# 8: Couche applications

Martin Quinson



# Épisode précédent

- Forme du réseau (un graphe de Autonomous Systems – réseaux indépendants)
  - types de liens (filaire, wifi, optiques)
  - Protocoles et IETF
  - Modèle de sécurité d'internet (y'en a pas)
- Performance du réseau
  - Commutation de paquets for the win
  - Intuitions fausses (qualité réseau=ordre total, lat/BW parfois liées, un lama va plus vite qu'un lien)
- Modèles en couches
  - TCP/IP en 4 couches (app, transport, routage, physique)
  - OSI en 7 couches, mais juste pour faire joli

# I) Formes d'organisation pour les applications distribuées

- Maintenant qu'on sait faire communiquer des machines entre elles, comment on organise leurs interactions?
- Le coeur de la question à cet étage porte sur les protocoles.

## I.1) client-serveur

- Le modèle le plus simple et tjs efficace
- Serveur: dispo et attent les requêtes (connexion permanente, IP fixe et connue. Plutôt datacenter)
- Client: contacte le serveur au besoin (intermitent, peut avoir IP dynamique)

- Les clients ne communiquent pas entre eux

## I.2) Pair-à-pair

- Pas de serveur permanent (ou juste pour le point de rendez-vous)
- Les participants sont alternativement demandeurs (clients) et fournisseurs (serveurs)
- On peut privilégier les services locaux, pour mieux passer à l'échelle
- C'est plus compliqué à faire marcher. Il faut des preuves et du beau code
- C'est mal vu légalement alors que techniquement c'est la meilleure approche. Ce sont les usages qui sont illégaux, pas l'approche[2]
- Il faudrait 10 heures juste sur ce sujet, pour bien faire.

## I.3) Push et Pull

- Deux modes d'interaction (par exemple pour les données des sondes de température sur l'IoT)
- Push: celui qui a la donnée l'envoi dès qu'elle est prête
- Pull: celui qui a besoin de la donnée la demande quand y'en a besoin

# II) Protocoles notables

## II.1) Le protocole mail (port 25)

- on regarde un premier protocole, très simple, pour voir comment on peut s'y prendre pour organiser les échanges
- Premier mail: 1965; programme UNIX mail: 1972. C'est l'un des plus vieux protocoles encore utilisés

### II.1.a) Composants du système

- User Agent: interface des humains
  - clients pour lire et écrire des mails
  - parlent à un serveur pour poster et lire sa boîte
- Mail Server
  - Une mailbox par utilisateur, qu'il peut relever quand il veut (Pull)
  - Une file (message queue) de messages en cours d'envoi
  - Connexions directes entre serveurs rares : mail relay et forwarders
- Le protocole SMTP: pour envoyer des mails (UA -> serveur, ou serveur -> serveur – Push)
- Plusieurs protocoles pour relever sa boîte (besoin de standardisation moindre): POP3 (disparu) ou IMAP (bel algo distribué pour synchroniser)

### II.1.b) Communications avec SMTP

- Toutes les requêtes du client on la forme: `motclé` (espace) (blabla pour humains ignoré) `\n`
  - Toutes les réponses ont la forme: `numéro` (info de debug pour humains) `\n`
- Exemple de dialogue SMTP ayant lieu sur le port 25:
  - C: `HELO admin@ens-rennes.fr`
  - S: `250 welcome`

  - C: `MAIL FROM dark@vador.net`
  - S: `250 ok`
  - C: `RCPTTO skywalker@rebelion.net`
  - S: `250 ok`
  - C: `DATA` (puis pleins de data, le contenu du mail, terminés par `.\n`)
  - S: `250 ok`
  - C: `QUIT`
  - S: `221 goodbye`
- Pour que le mail puisse être transferé au serveur suivant, il y a des entêtes au début des données.
  - Ca dit où va le mail, d'où il vient, et par où il est passé.
  - Les entêtes suivent la RFC 822 dans leur format, et sont séparés du corps du mail par une ligne vide (micro-protocole encapsulé dans SMTP)
  - Le bloc de texte du mail encapsule lui-même le protocole MIME (protection d'éventuels `.\n` sur une ligne dans le contenu pour pas mettre fin au transfert SMTP), multi-parties pour les attachements et spécification de l'encodage (le Subject est dans les entêtes donc il n'est pas protégé par MIME, donc il casse souvent).
- La forme générale du protocole est très intéressante: texte pur, intelligible sur la ligne
  - Les anciens n'ont pas cherché à faire efficace (malgré les capacités très limités des ordis à l'époque)
  - Ils ont privilégié maintenabilité et interopérabilité par rapport à performance.
  - De nombreuses idées ont été reprises dans d'autres protocoles d'internet

## II.2) Protocole FTP (port 21)

- un protocole maintenant disparu pour télécharger sur Internet au 20ieme siècle.
- Les commandes sont aussi en ASCII, sans info debug ignorée: USER ... / PASS ... / LIST / RETR fichier / STOR fichier
- Les réponses sont aussi numériques et du même genre: 4xx error / 3xx ok
- Au lieu de devoir protéger le marqueur de fin d'envoi dans le flux (comme `.\n` des mails), on ouvre une socket pour chaque fichier
- C'est peut-être ce qui explique sa disparition: c'est plus compliqué et mène à des performances moindres

## II.3) Le web (protocole HTTP, port 80)

### II.3.a) Histoire

- L'idée d'hypertexte date de 1945.
  - Article de Vannevar Bush (responsable de la recherche US en 40-44) intitulé "As we may think"
  - Présente graal d'échange d'informations et de collaboration entre scientifiques.
  - L'idée est de chercher à prolonger la très bonne collaboration entre scientifiques que la guerre a rendu nécessaire (radar, cybernétique, projet Manhattan).
  - Le moyen proposé (Memex) est aussi inspiré de l'encyclopédie de Diderot du 18ieme: du texte avec des liens entre
- En 1990, un physicien du CERN implémente HTTP pour l'échange de données au CERN.
  - Tim Berners-Lee (britanique)

### II.3.b) Composants

- Contenus (statiques ou générés): HTML (HyperText Markup Language): du texte pur avec des balises explicites. C'est faisable à la main, si on vise pas qqch compliqué.
- Clients et serveurs (HTTP, port 80), organisation Pull même si HTTP permet aussi le dépot de fichiers
- Proxies (explicites ou transparents): caches, logs, filtrage, anonymisation
- URL: Universal Resource Locator
  - proto:://[name@]host[:port]/chemin/ressource?key=val&k=v#tag

### II.3.c) Le protocole HTTP

- Protocole stateless: communication entre client et serveur fermée entre chaque requête (normalement).
  - Les cookies contenant l'état sont stockés coté client par le serveur
  - FTP était stateful, le dialogue était interactif entre client et serveur, ce qui complique le serveur puisqu'il doit stocker l'état du client
- Sinon, c'est encore un protocole ASCII comme FTP:
  - requetes `KEYWORD param param`
  - réponses: numériques + info debug. 1xx: info; 2xx: succès; 3xx: redirection (302: found elsewhere); 4xx: client error (404); 5xx: server error (503 unavailable)
- Exemple requête: GET / HTTP/1.0
- Réponse possible: `HTTP 1.1 200 OK`

  `Date: ...`

  `Server: ...`

  `Content-type: text/html`

  `Content-length: 342`

  (ligne vide)

  `<html><head><title>Blah</title></head><body>Bidule</body></html>`
- Pas de marqueur de fin de fichier à protéger, ni de socket séparée à ouvrir: on donne la taille dans les entêtes
- Version HTTP: c'est une trop bonne idée de versionner le protocole pour permettre les évolutions futures (et la négo entre client et serveur)
  - HTTP1: une connexion par objet (donc bcp de connexions)
  - HTTP2: passage de plusieurs objets dans le même message (page + CSS + image), voire dialogue interactif (AJAX). Connexions multiples //

### II.3.d) Le reste du web

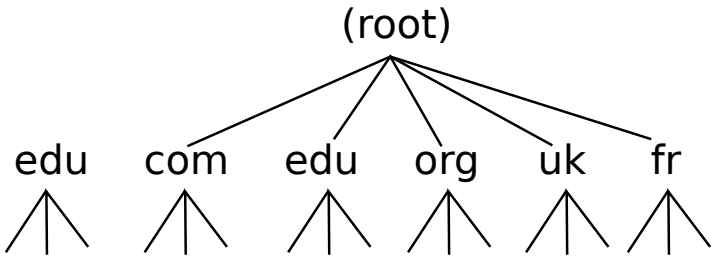
- Sécurité = HTTPS. Chiffrement de bout en bout, mais handshake initiaux garantis par une autorité centrale. Possiblement troué.
- Performance = caches, avec entêtes d'invalidation de cache pour contenu dynamique. Shift-F5 pour forcer un full refresh
  - Les caches doivent être bien placés pour couvrir des clients aux goûts similaires
- Content Delivery Networks (CDN): des caches proactifs, opérés par des entreprises spécialisées (Akamai)
  - Fournisseurs de contenu poussent les données populaires

- Le CDN les dupplique dans des serveurs au plus près des clients (dans le datacenter de l'ISP voire plus près encore)
- Les URL d'Akamai se résolvent vers le proxy le plus proche
- Netflix est son propre CDN avec ses films. Ils ont des ordinateurs chez l'ISP
- Les ISP ont tout intérêt à jouer le jeu:
  * pression des clients qui ont de meilleurs perfs si les données sont proches
  * moins d'échanges inter-ISP = factures auprès des autres moindre
  * les ordis du CDN sont payés et opérés par le CDN, pas par l'ISP

## II.4) DNS (port 53)

- Les humains préfèrent les lettres, les ordis ont besoin de l'IP numérique (voire hexadécimale pour IPv6) pour router
- A l'origine, correspondance dans un seul fichier de config centralisé au niveau d'Internet
  - mail à l'admin pour modifier ; download réguliers
  - Clairement, ça scale pas à l'échelle d'IoT, et ça constitue un single point of failure
- Cahier des charges de DNS: Internet-scale database
  - scalable (bcp de lecture, peu d'écriture)
  - contrôle distribué
  - tolérance aux pannes et un peu aux misconfigurations
- C'est impossible d'après le théorème CAP (Consistency/Availability/Partition-free: pick 2)
  - mais on relache la consistance (comme souvent sur internet: best effort). Eventually consistent
- Énormément de caches, puisque c'est surtout en lecture

### II.4.a) Base de données hiérarchique et distribuée




- Chaque zone s'administre séparément
- Chaque zone est servie par de nombreux serveurs répartis géographiquement (au pire, usage du multicast IP pour les trouver = on crie sur le tuyau sans savoir à qui on parle)
- Algo de résolution récursif:
  - 1: l'appli demande au DNS local (dans le même ordi). Si l'IP est connue, on arrête de chercher et on l'utilise
  - 2: DNS local cherche l'IP du root domain s'il ne l'a pas
  - 3: DNS local cherche l'IP du TLD (Top Level Domain – .fr) en charge. Demande à root, sauf si c'est déjà en cache
  - 4: DNS local demande l'authoritative DNS de l'AS ciblé (sauf si c'est déjà en cache)
- Mise en cache agressive: 99.9% des requêtes sont déjà en cache à un niveau ou un autre
  - Les entrées ont des TTL pour permettre les MAJ au bout d'un moment (consistance à terme)

- Scalabilité des serveurs root
  - 13 serveurs répartis partout dans le monde: 11 aux US, 1 japon, 1 allemagne. Ben oui, monde entier :)
  - Duppliqués par multicast IP (on crie dans les tuyaux sans savoir à qui on parle)

### II.4.b) Sécurité DNS: risque de poisonning

- Si ça marche, les requêtes sont détournées vers mon serveur (Man in the middle)
- Les serveurs peuvent répondre plus d'entrées que demandées (pour chauffer les caches)
- Pas d'authentification des réponses: possibilité de forger
- Attaque 1: je cherche à provoquer une requête DNS à laquelle je suis le seul à répondre (SMTP: `HELO www.evil.com`), et j'ajoute des entrées pour google et autres sites rigolos
- Attaque 2: je sniff et je répond plus vite que le serveur disant