBuf.

Ça n'a pas pris — il n'y a pas besoin d'avoir fait compil L3 pour implémenter des TLV, même imbriqués.