Introduction aux réseaux

Martin Quinson

December 12, 2023

Présentation du module SYS1

- Ce poly correspond à la seconde moitié du module SYS1 de l'ENS Rennes. Après le cours sur le C, voici venu le temps du réseau. les systèmes informatiques.
- Les cours de C sont terminés. Maintenant, on utilise le C mais on a dit tout ce qu'on voulait dire
- Sur la partie réseau, l'objectif est d'obtenir la culture générale nécessaire à n'importe quel informaticien qui se respecte.
  - C'est au programme de l'agreg d'info, donc c'est important (hihi)
  - On ne va pas apprendre à cabler la maison, ni même à configurer un réseau déjà cablé
  - On veut comprendre le monde pour savoir quels sont les problèmes intéressants à résoudre pour les futurs chercheurs que vous êtes.
- Un peu de bibliographie : les livres sont utiles pour accompagner ce cours
  - Le cours est dans un ordre qui, je l'espère, simplifie la compréhension. C'est plus pédagogique.
  - Les livres sont un ordre mieux rangé, plus logique. Mais ça demande de savoir ce qu'il y a dedans pour le lire (ou de lire plusieurs fois).
  - Il y a des liens sur la page web, et les livres suivants sont à la bibliothèque
- Sur le C:
  - Braquelaire: un bon livre pédagogique. Pour être honnête j'ai appris avec, y'a bien longtemps
  - Kernighan et Ritchie: un guide de référence, qui explique (mal) le C. A proscrire pour les apprenants. "man 3 printf" plus utile que le K&R, au final.
  - Nebra: un bon livre pédagogique, et moderne (cours en ligne associé sur OpenClassrooom)
- Sur le réseau:
  - Pujol: 1500 pages en bottom up. C'est la bible francophone, mais ça reste un peu indigeste
  - Kurose: un bon livre top-down. C'est ce que je suis pour ce cours
  - Bonaventure: un livre top-down très honnête, et intégralement disponible en ligne.
- Sur les deux à la fois, et sur le système (qui constitue la suite logique)
  - Computer Systems: A programmer perspective, par Bryant & O'Hallaron. Carnegie Mellon.

- Motivation pour l'étude des réseaux
  - La première motivation pour étudier le réseau, c'est de permettre à ses programmes de communiquer.
  - Mais le design de l'internet que nous avons est également intéressant, en tant que système bien pensé
    * Peu d'évolutions techniques, protocolaires, mais nb machines d'un millier en 1984 à des dizaines de milliards maintenant
    * Qu'est ce qui a rendu ce design si robuste ? Quelles étaient les alternatives ?
    * Répondre à ces questions sont l'autre motivation d'étudier le réseau, celle qui pousse à regarder dans la boite
  - 3ieme motivation: L'informatique distribuée a longtemps été la dernière frontière (en date) de l'informatique
    * Nos Euclide s'appelle Vint Cerf et Leslie Lamport, et ils ne sont même pas à la retraite
    * Il reste beaucoup de low hanging fruits en recherche, pour qui voudra bien s'en saisir
- Programme de l'agreg en réseau
  - Caractéristiques des réseaux et performances associées :
    * réseaux d'accès, réseaux de coeur ;
    * topologies de réseaux : point à point, à diffusion ;
    * performances : débit de transmission, délai, taux de perte.
  - Modélisation en couches : TCP/IP, encapsulation.
  - Transmission :
    * Adressage : adresses MAC, IPv4, IPv6 ;
    * Routage : principes ; routage à vecteur de distance (algorithme de Bellman-Ford), routage par état de liens (algorithme de Dijkstra) ;
    * Solutions de transport : principes ; TCP, UDP.
  - Programmation réseau : API Sockets en Python et en C à l'aide d'un aide-mémoire fourni.
  - Leçon 2022:
    * 21. Échanges de données et routage. Exemples.

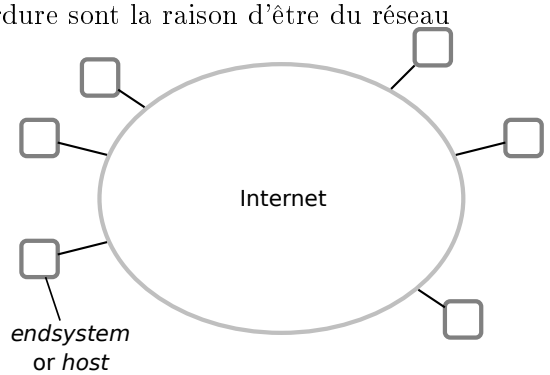
Chapter 1

Un réseau de réseaux

- En matière d'internet, il y a les choses qu'on ne pouvait pas faire autrement, et les choses qui sont le résultat de choix
- On va commencer par voir les aspects nécessaires d'un réseau de réseaux

I) Constitution du réseau

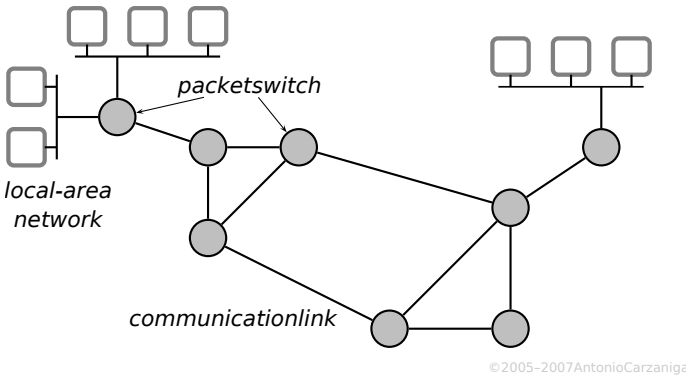
- Qu'est ce qu'Internet?
- Les éléments en bordure sont la raison d'être du réseau



I.1) Terminaux

- Ce sont les choses connectés à Internet, en bordure, pour profiter du service de connexion
- Ordinateurs et maintenant smartphones. Mais aussi objets connectés (frigo)
- Il parait que l'avenir de l'humanité passe par l'IoT, qui demande des milliards de sondes et des centaines de millions d'actuateurs
  - Borne température et chaudière
  - Panneau et feux de circulation, Sondes sur la voie TGV, surveillance réseau élec

I.2) Réseaux informatiques



- ne pas parler de contenu du réseau, car "contenu" désigne habituellement les données
- Équipements réseau: routeurs = switch (bien différents si on est au coeur ou en bordure)
- Liens entre tout ça
  - cuivre: permet de reprendre le réseau téléphone, qui est une paire cuivre (+: existance infra; -: perte en ligne)
    - A l'origine, réseau commuté (voix ou données, mais pas en même temps). ADSL= partage de fréquence grâce à des modems de chaque coté
    - Le dernier kilomètre est un réseau en étoile autour du DSLAM
  - fibre optique: infra nouvelle, mieux adaptée, mais couteuse à déployer.
    - Donc branchement en série sur une fibre, et multiplexage fréquence (une couleur par maison)
    - La SNCF et les sociétés d'autoroute sont des opérateurs d'internet: c'est eux qui posent les fibres opérées par d'autres
  - Sans fil: bcp techno.
    - wifi: 802.11; Bluetooth
    - GSM (techno téléphone). Avec les générations, portée baisse, débit monte. 5G pas raisonnable du tout. On a pas d'usage (sauf netflix 4k dans l'ascenseur) et ça coute une énergie de dingue.
    - Lora: débit assez faible, portée en kilomètres

I.3) Topologies

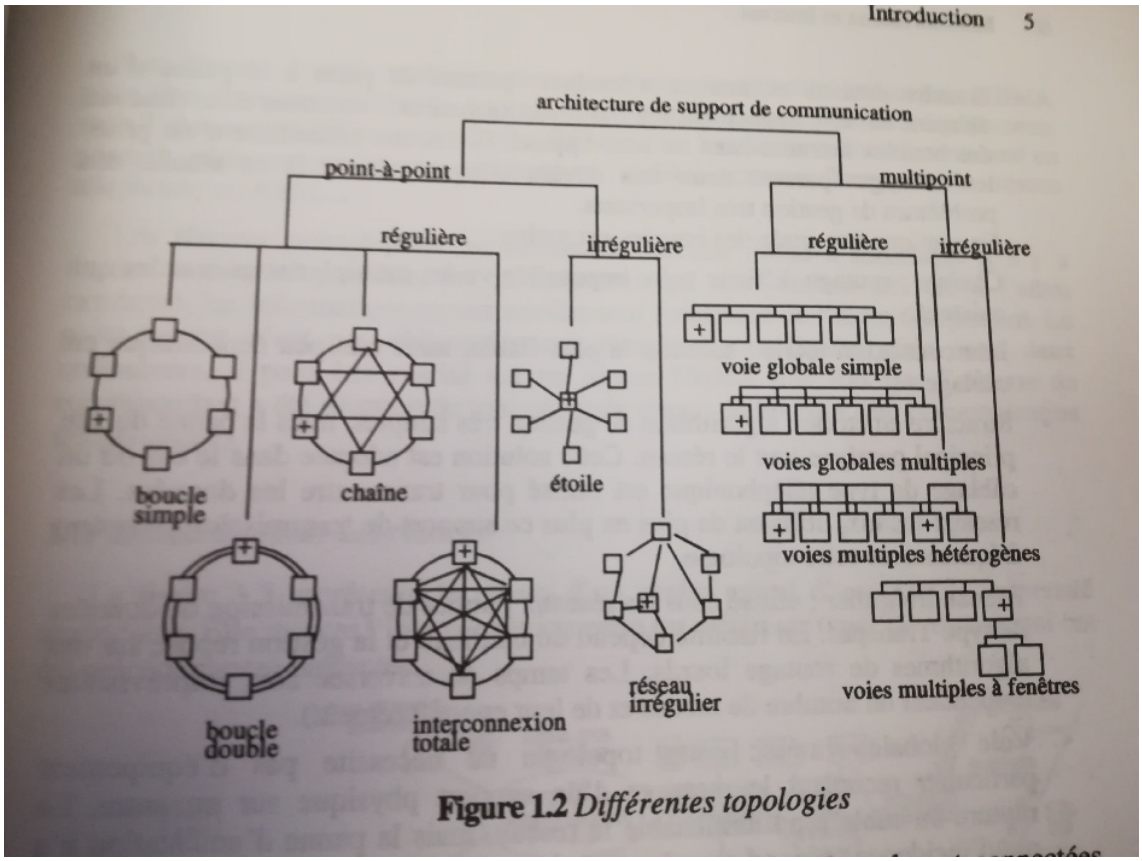


Figure 1.2 *Différentes topologies*

- On pourrait ajouter les dragonfly et autres joyeusetés

II) Partage de ressources

- Qu'est ce qui se passe quand on envoie un film au bout du monde ? Est-ce que ça voyage en un seul gros bloc ou en petits paquets? Est-ce qu'on réserve une place sur le réseau ?

II.1) Commutation de circuit

- Mot compliqué pour parler de continuité cuivre: le réseau est réservé, alloué pour le flux
- C'est l'organisation du réseau téléphonique à l'origine (avant sa numérisation). X.25 pré-allouait des ressources lors de la connexion (circuit virtuel, plus bas)
- Ca permet des garanties fortes de performance, mais c'est un gaspillage de ressource
  - Pour permettre à N personnes de parler entre Rennes et Paris, faut N paires de cuivres séparées, mais les fils sont inutilisés quand y'a un blanc dans la conversation entre les interlocuteurs.

- Donc non, le vrai Internet n'est pas si dispendieux. On a la ligne que le temps où on l'utilise vraiment, sans pré-réservation
  - Les bandes passantes sur Transpac étaient garanties, et incroyablement faibles

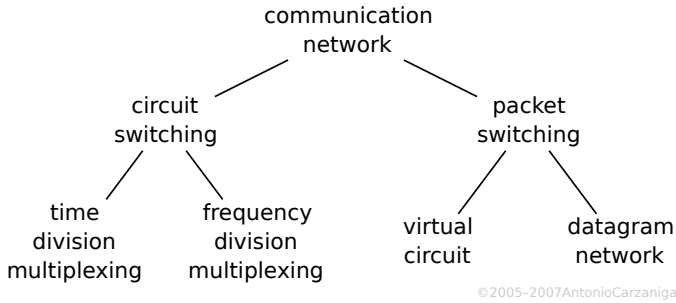
II.2) Commutation de paquets

- Sur Internet, les données sont découpées en petit paquets envoyés les uns derrière les autres et traités séparément par le réseau
  - Pas de coût de connection, meilleur partage des liens
  - Mais pas de garantie de performance, surcoût de traitement de chaque paquet, entêtes plus long
- Faire autrement s'appelle "Store and Forward": Youtube stocke la vidéo sur le premier routeur après, qui l'envoi au routeur suivant.
  - Belle analogie du camion, mais ça marche pas comme ça, pour des raisons de performances sur lesquelles on revient bientôt

II.3) Circuits virtuels

- L'idée est de combiner le meilleur des deux mondes.
- Etablissement d'un circuit virtuel = attribution d'un identifiant de communication
- Les paquets ensuite routés en utilisant cet identifiant, ce qui est plus efficace en temps de traitement et en taille d'entête

II.4) Taxonomie des réseaux



III) Protocoles

- Le mot "protocole" existait bien avant les ordis.
  - Le protocole, c'est qui a le droit de parler à la reine et quand. Une réponse fort peu protocolaire.
- C'est souvent culturel. Ne pas respecter le protocole $\rightsquigarrow$ incompréhensions et fait passer pour un malpoli
  - En France, quand on décroche le tél (en informatique: handshake), on a:
    - Répondant: Allo?
    - Appellant: Oui bonjour, c'est toto
    - Répondant: Ah bonjour toto, c'est bidule. Qu'est ce qui t'ammène?

  - En Allemagne, c'est un peu différent:
    * Répondant: Bidule.
    * Appellant: Oui bonjour Bidule, c'est Toto
    * Répondant: Ah bonjour Toto. Qu'est ce qui t'ammène?
- En informatique, les protocoles doivent définir le format et l'ordre des message
  - Ça se spécifie souvent avec plusieurs machines à états synchronisées
    * Ca donne des recherches amusantes, où les collègues de Nancy faisaient de l'apprentissage sur la machine à état du téléphone IP, puis attaquaient aux bordures inhabituelles du protocole. Ils ont découvert une série de message étranges à envoyer au téléphone pour décrocher et écouter, sans que le téléphone ne signale quoi que ce soit en facade... Ca fait un "bug" bien pratique et un peu complexe pour que ce soit dû au hasard.
  - traitement des erreurs de transmission, d'accès concurrent au média, timeout/retransmission
  - Un protocole est une sorte de programme distribué
- Autre exemple: le protocole du contrôle aérien
  - Extrêmement rigoureux.
  - Partage du temps de parole (la tour donne la parole), pas d'ambiguité et ACK, gestion des problèmes de transmission et retransmission
  - Multi-partite et sans connexion
- Sur Internet, il faut en plus se mettre d'accord sur leur specs
  - Internet Engineering Task Force (IETF): organisation d'orga et individus qui discutent et spécifient. C'est très américain.
  - Request For Comments: type de doc qui spécifie un protocole en particulier

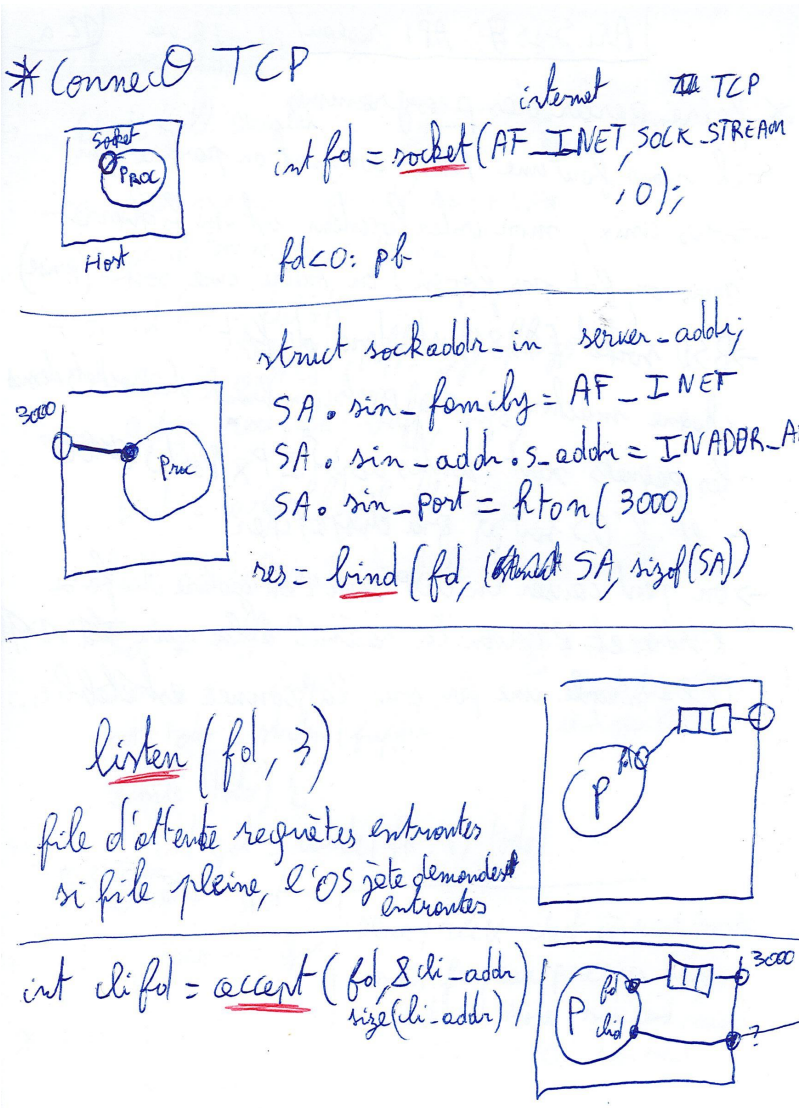
Chapter 2

Faire communiquer des programmes

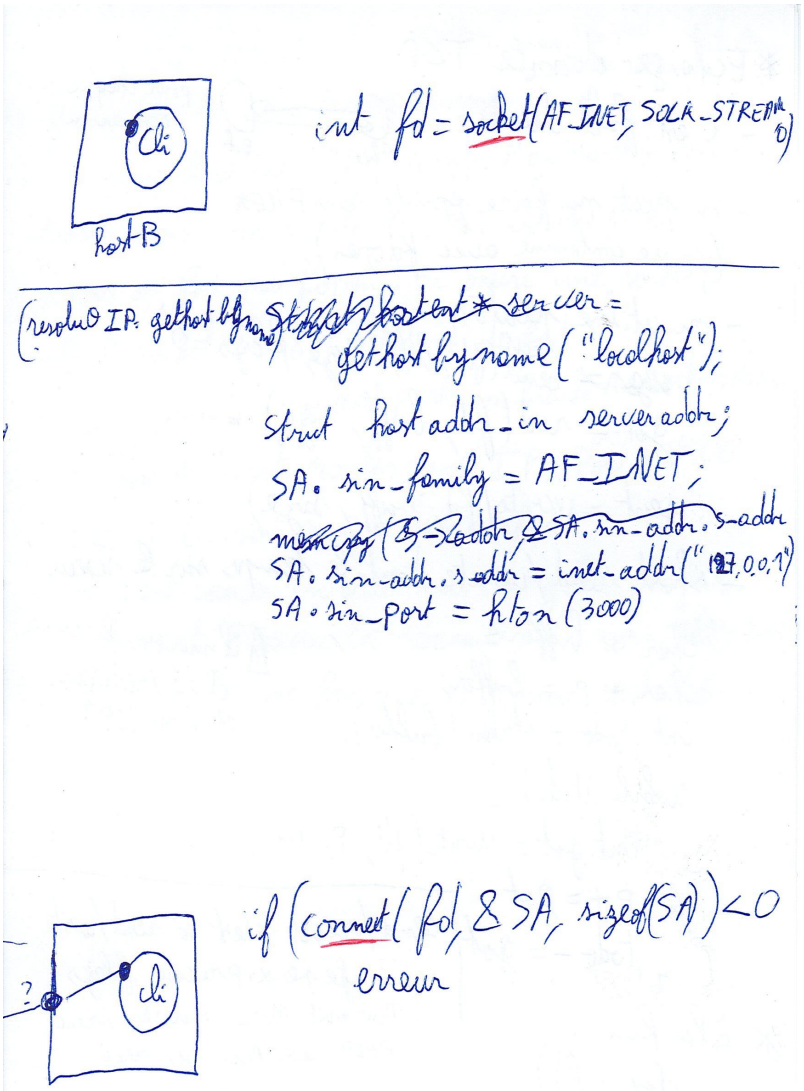
- Pour faire communiquer des programmes informatiques, il nous faut une API: comment on parle à qui.
- L'interface standard de facto, de l'internet que nous avons, c'est **BSD socket**
  - Cette interface date de 1983 (BSD 4.2), c'est à dire qu'UNIX avait déjà 13 ans quand ça a été introduit.
    - Certaines parties mal intégrées (fnctl), et UNIX n'est pas le design le mieux adapté au réseau.
    - Plan 9, par les mêmes, était mieux pensé mais ça n'a jamais pris car UNIX reste good enough dans son design, et ses implems sont très solides.
  - *Socket*, en anglais, ça veut pas dire chaussette, mais prise femelle. Une prise male, c'est *plug*
- En TP et partiel, on utilise le protocole TCP, où il faut se connecter pour avoir un flux bi-directionnel, mais qui permet d'utiliser le réseau sans réfléchir ensuite
  - Il y a une autre façon d'utiliser l'Internet moderne (protocole UDP) mais pas de mise en pratique cette année
- Sous UNIX, mon interlocuteur est un fichier. Ou plutôt, je le manipule comme un fichier. C'est à dire que `fprintf` marche, que mon interlocuteur soit:
  - un clavier/écran (par défaut)
  - un vrai fichier (redirection d'E/S dans le shell)
  - un autre programme en local (tube)
  - un autre programme à distance (socket réseau)
  - Si je représente mon processus comme un ovale, une socket est un rond en surface sur lequel un autre processus peut venir se connecter
- Sur internet, la destination des paquets d'information est donné par un couple {IP, port}
  - L'IP désigne l'ordinateur destination
  - le port désigne une sorte de boite-aux lettres à laquelle les paquets d'information doivent être délivrés. L'idée est qu'il y a un programme donné derrière une boite aux lettres donnée.

- Si je représente mon ordi comme un carré contenant des ovales-processus, un port est un rond en surface du carré:
  * ouvert vers l'extérieur pour permettre à des processus distants de se connecter dessus
  * connecté dedans à une socket d'un processus donné

I) Connexion TCP



1. Le serveur


Cli
hostB
int fd = socket(AF_INET, SOCK_STREAM, 0)
(resolution IP: gethostbyname)
gethostbyname("localhost");
struct hostaddr_in serveraddr;
SA.sin_family = AF_INET;
SA.sin_addr.s_addr
SA.sin_addr.s_addr = inet_addr("127.0.0.1")
SA.sin_port = htons(3000)
cli
if (connect(fd, &SA, sizeof(SA))<0
erreur

2. Le client

II) Echange de données TCP

* Echange donnée TCP
- C'est full duplex : 

pour chaque connexion
- on peut pas faire fprintf sans FILE* (ça se contourne avec fdopen)
- on utilise plutôt
  recv(fd, buff, size, flags=0)
  ~~size~~ = send(fd, buff, size, flags=0)
  got = read(fd, buff, size)
  sent = write(fd, buff, size)
- short read / write tout le temps sur le réseau

```
char * buffer = "....";
char * p = buffer;
int todo = strlen(buffer);
while (todo) {
    int got = write(fd, p, todo)
    p += got
    todo -= got
}
```

⚠ il manque le code client/serveur, et un peu P2P ?

* à la fin
  close(fd)

send/recv c'est co read/write
mais je peux preciser flags
Dont wait, More sur send/recv
PEER, WAITALL sur recv

Chapter 3

Performance du réseau

- Intuition: de l'eau dans les tuyaux, ou les voitures sur la route

I) Quantifier la performance

- Qu'est ce qui fait "une bonne co" ? Ou le contraire?
- Au niveau applicatif:
  - t0: premier bit s'envole sur le réseau; t1: premier bit arrive à destination; t2: dernier bit arrive à destination
  - Latence (en ms): temps de transmission d'un seul octet (t0-t1)
  - bande passante (en bit/sec): vitesse de transmission: taille / (t2-t1)
  - temps total: lat + taille / bw
- Au niveau transport
  - on a besoin de la gigue (jitter – variance de la latence) car TCP a besoin de deviner si le paquet est perdu ou pas
  - on a aussi le taux de perte pour UDP
- au niveau physique: taux d'atténuation (en dB)
- 1kib = 1024 bits; 1kB = 1000 octets.
- b minuscule: bits; B majuscules: bytes, octets
- kGPE: préfixes habituelles en puissance de 10; ki Gi Pi: puissances de 2 les plus proches

II) Performances de la commutation de paquets

- Imaginons un chemin de longueur 2
  - Alice — routeur A — routeur B — Bob
  - chaque lien 2Mb/s, sans latence.
  - Temps d'envoi d'un fichier de 8 Mbit: Alice->A (4 sec) + A->B (4sec) + B->Bob (4sec) = 12 sec.
- Si on coupe en 1000 paquets de 2kbit
  - `| Alice->A |  A -> B  | B->Bob`

- `t=0   | Paquet 1 |     -    |    -`
- `t=1   | Paquet 2 | Paquet 1 |    -`
- `t=2   | Paquet 3 | Paquet 2 | Paquet 1`
- `t=3   | Paquet 4 | Paquet 3 | Paquet 2`
- ...
- `t=999 |  P1000   |   P999   |  P998`
- `t=1000|     -    |   P1000  |  P999`
- `t=1001|     -    |     -    |  P1000`
- Comme avant, le dernier paquet arrive en A après 4 secondes. Mais comme les autres paquets ont déjà utilisé les liens suivants sans attendre sur A, le dernier paquet arrive à Bob quelques milisecondes plus tard

III) Intuitions fausses sur la performance du réseau

III.1) La qualité du réseau n'est pas un ordre total

- `débit | latence`
- `gros  | faible`: (fibre dans la ville)
- `faible| gros` : (56k de ma jeunesse, internet de ta grand mère)
- `gros  | gros` : (fibre transcontinentale)
- `faible| faible`: (bus CAN embarqué dans la voiture, temps réel pour les commandes)

III.2) Corrélation débit / latence

- Monter le débit peut augmenter la latence
- Ex1: régulation de vitesse sur l'autoroute
- Ex2: le RER A. 2x plus de rame -> elles doivent s'arrêter entre chaque gare -> latence augmentée

III.3) Internet est très rapide

- Un chien à 18km/h avec une clé USB de 21Go va aussi vite qu'un lien à 150Mbit/s
- Les données des téléscopes sont descendus à dos de lama des montagnes chiliennes

Chapter 4

Internet

I) Histoire partielle d'Internet

- Époque des précurseurs (avant 1973)
  - 29 octobre 1969: Arpanet est lancé aux US (programme militaire)
  - 1972: Arpanet a 15 noeuds; échange du premier email
  - D'autres réseaux existaient (Merit, NPL), mais ils étaient incompatibles.
    - Le réseau CYCLADES de Louis Pouzin (Inria) est le premier avec la compatibilité inter-réseau en tête (design 1972, demo 73).
    - Arrêté en 1976 car les PTT n'aimaient pas la concurrence... (ils poussaient Transpac, l'implem locale de X.25 ayant permis le minitel)
  - Arpanet version 1973 intègre les idées de CYCLADES
- Fusion des réseaux et création de l'Internet (1973-1983)
  - Les réseaux sont là et la question est de comment unifier leurs forces
  - 10 ans de recherche (Vint Cerf et Bob Kahn prix Turing 2004)
  - 1983: Protocole TCP/IP et sockets BSD pour s'y brancher, qui fait encore tourner internet
- Essort d'Internet (depuis 83)
  - 1989: création du web, HTTP
  - 1992: Ouverture au public pour un usage commercial (ce qui est à l'état américain doit profiter à ses citoyens, impact mondial)
- Guerre des protocoles (fin 80, début 90)
  - Depuis les années 70, les PTT de nombreux pays poussent le protocole X.25 pour interconnecter les ordinateurs (travaux de Rémi Després)
  - En 1984, il y avait 1000 ordis sur internet, et 24000 sur Transpac
  - Fin des années 80, le monde est polarisé entre le monde TCP/IP (les USA quasi seuls) et le monde X.25 (surtout en Europe)
  - Le CERN est le premier centre européen connecté au réseau TCP/IP
  - X.25 a perdu car (perçu comme) trop bureaucratique, pas assez pragmatique. Il était plus rentable pour les opérateurs

  - Le service de Minitel a fermé en 2012, il restait 800k terminaux en fonctionnement, après 9M au pic, en 2003.

II) Forme générale d'Internet

- (on s'appuie sur le document du Kurose, à imprimer, ou des patates interconnectés)
- C'est un réseau de réseaux interconnectés
  - Cherche à interconnecter les réseaux existants
  - Il fallait survivre aux attaques nucléaires (donc sans single point of failure).
  - Et puis c'est pratique à la fin. Ca permet l'auto-organisation, et la mise en émulation d'entités concurrentes.
  - Remarquez qu'Internet est une construction politique avant d'être une construction technique. Internet inventé par les français aurait été bien différent : centralisation, et passage obligé par Paris.
  - On parle d'AS: autonomous system. Entité auto-organisée (entreprise pour son usage, ou fournisseur contenu, ou ISP/FAI entre)
  - Les ISP sont interconnectés en mesh, avec des points d'échange. Carte des ISP != géographie, car Bouygues et Free couvrent le même territoire, avec multiples point d'échanges entre eux, à différents points de la géographie.
- Les routes d'internet sont multi-hop, et potentiellement multi-technologie

III) Types de services

- Les besoins des applications sont divers, et incompatibles:
  - Intégrité des données: mail, téléchargement, web
  - Performances garanties (débit ou temps de réponse): contrôle robot à distance?
  - Certaines applications sont adaptatives : en VoIP ou videoconf, les paquets sont obsolètes au bout d'un temps. On a une contrainte temps réel, mais faible: si le paquet est vieux, on le jette et on privilégie les nouveaux. Les protocoles se resynchronisent (et négocient la qualité vidéo).
- Internet offre deux types de services de bout en bout car impossible de servir tous les besoins
  - "best effort" sans connexion (UDP)
    * On envoie des datagrammes, comme on envoie des lettres à la poste
    * best effort veut dire "non fiable" sans être mal intentionné
      · Similaire aux SMS, qu'on peut recevoir de nombreuses fois dans le métro
    * UDP ne fait rien et permet de faire exactement ce qu'on veut
    * Les applications peuvent implémenter exactement ce qu'elles veulent, et potentiellement plusieurs modèles de garantie de service.
    * Dans les jeux, il y a de la voix (best effort avec invalidation des vieux messages), des messages pour les affichages (priorité basse) et des messages de contrôle (priorité haute)

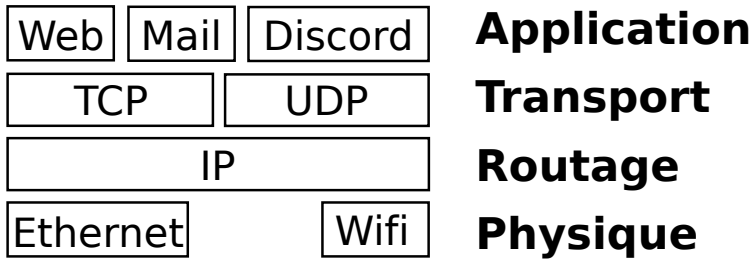
  - connecté, pour un flux fiable et efficace (TCP)
    * On établie une connexion logique (vue uniquement par les interlocuteurs), utilisée comme le téléphone
      · orienté connexion: on ouvre des sockets et on écrit dans le file descriptor
    * Transmission fiable et efficace
      · flux: envois de paquets transparents (après bufferisation)
      · fiable: réémission, gestion des dupplicats, ordre des paquets
      · contrôle de flot: l'émetteur freine pour pas saturer le récepteur
      · congestion: la communication freine si le réseau patine
- Internet n'offre pas garantie de performance (ni bande passante ni surtout en latence)
- Modèle de sécurité d'Internet:
  - par défaut, il n'y en a pas.
    * Initialement, projet militaire $\rightsquigarrow$ on place un GI pour sécuriser l'accès au matériel
    * Ensuite, projet californien un peu flower power
    * C'était bien différent sur X.25, qui utilisait la ligne téléphonique après établissement du circuit
  - Donc, il y a énormément de problèmes potentiels: Virus, Ver, BotNet
  - Pour des charges utiles variées: DDos, sniffing, spoofing, phishing
  - White hat vs. Black hat
  - On a déjà parlé de sécurité informatique dans le chapitre sur les fichiers (bluetoff, LPM, inversion de charge de preuve d'hadopi, et agravation anti-terroriste de la LPM). Ces choses s'appliquent aussi au réseau, bien sûr

IV) Modèle en couches

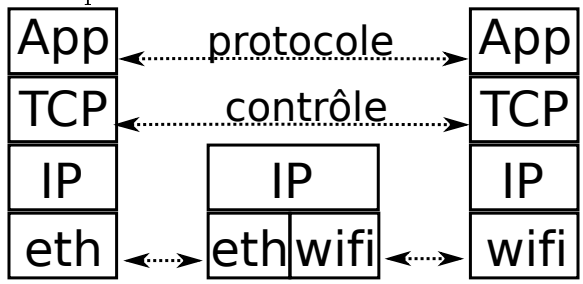
- Le découpage entre les applis et les protocoles TCP et UDP est bien pratique: les applis peuvent s'appuyer sur ces protocoles, qui font leur travail (et seulement ça).
- Ce genre de découpage est généralisé dans Internet, car cela permet d'organiser l'hétérogénéité énorme des réseaux.
  - L'idée était prototypée dans CYCLADES; la première version TCP/IP n'était pas découpée
- On aurait pu prendre un modèle rigide et standardisé (comme étaient les premiers protocoles réseaux), mais c'est error prone, et ça n'évolue pas. En plus, les standards un peu gros contiennent des bugs qu'il est impossible de corriger après coup.
- A la place, Internet utilise un modèle en couche:
  - On coupe le service en sous-services
  - On rigidifie les interfaces entre les couches
  - On laisse les alternatives venir à l'intérieur d'une couche donnée
- Avantages du modèle en couche d'internet:
  - Découper le problème est une bonne réponse à la complexité
  - Ce modèle donne du vocabulaire pour discuter entre architectes

– On peut faire évoluer le réseau par parties

IV.1) Modèle internet



- Couche Application
- Couche Transport: envoi de données de bout en bout. Deux protocoles majeurs.
  - TCP maintient un flux fiable et efficace
    - orienté connexion: on ouvre des sockets et on écrit dans le file descriptor
    - flux: envois de paquets transparents (après bufferisation)
    - fiable: réémission, gestion des dupplicats, ordre des paquets
    - contrôle de flot: l'émetteur freine pour pas saturer le récepteur
    - congestion: la communication freine si le réseau patine
    - mais pas de garantie de performance (best effort) ni de sécurité
  - UDP ne fait rien et permet de faire exactement ce qu'on veut
- Couche Routage: passage de réseau en réseau
- Couche Physique: échange de données sur un réseau donné
- Chaque couche parle à son interlocuteur en face, en passant par la couche du dessous, pour rendre service à la couche du dessus
  - Les routeurs au coeur du réseau (avec plusieurs interfaces réseaux) n'ont que jusqu'à la couche IP.
  - TCP et App ne sont que sur les bordures.



- En pratique, on encapsule les paquets, et les entêtes sont des préfixes

IV.2) Modèle OSI

- un modèle standardisé en 7 couches. Tout le monde l'utilise pour parler (c'est un trigramme)
- mais en pratique, c'est la pile TCP/IP en 4 couches qui a gagné (les protocoles OSI n'ont jamais décollé)
- Trop de couches = latence de traîtement et accumulation d'entêtes

Chapter 5

Couche Applications

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

1. 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)
2. 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)

1. 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)
2. 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

3. 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)

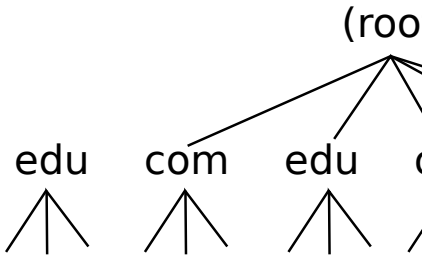
     `<html><head><title>Blah</title></head><body>Bidule</body></html>`
   - Pas de marqueur de fin de fichier à protéger, ni de socket séparée à ouvrir: on donne la taille dans les entêtes
   - Version HTTP: c'est une trop bonne idée de versionner le protocole pour permettre les évolutions futures (et la 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 //

4. 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



1. 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)

2. 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

Chapter 6

Couche Transport

I) Introduction à la couche transport

- Ce chapitre porte sur la couche Transport, c'est à dire l'API offerte aux applications pour le réseau
- On a déjà dit qu'il était impossible de servir tous les objectifs de toutes les applications avec une solution unique
  - Besoins contradictoires: fiabilité (mail, web, téléchargement: on veut rien perdre) vs. ponctualité (voix ou appel vidéo où les données de plus de 0.3 secondes ne servent plus à rien)
  - Impossible d'avoir toutes les qualités à la fois
- Internet propose donc deux protocoles par défaut : TCP (avec la fiabilité) UDP (en mode DIY)

I.1) UDP

- On contrôle l'envoi de paquets, qui est la réalité du réseau en dessous.
  - Chaque paquet est indépendant
  - $\oplus$ Envoyé à {Adresse IP $\times$ numéro de port}. A la réception, on écoute sur un port donné. Encore une fois, les ports sont une vue de l'esprit. C'est dans les entêtes du paquet que sont ces infos
  - $\ominus$ les paquets peuvent se perdre ou arriver dans le désordre (UDP est en DIY, TCP gère ce pb)
  - $\oplus$ chaque paquet est protégé contre les corruptions de comm par un checksum simpliste: ajout des octets sans retenue (détecte les 1-error mais pas toutes les 2-errors)

I.2) TCP

- C'est l'interface qu'on a utilisé en TP, y'a longtemps.
- On établi une liaison comme on décroche au téléphone, puis on pousse des octets dans le tube. Ils sont reçus, peu importe comment.
- TCP juste fait son travail:

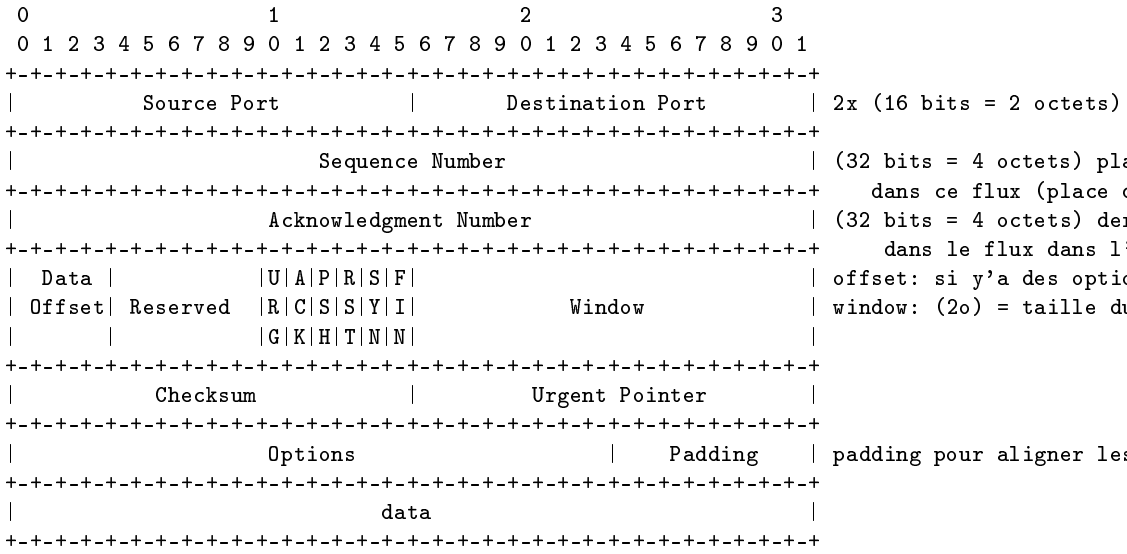
- gestion des paquets perdus, dupliqué (ce qui ne peut pas trop arriver en UDP), en désordre, très vieux (>120s)
- en plus du service aux utilisateurs, TCP cherche à optimiser l'usage réseau: remplissage sans saturation,
- Les pbs sont nombreux: aucune connaissance préalable du réseau; plusieurs machines, multi-hop, multi-path; conditions (RTT) très changeant

1. Historique de TCP
   - Inclu dans IP à l'origine, puis séparé ensuite (mais on parle encore de la pile TCP/IP)
     - 74: premières idées (3-ways handshake); 78: séparation TCP/IP ; 83: Arpanet passe à TCP/IP
   - RFC de 1981, 1989, 2009 (congestion). Multiples variantes successives (TCP Tahoe, TCP Reno, TCP Cubic) et intercompatibles.
   - Il y a encore de la recherche dans TCP (big latency-bandwidth product networks il y a peu), car le protocole ne fixe que ce qui doit l'être
     - Google/Facebook poussent un concurrent à TCP nommé QUIC. Sorte de TCP multi-sockets. Parfois appellé TCP/2 car ça va bien avec HTTP/2 qui fait aussi des envois en parallèle. Implémenté dans les navigateurs.
2. Design de TCP
   - Il faut voir ça comme un protocole distribué pratique, même si certains points semblent fixés au doigt mouillé, c'est plutôt du pragmatisme (=doigt mouillé résultant d'expériences pratiques)
   - Le parti pris d'origine est de faire un protocole bout-en-bout. = intelligence en bordure, ne supposant rien sur le réseau. Ca s'appelle le End-to-End argument, et c'est une réflexion poussée, pas un hasard.
     - De toute façon, on peut pas réfléchir lien par lien puisqu'on ne peut pas obtenir de vision globale du réseau. Donc bout en bout plus sage.
     - Chacun veut payer (en temps de traitement) pour ce qu'il utilise, pas plus.
   - Ce End-to-end argument pousse maintenant à la neutralité du net dont certains se font les ardents défenseurs.
     - D'autres disent qu'ils serait plus sage de faire de la différentiation de service, c'est à dire de laisser le postier ouvrir votre courier pour faire circuler plus vite les lettres d'amour que les factures.
     - Bien entendu, ce sont les postiers (mais pas seulement) qui militent pour la différentiation et les libertaires (mais pas seulement) qui militent pour la neutralité.
     - Certaines techniques anodines peuvent tomber sous le coup de la différentiation (filtrage en fonction de l'IP destinataire). D'autres sont beaucoup plus intrusives (Deep Packet Inspection). Est-ce la technique qui est mauvaise par défaut ou ses usages ?
     - L'europe est plutôt impliquée pour la neutralité, et les USA (sauf l'UCLA et l'EFF) plutôt prompts à envisager la différentiation

- Tout, dans les réseaux informatiques, est une question de choix politique. Et aussi dans les systèmes informatiques plus largement. De l'intérêt de comprendre ces choses.

3. Anatomie de TCP

- Il y a beaucoup de champs absconds dans un paquet TCP.
  - On peut faire un merveilleux cours de trigramme, avec la dizaine de flag binaires tels que URG (urgent), ACK, PSH (push to app asap), RST (reset), SYN (synchronize), FIN (finalize, raccroche).
  - On peut faire un cours de C parfaitement indigeste, avec car le format (la représentation mémoire) des 32 à 40 octets d'entête est spécifié au bit près.
    * On va faire le schéma suivant peu à peu, surtout pas dès le début.



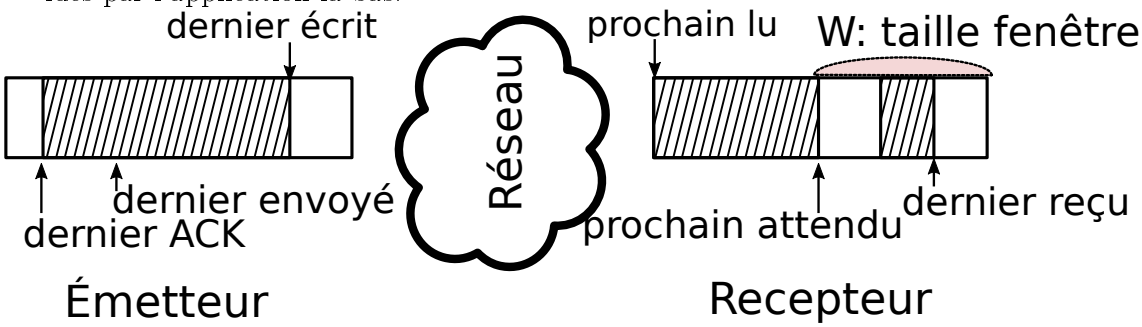
- On peut aussi faire de beaux diagrammes d'états pour l'établissement de la connexion, ou de gantt chart sur la fin d'une connexion.
- C'est magnifique, on peut parler pendant des heures sans que personne comprenne.
- Je vais plutôt chercher à expliquer **pourquoi** le protocole TCP est tel qu'il est, même si vous devrez aller chercher dans un cours classique comment ça marche en détail si ça vous intéresse.
  - On va détailler les entêtes, mais au fur et à mesure qu'on en a besoin dans notre re-construction du protocole.
- Le protocole TCP est découpé en 3 blocs logiques, que l'on va étudier à tour de rôle
  - Service de base: le fait qu'on puisse envoyer des octets après connexion, sans se prendre la tête
  - contrôle de flux: l'émetteur cherche à ne pas saturer le récepteur (qui peut dire où il en est)

– contrôle de congestion: la communication cherche à ne pas saturer le réseau intermédiaire (qui n'indique pas son état puisqu'il pourrait y avoir une meilleure route. On detecte les pbs sous forme de pertes de paquets)

II) Mécanismes de base de TCP

II.1) Buffers de communication

- Pour que le service fonctionne, il faut que chaque paquet contienne le numéro de port émetteur et destinataire (les adresses IP sont stockées dans les entêtes de cette couche là)
  - Ca prend déjà 4 octets sur les 32 d'entête. Les ports sont donc sur $[1; 2^{16}]$, ie 65534.
- Les données envoyés dans le tube sont mise en buffer coté emetteur jusqu'à ce qu'elles soient envoyées et reçues, puis dans un buffer coté recepteur jusqu'à ce qu'elles soient lues par l'application là-bas.



- Voici la situation des buffers dans une communication à un instant donné.
  - On a représenté que la moitié de la situation: c'est du full-duplex et donc le récepteur est en même temps récepteur de l'autre sens de comm.
  - On a représenté les buffers à plat, mais ce sont en fait des buffers circulaires: quand on arrive à la fin, on reprend du début.
- Coté émetteur:
  - Quand l'application utilise `write()`, on ajoute les octets après "dernier écrit", s'il reste de la place. Sinon, c'est bloquant pour l'appli.
  - Quand le recepteur confirme avoir reçu des données, on avance "dernier ACK", et le contenu du buffer à gauche de ça peut être écrasé.
- Coté récepteur:
  - Quand l'application utilise `read()`, on lui donne ce qui est entre "prochain lu" et "prochain attendu". Si y'a assez, sinon c'est bloquant.
  - Quand on reçoit des données de l'émetteur, il faut savoir où les mettre dans le flux (numéro d'ordre du paquet).
    * Numéro d'ordre = 4 octets d'entete supplémentaires. Position dans le buffer circulaire, donc y'a pas vraiment de max: retour à 0 après MAXINT.
    * Si ça fait avancer "prochain attendu" (ie, si ça grandi le bloc de données prêtes à être lues), il faut accuser réception à l'émetteur pour qu'il avance son "dernier ACK"

      - On ajoute 4 octets supplémentaires pour dire la position du pointeur dans le buffer émetteur
    - On pourrait vouloir dire qu'on a reçu un bloc au milieu, mais on a que 32 octets d'entête donc on peut pas. Faudra trouver des ruses.
- Questions à résoudre:
  - Quand envoyer un paquet avant remplissage ? si paquets mal remplis, c'est inefficace. si interaction bloquée sur 3 octets qui dorment, c'est inefficace.
  - Quand décider qu'un paquet s'est perdu (et le renvoyer) ? Si je renvois trop vite, je sature le réseau (et risque d'agraver le pb). Si j'attend trop, mauvaise interactivité.
  - Combien de données faire circuler sur le réseau au max ? Pour ne saturer ni le récepteur, ni le réseau. Sinon, perte de paquets et je devrais ré-envoyer.

II.2) Etablissement de connexion TCP

- Il y a un autre pb inattendu: Comme la connexion est seulement logique, si on "raccroche" puis refait une connexion juste derriere, il y a tjs un risque pour que des paquets du passé viennent planter le protocole.
- L'idée est simplement de tirer des nombres au hasard comme position dans les buffers circulaires. Ensuite, on ignore les paquets dont les valeurs ne sont pas crédibles. Ca serait bien le diable si des paquets du passé tombaient pile dans le protocole.
- Pour établir ces valeurs, on est obligé d'avoir un protocole en 3 voyages: 3 ways handshake
  - SYN: client envoie la position de son buffer d'émission (tiré aléatoire, donc). Sequence number n'a pas le sens habituel.
  - ACK+SYN: le serveur accuse réception de la position buffer du client, et annonce la sienne
  - ACK: le client accuse réception de la position buffer du serveur
- Ca demande des flags binaires pour modifier un peu la sémantique des champs, mais qu'à cela ne tienne. TCP en déclare 12, dont certains ne sont pas utilisés.
- La fin de connexion demande aussi 4 messages, et un délai de cooldown à 2mn, tjs pour tenter d'éviter les paquets zombis du passé

III) Contrôle de flux

III.1) Fenêtre TCP

- Si l'émetteur est trop rapide, le buffer recepteur va saturer son buffer et risque soit de saturer sa mémoire, soit de perdre des données reçues.
- Ce qu'on appelle fenêtre est la taille de buffer disponible. Le recepteur l'indique à l'émetteur à chaque occasion, pour ajuster les envois. On appelle donc ça *Advertised-Window*.
  - On commence à être justes en entête donc on va prendre que 2 octets pour ça. Ca fait qu'on peut annoncer que 65k au max

– En pratique sous linux, le buffer fait 4Mo. On annoncera une fenêtre de min(taille buffer, 65k)

III.2) Maximiser le remplissage des paquets

1. Algorithme de Naggle

   - Répond à la question de quand bufferiser et quand envoyer même si le paquet n'est pas plein.
     – Comme dit, si les paquets ne sont pas pleins, c'est inefficace. Si une interaction est bloquée sur 3 octets qui dorment, c'est inefficace.
   - La taille minimale d'un paquet est donnée par les technos du dessous.
     – Ethernet a besoin d'envoyer au moins 1500 octets par paquet pour détecter les collisions.
     – Et puis, si on envoie des petits paquets, le poids des entêtes devient prédominants.
     – Mais si on envoi de trop gros paquets, on perd les bonnes propriétés de la commutation de paquets vues la semaine passée.
     – TCP tente d'envoyer des multiples de MSS (max segment size) qui sont une grandeur IP pour éviter la fragmentation: IPv4: 536 octets; IPv6: 1220 octets
   - Algorithme de Naggle:
     – Si quantité data >= MSS et Windows >= MSS
       * Envoyer un segment de taille MSS
     – Sinon, si y'a des données en l'air (attente d'ACK)
       * Mettre en buffer
     – Sinon
       * Envoyer le paquet sans attendre pour éviter de bloquer le dialogue interactif
   - Cet "algorithme" évite de saturer de petits paquets sans trop tuer l'interactivité. On se contente de peu de choses.

2. Delayed ACK

   - Quand on a reçu des informations, il faut prévenir l'émetteur que c'est bon, c'est bien reçu. Mais au lieu de faire un paquet avec juste numéro d'ACK, on tente de faire passer ça avec des données dans l'autre sens applicatif. En pratique, avec la réponse de l'application.
   - Piggybacking: voyager à dos de cochons en anglais, c'est un peu comme voyager au frais de quelqu'un d'autre (minecraft anyone?)
   - Le mécanisme de delayed ACK retarde l'envoi d'un paquet avec juste l'ACK de 200ms pour laisser l'appli faire une réponse dans laquelle on pourra se greffer.
   - Les interactions avec Naggle sont mauvaises (deadlock temporaire) alors on peut désactiver Naggle d'un `TCP_NODELAY` et bufferiser coté applicatif

III.3) Communiquer malgré les pertes de paquet

- Quand réémettre?
  - RTO (retransmit timeout): mécanisme de base basé sur la prédiction de RTT. On réémet quand on n'a pas reçu d'ACK alors qu'on aurait dû depuis 3RTT. Pourquoi 3? Ben parce que ça marche en pratique. Une autre version de TCP qui ferait de meilleures stats sur la gigue réseau pourrait utiliser une valeur dynamique plus adaptée. Et je n'oserais pas affirmer que c'est pas le cas en pratique.
  - Triple ACK: quand le receveur m'annonce 3 fois avoir reçu le même paquet, c'est qu'il a perdu celui d'après dans l'ordre logique. Je peux le renvoyer sans attendre le timeout.
- Quoi réémettre ?
  - un seul paquet par défaut. Y'a des ruses (selective ACK) pour faire mieux, mais c'est hors sujet.
- Si la fenêtre passe à 0, l'émetteur est bloqué. On a donc un risque de deadlock si l'ACK qui réaugmente W se perd en chemin.
  - le sender peut envoyer un paquet inattendu après un temps quand W=0, pour vérifier que c'est encore le cas. Au pire, le receveur le jete.

IV) Contrôle de congestion

- Un réseau saturé perd des paquets. Il faut tenter de l'épargner même s'il ne peut pas signaler explicitement ses problèmes

IV.1) Motivation: l'écroulement historique de 1986

- Le réseau était très chargé, les routeurs étaient en limite de charge, avec les files d'attente trop pleines
- Une arrivé massive de données fait que les files d'attentes explosent. Pertes de paquets massives (drop par les routeurs)
- Les communications en cours timeoutent, et réémettent massivement les données perdues.
- Les routeurs saignent, et tout prend feu

IV.2) Problématique et principe

- Challenge:
  - Déterminer la capacité instantanée du réseau (qui ne se signale pas puisqu'il est neutre, avec de la redondance)
  - S'adapter dynamiquement au changements
  - En plus, on voudrait un partage équitable entre les flots concurrents
- Idée générale: l'émetteur ajuste une taille de fenêtre à l'état du réseau
  - La difficulté majeure est de savoir interpréter les signaux faibles de saturation: drop, delay, ACK

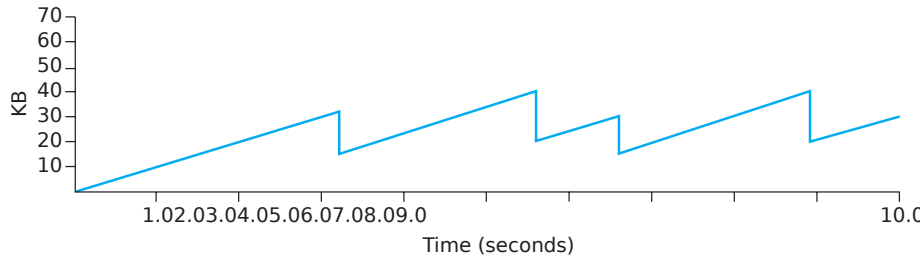
  - Ensuite, on utilise cwnd = $W_{congestion}$ à coté de W_{flow}: $W = min(W_{congestion}, W_{flow})$
- C'est le slow-start de TCP: on n'envoi pas 5Go sur le réseau sans réfléchir. On commence petit et on augmente les doses peu à peu jusqu'à toucher la limite.
  - Une perte (timeout ou triple ACK) signale une congestion. Il faut réduire, mais de combien ?
  - Un ACK signale une absence de congestion. On peut augmenter, mais de combien ?

IV.3) Tirer le meilleur parti du réseau

- L'objectif serait d'avoir un nombre constant de paquets en vol.
- Si l'émission est trop rapide par rapport à l'optimal, la saturation augmente exponentiellement. TCP divise alors la taille par 2 pour réduire exponentiellement en cas de pb.
- Si les augmentations sont exponentielles aussi, le protocole fait des à-coups, des saturations brutales et des fluctuations violentes sans converger. Mais si les augmentations sont linéaires, on va mettre trop longtemps à découvrir la bonne bande passante (lien plein sans perte).
- Ce qu'on voudrait, ce serait de faire des dents de scie autour de la valeur actuelle de la bande passante disponible (pour s'adapter si ça change)

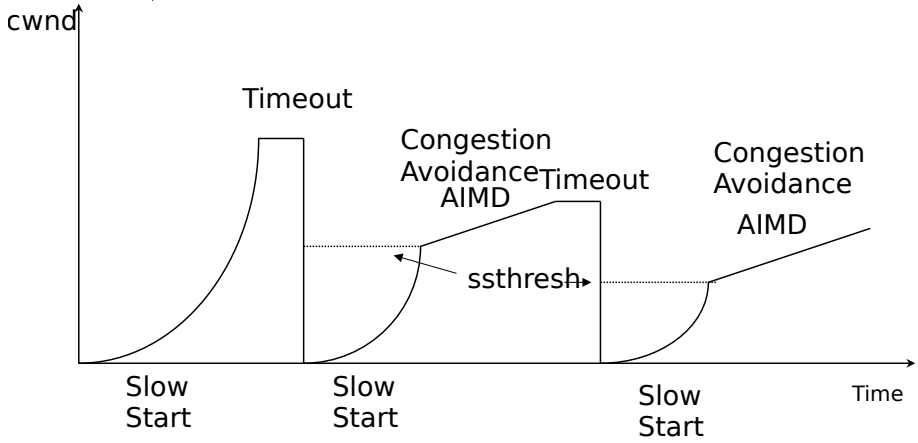
IV.4) Comportement de TCP

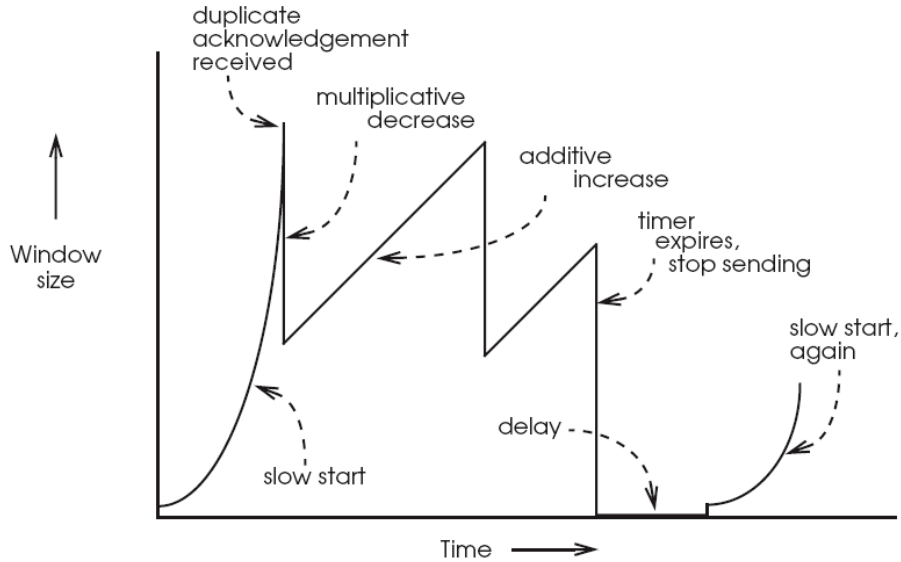
- Donc TCP a deux modes: SS et CA.
  - Il commence en slow-start (SS) en première estimation, et cherche à saturer le lien. Augmentation exponentielle pendant cette première phase.
  - Dès qu'on a réussi à saturer le lien (à la première perte de paquet), on passe en Congestion Avoidance autour de cette valeur estimée.
  - Ensuite, alterne entre SS et CA en fonction de la taille la fenêtre par rapport à un seuil `ssthresh` qui cherche à estimer la bw dispo:
    * W <= `ssthresh`: mode SS ; W > *ssthresh*: mode CA
- En mode CA, TCP est AIMD (additive increase, multiplicative decrease). Ca fait des dents de scie autour de la bande passante optimale
  - Mais remarquez que ça décolle pas vite au début, d'où le mode SS pour accélérer les choses



- Réagir aux bonnes nouvelles: Quand j'ai un ACK

  - SS est MIMD $\rightsquigarrow$ double cwnd sur un RTT (donc cwnd += MSS par ACK : pour chaque paquet qui passe, j'en pousse MSS de plus)
  - CA est AIMD $\rightsquigarrow$ cwnd += MSS par RTT (donc cwnd += $\frac{MSS^2}{cwnd}$ par ACK)
- Réagir aux mauvaises nouvelles:
  - si j'ai un triple ACK (3 fois la même valeur annoncée): on a perdu un paquet, donc on poussé le bouchon trop loin le coup d'avant. donc on émet 2 fois trop vu qu'on est en multiplicative increase.
    * Donc: ssthresh = cwnd/2; cwnd /= 2, et je repars en mode CA
  - si j'ai un timeout, l'heure est grave, les conditions ont dû changer.
    * Donc: ssthresh = cwnd/2; cwnd = 1 , et je repars en mode SS (jusqu'à que cwnd > ssthresh)





- En plus, l'AIMD converge vers un point d'équilibre à la fois optimal en partage de BW entre les flux et efficace (usage complet du lien).
  - Pour plus d'info, lisez directement le papier de Chiu Jain (`https://www.cse.wustl.edu/~jain/papers/ftp/cong_av.pdf`)
    * ou ces analyses plus digestes `https://www.cs.helsinki.fi/u/ldaniel/mm_cn/`

lec1.1_cc_aimd_chiu_jain.pdf https://www.cs.helsinki.fi/u/ldaniel/mm_cn/chiu-jain-slides.pdf

– Pour une version plus abstraite, game theory, regardez https://www.di.ens.fr/~busic/mar/

Chapter 7

Couche Routage

I) Introduction

- IP, c'est LE protocole d'internet.
  - L'objectif est de passer des paquets entre différents réseaux interconnectés
  - pour constituer un super-réseau formé de sous-réseaux indépendants
- Première réponse historique
  - protocole de transport de paquets IMP (1969, Arpanet), protocole NCP par dessus
  - Toutes les fonctionnalités dedans, très intégré, mais modèle rigide. Difficile d'interconnecter des réseaux hétérogènes
- Réponse d'Internet: protocole tout intégré en 1974 et bientôt découpé en TCP, UDP et IP.
  - Cela a valu le prix Turing 2004 à Vint Cerf et Robert Kahn
  - David C Clarke "The design philosophy of DARPA Internet Protocols" (1988) explique à posteriori les choix de design (notes de lecture "morning paper" ajouté sur le site du cours pour les curieux)
  - Objectifs principaux:
    * Usage multiplexé (pas de circuit établis)
    * Interconnexions de réseaux disparates existants
  - Objectifs secondaires:
    * Continuité de service malgré les pannes
    * Différents types de services et de réseaux: Tailles de paquets, adressage, modèle de service
    * Management/contrôle décentralisé et dynamique (arrivées/départs noeuds)
    * Objectif d'efficacité (contrôle de congestion, usage optimal des ressources malgré les variations perf)
  - Modèle de service choisi: datagram sans connexion en best-effort
    * C'est le plus petit dénominateur commun, mais faire plus est parfois contre-productif donc autant laisser les couches hautes faire ce qu'elles veulent
- Principe d'IP (annonce de plan)

  - Entêtes des paquets comme interface standardisée
    * élément fondamental du protocole entre émetteur/récepteur et récepteur/applications
    * l'implémentation n'est pas standardisée (moins rigide)
  - Adressage: chaque noeud a une adresse unique
  - Forwarding: quand un paquet arrive sur un routeur, il est renvoyé sur la bonne interface (la bonne carte réseau) en sortie, en fonction d'une table de routage.
  - Routage: calcul des tables de routage

II) Adressage IP

- En fait, c'est chaque carte réseau qui a une adresse. Donc un routeur avec 4 cartes réseaux a 4 adresses IP.
- IPv4: entier 32 bits, lu par paquet de 8 bits; IPv6: entier 128 bits, lu en hexadécimal
- On voudrait donner des adresses similaires aux ordinateurs proches dans le réseau afin de compacter les tables de routage
  - À la poste, on fait suivre par pays puis par région puis par ville puis ...
  - IP groupe les adresses "proches" par préfixe commun. Il faut différencier entre le préfixe "réseau" et le suffixe "ordinateur de ce réseau là"
- À l'origine, la frontière est directement dans l'adresse, en fonction du préfixe:
  - si ça commence par "0" (en binaire), c'est une adresse classe A, donc 7 bits pour encoder le réseau et 24 bits pour encoder la machine. On peut avoir 127 tels réseaux, de 2^{24} machines chaque 16,777,216.
  - si ça commence par "10", classe B. Donc 14 bits pour le réseau et 16 bits pour la machine. 2^{14} réseaux (16,384) de 2^{16} machines (65,536)
  - si ça commence par "110", classe C. Donc 24 bits pour réseau et 8 bits pour la machine. 2^{21} réseaux (2,097,152) de 2^{8} machines (256)
- Mais il y a rapidement eu pénurie de réseaux (surtout les B)
  - Maintenant adresses CIDR (Classless InterDomain Routing), où l'on précise la taille du préfixe à part
  - Deux formes équivalentes: masque binaire (255.255.255.0) ou nombre de bits du réseau en postfix (/24)
  - Le préfix n'est pas limité à une taille multiple de 8. Adresse /17 pour un gros ISP
- Question: quelle est la plage d'adresse représentée par 128.138.207.160/27 ?
  - 128.138.207.160 = 10000000 10001010 11001111 101|00000 en binaire, et 27 bits vont jusqu'à la barre
  - 10000000 10001010 11001111 101|11111 en binaire (la plus grande machine), c'est 128.138.207.191
  - Donc la plage, c'est 128.138.207.160 - 128.138.207.191
- Question: quel est le masque dans le cas précédent?
  - 128.138.207.160/27 = 128.138.207.160/255.255.255.224
  - Autre: 195.176.181.11/30 = 195.176.181.11/255.255.255.252

  - $\rightsquigarrow$ c'est une adresse dans la plage [195.176.181.8; 195.176.181.11]
    - $11_{\text{d}} = 0000\ 1011_{\text{b}}$ et $0000\ 1000_{\text{b}} = 8_{\text{d}}$
  - Autre: 192.168.0.3/24 = 192.168.0.3/255.255.255.0
    $\rightsquigarrow$ c'est une adresse dans la plage [192.168.0.0; 192.168.0.255]
- On regarde l'exemple d'adressage de sous-réseau sur le document
  - Réseau 1 est forcément 128.96.34.0 car il contient des petites valeurs et est de masque 128, donc le premier bit est un 0
  - Valeurs possibles pour Masque 2: ce qu'il faut pour encadrer le contenu
    - Contenu: $129_d = 1000\,0001_b$; $130_d = 1000\,0010_b$; $140_d = 1000\,1100_b$
    - 255.255.255.128 ($128_d = 1000\,0000_b$)
    - 255.255.255.192 ($192_d = 1100\,0000_b$)
    - 255.255.255.224 ($224_d = 1110\,0000_b$)
    - 255.255.255.240 ($240_d = 1111\,0000_b$)
  - Pour le réseau 3, on peut prendre simplement 128.96.33.0/24 (255.255.255.0)
- Il existe des adresses privées, spécifiques.
  - 127.0.0.1/32: c'est l'adresse de la machine locale, tout le temps
  - 192.168.0.0/16: réseau privé, utilisé pour à la maison derrière un NAT
- Habituellement, la première adresse de la plage est pour le réseau ..*.0
  - La dernière adresse de la plage est pour le broadcast
  - Donc on peut mettre 254 ordinateurs par réseau/24

III) Forwarding

III.1) Principe

- Principe: on prend le préfixe de longueur maximale parmi les routes existantes
- Le travail de base d'un routeur est donc de trouver le lien de sortie pour chaque adresse destination des paquets entrants
  - Il faut aller très vite (jusqu'à max 20 ns par paquet)
  - Si les préfixes étaient bien répartis, ça irait mais
    pénurie d'adresses $\rightsquigarrow$ fragmentation
    (qques tables de 25k lignes au coeur du réseau)
  - ASIC = processeurs spécialisés (+ des ruses algorithmiques hors sujet ici)
- Exercice 2 sur le document associé. Calculer l'interface de sortie
  - 123.4.1.69 $\rightarrow$ 1 (seulement dans 123.4.0.0/16, ligne 1)
  - 68.142.226.44 $\rightarrow$ 4 (seulement dans 0.0.0.0/1, avant dernière ligne)
  - 98.7.2.71 $\rightarrow$ 2 (dans le réseau 98.7.1.0/16, ligne 2)
  - 200.100.2.1 $\rightarrow$ 3 (dans le réseau énorme 128.0.0.0/1, ligne 4)
  - 128.138.207.167 $\rightarrow$ 4 (à la fois dans le réseau énorme et dans 128.138.0.0/16)
  - 123.4.20.11 $\rightarrow$ 2 (dans le réseau 123.4.20.0/24 et 123.4.0.0/16, donc on prend le long)
  - 123.4.21.10 $\rightarrow$ 1 (seulement dans 123.4.0.0/16)

- Rien ne dit que le routage est symétrique sur Internet, et calculer les tables optimales est difficile. On y revient

III.2) Autres mécanismes d'IP

- Fragmentation, quand le MTU du second réseau est plus petit
  - Il y a un champ dans les entêtes pour numéroter les sous-paquets
  - Réassemblage à destination ou en chemin si les paquets se croisent
  - Forcément, c'est moins efficace quand ça fragmente
- TTL, pour éviter les boucles (dues aux mauvaises configurations)
  - c'est en nombre de sauts, et le diamètre max d'internet est donc 256
- Protocole de contrôle IP: ICMP
  - Signalement des pbs de TTL ou de checksum
  - Permet de faire des ping et des traceroutes, mais aussi des DDos donc de moins en moins utilisé
- Pénurie d'IP publiques: NAT (network address translation)
  - On cache un réseau privé entier derrière une seule IP publique
  - Exemples: salle de TP ou tous les hôtes derrière un set-top-box = modem router
  - Multiplexing des machines masquées sur les ports de la machine publique

III.3) Protocoles associés (OPTIONNEL)

- Dans un réseau donné, il faut trouver l'adresse MAC du destinataire. C'est le protocole ARP
  - broadcast "Qui a l'IP bidule?"; réponse de l'intéressé
  - tout le monde met toutes les infos (y compris MAC demandeur) en cache avec timeout
  - car oui, tout le monde écoute tout sur un réseau éthernet ou wifi
- Pour obtenir une adresse IP dynamique, sans config: DHCP
  - client broadcast "Discover"
  - serveur broadcast "offer"
  - client broadcast "request" (pour gérer le cas où deux serveurs font des offres différentes)
  - serveur broadacst "ACK"

IV) Routage

IV.1) Introduction

- L'objectif est de calculer les tables de routages
  - Doit fonctionner de façon dynamique, décentralisée et à large échelle
- Le problème est très différent intra- ou inter-AS (autonomous system = ISP ou entreprise ou univ)

- intra-AS: dans une organisation donnée
  - on a toutes les infos, relativement homogène
  - on vise l'optimal technique (usage des ressources et performance)
- inter-AS: entre institutions ou entreprises concurrentes
  - configuration interne = secret commercial;
  - on vise l'optimal commercial (négociation financière)

IV.2) Routage intra-AS

- Le réseau est un graphe, chacun connait ses voisins (remember ARP)
- Latence comme métrique de distance additive
- Le problème est une recherche de plus court chemin, à faire en temps réel distribué (et dynamique)
- Deux approches possibles pour calculer le plus court chemin:
  - Vue locale du réseau, et calcul distribué (vecteur d'état)
  - Vue globale du réseau, et calcul local (état des liens)

1. Algorithme par vecteurs d'états (Bellman-Ford 1956)

   - Distance-vector routing en anglais
   - Chaque routeur u maintient les informations suivantes pour toute destination v:
     - $D_u[v]$: Longueur du plus court chemin connu vers v
     - $N_u[v]$: Next hop, i.e. voisin à qui u doit envoyer les infos à destination de v
   - Deux vecteurs de taille 10 dans l'exemple du doc (10 machines nommées)
   - Algorithme
     - Valeur initiale de $D_u[v] = \infty$ si v n'est pas voisin de u, ou poids de l'arrête
     - À chaque itération, chaque routeur envoie son vecteur D à ses voisins
     - À la réception du vecteur de distance D_x envoyé par x + $c(u,x)$ coût de $\vec{ux}$, le routeur u décide de passer par x si c'est mieux que la route connue
       - $\forall v$, si $c(u,x) + D_x[v] < D_u[v]$ alors $N_u[v] = x$ et $D_u[v] = D_u[x] + D_x[v]$
       - Les infos reçus sont conservées. Utile quand les choses empirent
   - Déroulé de l'algorithme sur l'exemple de la feuille (moitié gauche seulement)

nD↷	A	c	d	e	f
A	–	?∞	?∞	e2	?∞
c	?∞	–	d4	e3	?∞
d	?∞	c4	–	e5	f7
e	A2	c3	d5	–	f3
f	?∞	?∞	d7	e3	–

Situation initiale

nD↷	A	c	d	e	f
A	–	?∞	?∞	e2	?∞
c	?∞	–	d4	e3	**d11**
d	?∞	c4	–	e5	f7
e	A2	c3	d5	–	f3
f	?∞	**d11**	d7	e3	–

Broadcast de D_d vers c, e et f.

nD↷	A	c	d	e	f
A	–	**e5**	**e7**	e2	**e5**
c	**e5**	–	d4	e3	d11
d	**e7**	c4	–	e5	f7
e	A2	c3	d5	–	f3
f	**e5**	**e6**	d7	e3	–

Broadcast D_e vers A, c, d, f.

- Cet algorithme converge en au plus $O(diametre)$ étapes vers le routage parfait
- Mais les mauvaises nouvelles convergent mal. Si $e\vec{A}$ tombe en panne
  - e annonce $[\infty, ...]$
  - À la réception, c annonce $[11, ...]$ en passant par d
  - À la réception, d annonce $[15, ...]$ en passant par c
  - À la réception, c annonce $[19, ...]$ en passant par d
  - À la réception, d annonce $[23, ...]$ en passant par c
  - À la réception, c annonce $[27, ...]$ en passant par d
  - Ils vont mettre longtemps à revenir à ∞ :(
  - En pratique (protocole RIP), on compte en nombre de sauts, et $16 = \infty$
- Clivage de l'horizon
  - u n'annonce pas à v les distances des destinations x où $N_u[x] = v$
  - Converge plus vite quand un lien tombe, mais pas parfait
- Reverse poisonning: $u \to v$: $D_u[x] = \infty$ si $N_u[x] = v$
  - Converge encore plus vite, mais boucle à 3 éléments encore possible
  - BGP utilise un algorithme proche, mais chacun annonce les chemins complets qu'il prend pour les destinations, ce qui permet à l'interlocuteur de gérer les cycles
- Une variante de cet algo est utilisé en pratique par le protocole RIP

2. Algorithme par état des liens (Dijkstra 1960)
   - L'objectif est de donner à tous une vision complete de l'état du réseau, pour que chacun applique Dijkstra pour lui
   - Chacun broadcast l'état de ses liens locaux de temps en temps;
     Comment faire pour que tout le monde reçoive tout?
   - Idée 1: flooding.
     - Quand je reçois un tel message, je le fais suivre à tous mes voisins (sauf émetteur)
     - Sympa, mais explosion nb messages s'il y a des boucles
   - Idée 2: Reverse Path Broadcast
     - Quand u reçoit de v les infos que x a broadcasté, u fait suivre ssi $N_u[x] = v$
     - (broadcast uniquement les infos qui parviennent sur le plus court chemin)
     - c fait suivre les infos de A seulement quand elles viennent de e (pas si elles arrivent de d)

  - Sympa, sauf que je ne connais justement pas le plus court chemin avant que tout le monde ait tout broadcasté
- Idée 3: Flood contrôlé par numéro de séquence
  - À l'émission, chaque paquet broadcasté a un numéro de séquence, qui augmente pour le broadcast du prochain
  - En local, chacun stocke n_v, le dernier numéro de séquence reçu de v
  - À la réception d'un paquet de v numéroté N, je le broadcast ssi $N > n_v$, puis mise à jour $n_v \leftarrow N$
  - (je fais suivre les paquets seulement la première fois que je les vois)
- Ensuite, l'algo de Dijkstra est similaire à Bellman-Ford mais en local et sans pb
  - D_u : distance pour aller de là où je suis jusqu'à u
    $p[v]$: précédent de v, là par où j'arrive à v en venant de là où je suis
  - Initialisation de toutes les distances à ∞, sauf les voisins du noeud local
    * ensemble todo contenant tous les noeuds sauf le noeud local
  - À chaque tour de boucle, sélectionne le noeud $u \in todo$ tq D_u est minimale
    * pour tous les voisins de u dans *todo*, notés v
      · si $D_u + c(u,v) < D_v$ alors $D_v = D_u + c(u,v)$ et $p_v = u$
  - Quand j'ai fini l'exécution (quand $todo = \emptyset$), j'utilise récursivement les p_x pour savoir par où sortir pour aller à un noeud donné
- Exemple d'exécution de Dijkstra pour le noeud d

A		B		c		**d**		e		f		g		h		i		j		
D	p	D	p	D	p	**D**	**p**	D	p	D	p	D	p	D	p	D	p	D	p	
∞	?	∞	?	4	d	–	–	5	d	7	d	3	d	∞	?	∞	?	∞	?	init
						–	–							15	g	9	g	10	g	g
						–	–													c
7	e					–	–													e
						–	–													A
						–	–							14	f					f
		12	i			–	–													i
						–	–							11	g					j
						–	–													h, B

- Une variante de cet algo est utilisé en pratique par le protocole OSPF

IV.3) Routage inter-AS

- C'est très différent d'intra-AS car c'est un problème économique et non un pb technique
  - Les ISP ne veulent pas communiquer l'état exact de leur réseau interne (trust issue)
  - On peut vouloir forwarder vers un réseau moins performant mais moins cher
- Link state/Dijstra nécessite info complete, pb extensibilité et impossible de moduler sa politique
- Vecteur distance/Bellman ne permet pas de moduler sa politique, converge lentement

- Algo de vecteur de chemins à la place.
  - Comme un vecteur de distance, mais on propage le chemin complet qu'on prend pour chaque destination
  - Plus de mémoire et taille message, mais pas plus de calculs
- Protocole BGP (Border Gateway Protocol) utilisé en pratique
  - Chaque AS annonce les autres AS qu'il est capable d'atteindre, avec info de distance (et la route des AS utilisées pour y aller, donc)
  - Pas de boucles possibles puisque routes complètes annoncées (sauf si churn sévère)
  - Chaque AS peut mentir par omission et ne pas annoncer certaines routes pour lesquelles il sait router mais ne souhaite pas prendre en charge les données des autres
  - Ca passe bien à l'échelle, mais atteignabilité pas garantie : si personne veut faire passer le trafic des autres, ça marche forcément pas

- Exemple sur le réseau à router du doc.
  - 1: e annonce savoir aller à A directement. e:(A,1,$\emptyset$)
  - 2: d et j ne font pas suivre l'info de e (trop cher?)
  - 2: f fait suivre l'info (A,2,{e})
  - 3: d:(A,3,{e,f}) et h:(A,3,{e,f})
  - 4: g:(A,4,{e,f,d})
  - 5: i peut choisir entre (A,4,{e,f,h}) et (A,5,{e,f,d,g})
- Si e annonce ensuite ne pas pouvoir rejoindre A, d et f ne vont pas partir en boucle puisque chacun sait que l'autre comptait sur le lien qui vient de disparaître

Chapter 8

Conclusion

- C'est la fin du module. Lundi prochain, partiel.
- D'autres modules vous demandent de savoir lire du C (archi, C++), mais si vous ne cherchez pas à poursuivre, vous ne referez jamais plus de réseau. Avant l'agreg :)

I) Biblio

Mon approche pour enseigner la partie réseau est directement inspirée du livre de Jim Kurose (dans toutes les bonnes bibliothèques), centrée sur les concepts (qui restent) plus que sur les technologies (qui lassent et passent), et suivant une approche top-down voire problem solving. J'insiste également volontiers sur les algorithmes distribués mis en jeu, même si le temps manque dans ce module pour les détails.

La plupart des ressources disponibles en ligne sont plutôt bottom-up, où l'on regarde ce qu'on peut faire avec les outils qu'on comprend déjà. Je n'aime pas cette approche car on ne comprend qu'à posteriori pourquoi chaque bout est tel qu'il est. Pourtant, la plupart des choix de design que les architectes d'internet ont fait étaient motivés par un besoin dans une couche supérieure.

Méfiez-vous surtout des ressources trop techniques (qui se limitent souvent à des catalogues de technos n'abordant jamais le pourquoi), voire technologiques (qui peuvent tout au plus vous apprendre à configurer votre réseau à la main). Voici quelques saines lectures pour compléter le cours :

- Le livre de Kurose, bien sûr, mais il n'est pas disponible en ligne.
- Tanenbaum and Wetherall, Computer Networks. Un autre livre de référence habituel.
- Peterson and Davie, Computer Networks: A Systems Approach. Un livre complet en accès libre. `https://book.systemsapproach.org/`
- Computer Networks, cours de l'université de Brown enseigné par Rodrigo Fonseca. Les transparents associés aux cours sont très bien. L'approche est bottom-up alors je vous conseille de lire les chapitres à l'envers pour bien comprendre, mais ils ont un très bon niveau de détail et d'abstraction. Les projets proposés sont également très bien, mais trop chronophages au vu du volume horaire attribué au module. `http://cs.brown.edu/courses/cs168/f17/schedule.html`
- Computer Networking un livre disponible sous licence libre. Il est de bonne facture,

même s'il est lui aussi bottom-up. http://cnp3book.info.ucl.ac.be/

Notes d'animation

Séance 6: Communiquer en réseau

- **1 doc à imprimer:** code client/serveur BSD
- **Programme du jour:**
  - Présentation du demi-module
  - chap1: réseau de réseaux
  - chap2: Faire communiquer des programmes
- **TODO:**
  - Passer le chap2 (faire communiquer des progs) en tête, en préparation des séances pratiques,
  - réorganiser l'ensemble pour fusionner les chap 1, 4 et 3, présentant un réseau de réseaux en général puis Internet en particulier

Séance 7: Internet

- **rien à imprimer**
- **Programme du jour:**
  - Chap3: Performance du réseau
  - Chap4: Internet
- **TODO:**

Séance 8: Couche applications

- **rien à imprimer**
- **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

Séance 9: Couche transport

- **rien à imprimer**
- **Épisode précédent:**
  - Forme d'organisation des applications distribuées
    * Client/serveur, Pair-à-pair, Coordinator/worker
    * Push et Pull
  - Protocoles notables
    * mail: affreusement complexe, avec des protocoles imbriqués sur 3 étages et un protocole réseau pas simple
    * ftp: un protocole réseau plus simple, mais pas très extensible
    * http: une trèe belle idée, avec une encapsulation sur 2 étages seulement (entête et data), évolutif
    * et l'idée générale que la généricité et l'évolutivité valent mieux que les performances
  - Rq 2122:
    * Statefull/stateless: état persistant ou non.
    * Rdv initial en P2P sur un point central.
    * HTTPS est un protocole encapsulé (comme MIME dans les mails), tandis qu'AJAX est une façon d'utiliser les HTTP modernes pour permettre la communication entre le client et le serveur sans devoir retélécharger toute la page. Au passage, depuis 2011, Websocket est préférable à AJAX :)
    * ACID: atomique, cohérent (d'état valide en état valide, tout le monde a la même réponse à la même question), isolé (linéarisable, pas d'interaction entre transitions), durable (si on tire la prise)
    * CAP: sur un systeme asynchrone qui peut perdre des messages. On coupe le système en deux et tous les messages entre sont perdus. Si y'a une écriture à gch, la lecture à droite ne peut pas l'avoir. Donc si elle répond, y'a eu un pb
    * BASE:
      · Basically Available (toujours une réponse),
      · Soft State (pas durable: les données s'effacent au bout d'un temps si on ne les republie pas, donc tout est en cache. Mais potentiellement, pas de cohérence géographique),
      · Eventual Consistent (pas d'atomicité ni d'isolation en chemin, seulement à terme)

- **Logistique ENS**: Ce moment de l'année est peut-être un peu stressant pour certains. Disons que ça s'est déjà vu dans le passé.
  - Discutez avec vos camarades de classe, discutez en avec moi, discutez en avec vos tuteurs
  - Mon discord et ma boite mail vous sont grand ouverts
  - 2122: Préparez vous à la vague COVID dont on espère qu'elle va se dégonfler. Un téléphone avec caméra, un casque micro, une bonne co, un endroit OK

Séance 10: Couche routage

- **1 doc à imprimer:** document 10
- **Épisode précédent:**
  - TCP offre la fiabilité, UDP est en mode DIY pour le reste
    - mécanisme de base avec deux buffers sender/recver
    - Etablissement en 3-ways pour éviter les paquets du passé
  - Controle de flux (préserver le recepteur)
    - fenêtre annoncée dans flux en sens inverse
    - Problématique de remplissage des paquets (Naggle et delayed ACK)
    - Perte de paquets (triple ACK=NACK; timeout)
  - Controle de congestion (usage au plus juste de la BW dispo)
    - Alternance entre SS (MIMD, si cwnd<ssthresh) et CA (AIMD, si cwnd>ssthresh)
    - triple ACK: ssthresh := cwnd/2; cwnd /= 2 et on repart en CA
    - timeout: ssthresh := cwnd/2; cwnd = 1 et on repart en SS