Protocoles réseaux : cours1

02 octobre 2023

1 Rappels

Cours m1 :

$$\text{application} \neq \text{protocole}$$

Protocoles de couche application :

- PC→serveur de mail, deux protocoles : SMTP, IMAP4.
- tel→serveur de mail : SMTP, IMAP4.
- Fair Email→serveur de mail : SMTP, IMAP4.

$\neq$

- whatsapp ou on peut parler qu'en whatsapp (le client est sur l'appli)(probleme : c'est un protocole ET une appli)
- zulip : on peut faire son propre protocole pour echanger avec le serveur mais le protocole zulip change a chaque vers

1.1 Couches

Les différentes couches :

1. Physique
2. Lien
3. Reseau

4. Transport
5. (7) Application

En bas : le materiel. En haut : utilisateur.

2 Internet est cassé

"internet est cassé".

1. Il est "Ossifié"
2. "Tout le monde vous en veut"

2.1 "Ossification"

Remarque 1. Le routeur a que trois couches, il depasse pas IP et regarde pas les paquets. Permet de changer les protocoles du dessus.

Le problème c'est que certains le font.

Définition 2.1.1. Middlebox : entité au seins du réseau (pas aux bouts, typiquement au dessus de la troisieme couche d'un routeur) qui regarde à l'interieur des paquets.

Ex :

- Firewalls : voit ce qui respecte pas les protocoles ? Et plus mais sert a rien askip.
- En pratique, le firewall jette tout ce qui est pas TCP, UDP. Problématique.
- NAT, pareil jette les paquets d'autres protocoles.
- accélerateurs
- black boxes (déployé par les services internet du a ansi/DGSE/autres, espionnage)
- NAT→limitations de la connectivité.

Lemme 2.1.2. *Pour programmer sur le vrai internet :*

- *Faut utiliser TCP/UDP.*

Faut traverser le NAT/Firewall.

Ducoup on utilise des protocoles qui traversent le NAT et le Firewall.

Lemme 2.1.3. *Http/https traversent le NAT/Firewall*

Théorème 2.1.4. *La vraie couche de convergence c'est pas IP c'est http/https.*

Remarque 2. C'est un protocole nul : requete/reponse avec 500 octets d'entete a chaque paquet. Autre triche :

- Http utilise un autre protocole de transport encapsulé en UDP, quick ou microTP !
- ducoup on considère UDP en couche 3.5 et ducoup 4 octets par paquet au lieu des 500 de http.

Questions de **Sécurité**.

3 Web et http

(Web = application, http = protocole application)

Définition 3.0.1. Je cite "Le Web c'est un hypertexte distribué":

1. hypertexte : contient des hyperlien (lien qui envoient ailleurs dans le fichier)
2. distribué : les hyperliens envoient dans ailleurs ("impossible" de gerer les liens qui vont nul part → error 404 (on s'en fout))

3.1 http/0.9

- Ressources identifiées par des URL : "http://www.irif.fr/ jch/enseignement
- En rouge le chemin : http → Client envoie GET(chemin) au serveur qui renvoie la ressource a l'adresse.
- Plein de defauts : le serveur fait RIGHT jusqu'a la fin FIN et close(fichier). Probleme si le serveur plante et coupe la co on peut pas savoir si c'est une interruption ou un fichier court.

Remarque 3.

1. TCP est lent au demarrage (petite fenetre qui grandit au fur et a mesure et devient efficace)
2. A chaque GET, http ouvre une connexion TCP (PB serait : TCP=couche 4, http=couche 7, il faudrait une autre couche)
3. pb quand il y a eu des img dans des fichiers (chaque balise IMG engrengeait un GET)

http://www.irif.fr/ jch/enseignement

3.2 http/1.0

On a maintenant plus de requetes

1. GET
2. POST
3. PUT
4. -version

Par exemple : GET /index.html .HTTP/1.0 permet d'obtenir /index.html avec une version 1.0 ou anterieure.
Nouvelles entetes :

- Content-Type : text/html (possibilité de faire attention a pas lire un fichier pas .html en html et avoir d'autres extensions en particuliers images)
- Content-Length : résoud un des pbs

3.3 http/1.1

pas mal

- entete en type texte
- terminaison fiable(chunked)
- transferts multiples sur 1 connexion

A ce moment la → : -Javascript (scripts dans les pages web)(pas forcément utile en soit).

- XMLHttpRequest permet de faire des requete Http depuis le script !

Ducoup application web !

Théorème 3.3.1. *Javascript + XMLHttpRequest → Web 2.0*

AJAX : L'interface est plus gerée par l'utilisateur.

1. Le serveur envoie une page html
2. le programme js envoie des requetes de **structures de données** (point important)
3. le serveur ne renvoie pas du html mais des données qui permettent de remplir la page html

XMLHttpRequest devient fetch.

3.4 http/2

mauvais

- entete en binaire (trop de requetes) (moins lourd : algo de compression)
- multiplexage : requetes/reponses simultanées → pas compatible avec TCP

3.5 http/3

Tres bien mais compliqué : que Google qui utilise car les seuls qui ont reimplementé

- UDP (quick)
- multiplexage efficace

3.6 Sérialisation/désérialisation

http transmet des octets : java/js/python utilisent des structures de données.

- Il faut traduire en octets les structures de données

Comment sérialiser/déserialiser

1. Ad hoc : LL(1) à la main (faire une grammaire)
2. XML : nul
3. JSON :

   1,2,3 (sous ensemble de la syntaxe js)
   - string utf-16 (pas de binaire)
   - nombre (javascript, pas tout)
   - tableaux
   - dictionnaires (clés sont forcément des chaines)

   Bien mieux que XML mais tjr pas super. Le binaire doit être géré ad hoc, y faut aussi sérialiser les structures

4. Formats binaires
   (a) Ad hoc
   (b) génériques : CBOR
   (c) Protobuf : Structures → codage ad hoc a la compilation (dependance) → tres instable (si on change la struct le code peut etre completement different)

but : protocoles qui marche aussi bien pour une appli web que native

4 Structure des API

Pour faire une interface Web : faut ecrire du html. Pour faire une base de donnée SQL : faut ecrire du SQL en html

1. POST /sql.php HTTP/1.1

2. HOST sql.example.com
3. Content-type : application/sql
4. Content-Length: 18
5. DROP TABLE USERS (TABLE USERS est la table sql, TABLE USERS est stockée dans la RAM)

Interet de ça : on utilise toute la puissance de la base de donnée. pb : fuite de l'implémentation. Mieux :

1. POST /sql.php?query=DROP%20TABLE...
2. DELETE /table/users

4.1 Des APIS

SOAP :

- complexe
- utilisant XML ($\rightarrow$ XMC-RPC)

requetes enormes codées en XML "horrible"
REST :

- API représente un ensemble d'états (Objets sans methodes) côté serveur
- Si on a un serveur d'impression pour imprimer on fait pas une requete imprimer un fichier $\rightarrow$ on créer un objet impression
- Pas de codage (on stocke les données telle quelles, pas d'agrégation, un GET pour chacun des attributs)
- etats identifiés par des URL (diff entre DROP TABLE USERS $\leftrightarrow$ DELETE /table/users)
- opératons codées dans la méthode
- Pas d'état de session (pas besoin de se souvenir de chaque client)

lulu.informatique.univ-paris-diderot.fr :8443
La lumière est "glacialement lente" : Cacher la latence de l'univers,

- batching(lots de requetes) → Pas de REST. (probleme de gestion des erreurs, si la premiere requete est refusée on peut plus arreter)
- pipelining.
- prevoir l'avenir .(sur un mmo les npc apparaissent d'abord a un endroit predit par l'ordi puis ailleurs une fois que le serveur repond.)
- Cache ! (copie des données, attention a savoir quelles données sont celles à jour)(pb d'invalidation du cache)

4.2 REST-fut ou REST-like

4.3 Tout le contraire → WebSocket

5 Proxys et caches

5.1 Localisation des caches

- Cache application. (le chat du site qui se met a jour que au reload (les msgs sont en caches))
- Cache navigateur. (partagé entre les pages)
- Proxy

Digression : proxy(censé être proche du client)(serveur mandataire)

Apparemment pas d'interco en couche 4 à 7. Interconnexion

1. couche 1 : prise/hub
2. couche 2 : switch
3. couche 3 : routeur
4. couche 4, 7 (plus "bout en bout") : proxy

Types de proxy HTTP :

- proxy traditionnel : explicitement configuré

- proxy transparent : le client à aucune idée du proxy, par exemple le routeur envoie à un proxy. Pb, c'est un middlebox → mal. Par interception
- proxy inverse permet :
    1. plusieurs serveurs, répliqués, distincts.
    2. politiques de securité
    3. cache (crétin?)

5.2 proxy inverse : CDN

Au moment d'une nouvelle maj ios : 500 000 000 de maj de 10 GO. Stratégies ? Mettre des proxys qui ont la maj PARTOUT. Pb en dehors des majs les proxys sont loués? Solution :

- Il y a des boites de location de reseau de proxys avec cache : akamai (host www.apple.com)

70% du reseau passe par akamai !

5.3 Validations

Une requete HTTP qui semble aller direct au serveur passe en fait par au plus 5 caches (proxy, CDN, proxy inverse). En HTTP il y'a 3 validateurs (metadonnée associée au paquets : si il a changé la donnée est pas reutilisable) :

- URL (si sur un blog on change une entrée, comment on met a jour tout les utilisateurs ? On peut modifier l'URL !)
- Last-Modified (Pas précis, a pas utiliser)
- Etag (meme etag alors on peut reutiliser)

Pas de Etag, pas de Last-Modified su rle document : on utilise pas de cache.

5.4 Controle de Cache

(Ctrl+R, pas de revalidation. Ctrl+shift+R, revalidation de bout en bout)
On veut pas forcément la dernière version systématiquement. Par exemple un logo sur un site.

- l'entete Cache-Control :
    1. no-cache (ctrl+r)
    2. max-age en seconde, update le cache si la donnée est plus veille que max-age.
    3. Pas d'entete :
        - si POST→ On peut pas mettre en cache
        - si GET→ On peut mettre en cache

Pour la revalidation:

1. HEAD : serveur renvoie le Etag
2. if meme Etag : GET

Problème, faut attendre 2 requetes. Sol : GET conditionnel

1. GET (header : if-None-Match) : retourne les données ou retourne none-changed

6 Notifications asynchrone

Pas de notifications asynchrones : Le serveur n'envoie rien sans requetes.

7 web socket

API REST, etc en : http et udp. Ou tcp : flot d'octets 1 par 1. Gros problème.

- Web Socket : On utilise le protocole qu'on veut ! On peut faire des flots d'objets JSON.
- Probleme des Web socket :
    1. perte du cache
    2. perte de la resistance
    3. perte du load balancing