9: Couche transport

Martin Quinson

Épisode précédent 1
(rien à imprimer) 1

I) Introduction à la couche transport 2
I.1) UDP 2
I.2) TCP 2

II) Mécanismes de base de TCP 4
II.1) Buffers de communication 4
II.2) Etablissement de connexion TCP 5

III) Contrôle de flux 6
III.1) Fenêtre TCP 6
III.2) Maximiser le remplissage des paquets 6
III.3) Communiquer malgré les pertes de paquet 7

IV) Contrôle de congestion 7
IV.1) Motivation: l'écroulement historique de 1986 7
IV.2) Problématique et principe 8
IV.3) Tirer le meilleur parti du réseau 8
IV.4) Comportement de TCP 8

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)
- **Info 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

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

I.2.a) 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.

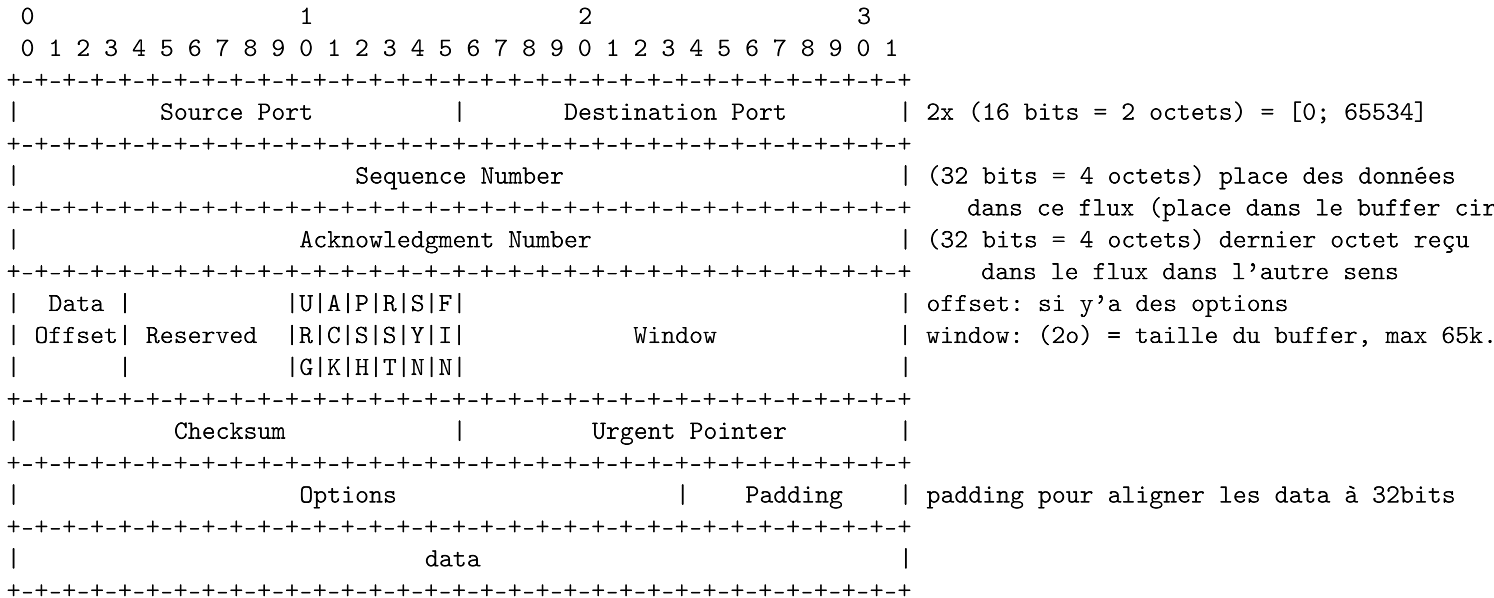
I.2.b) Design de TCP

- 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.

I.2.c) 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.

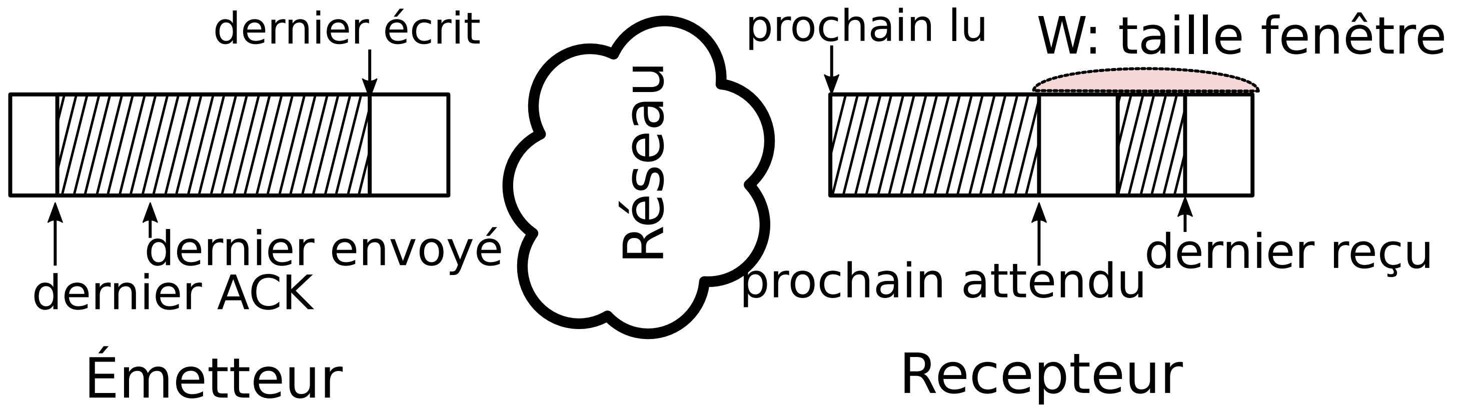


- On peut aussi faire de beaux diagrammes d'états pour l'établissement de la connexion, ou de gantt chart sur la fin d'une connexion.
- C'est magnifique, on peut parler pendant des heures sans que personne comprenne.
- Je vais plutôt chercher à expliquer **pourquoi** le protocole TCP est tel qu'il est, 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 *AdvertisedWindow*.
  - 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

III.2.a) 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.

III.2.b) 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

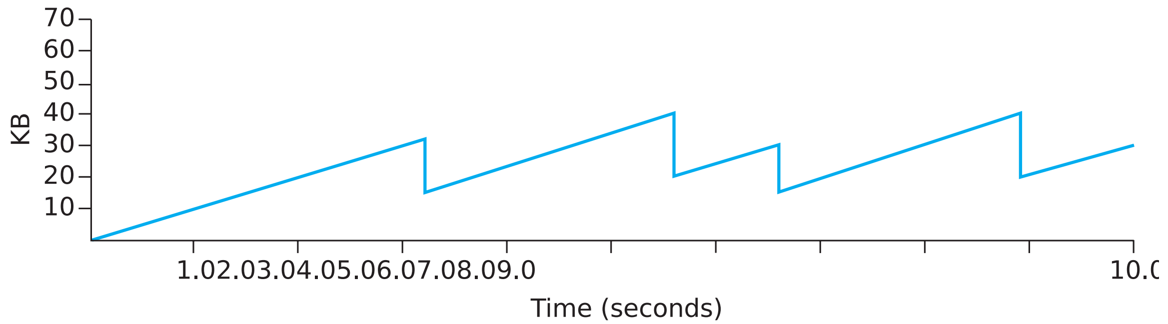
- Challenge:
  - Déterminer la capacité instantanée du réseau (qui ne se signale pas puisqu'il est neutre, avec de la redondance)
  - S'adapter dynamiquement au changements
  - En plus, on voudrait un partage équitable entre les flots concurrents
- Idée générale: l'émetteur ajuste une taille de fenêtre 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

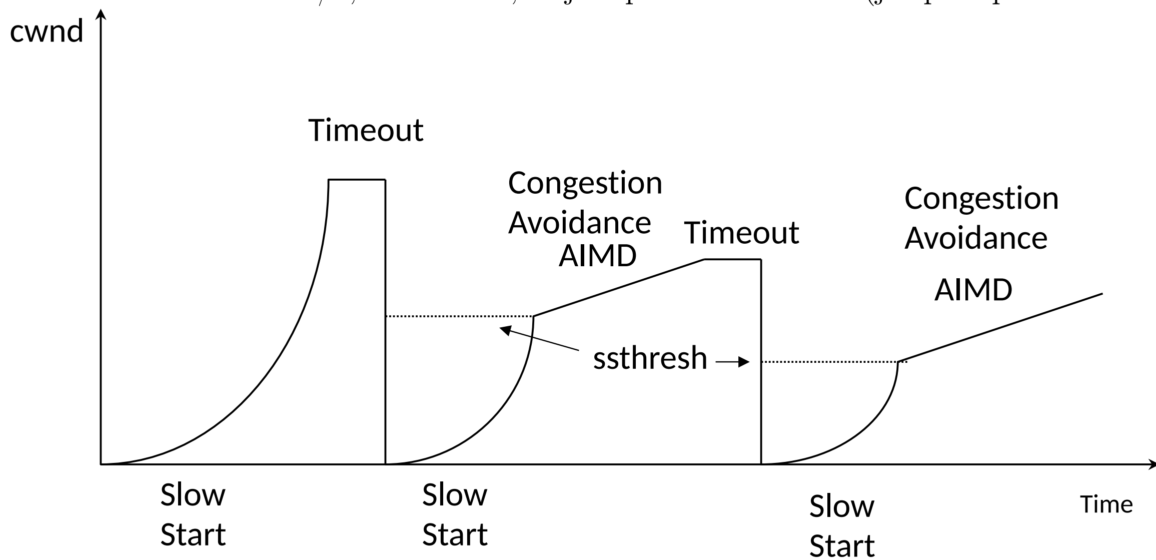
- L'objectif serait d'avoir un nombre constant de paquets en vol.
- Si l'émission est trop rapide par rapport à l'optimal, la saturation augmente exponentiellement. TCP divise alors la taille par 2 pour réduire exponentiellement en cas de pb.
- Si les augmentations sont exponentielles aussi, le protocole 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)

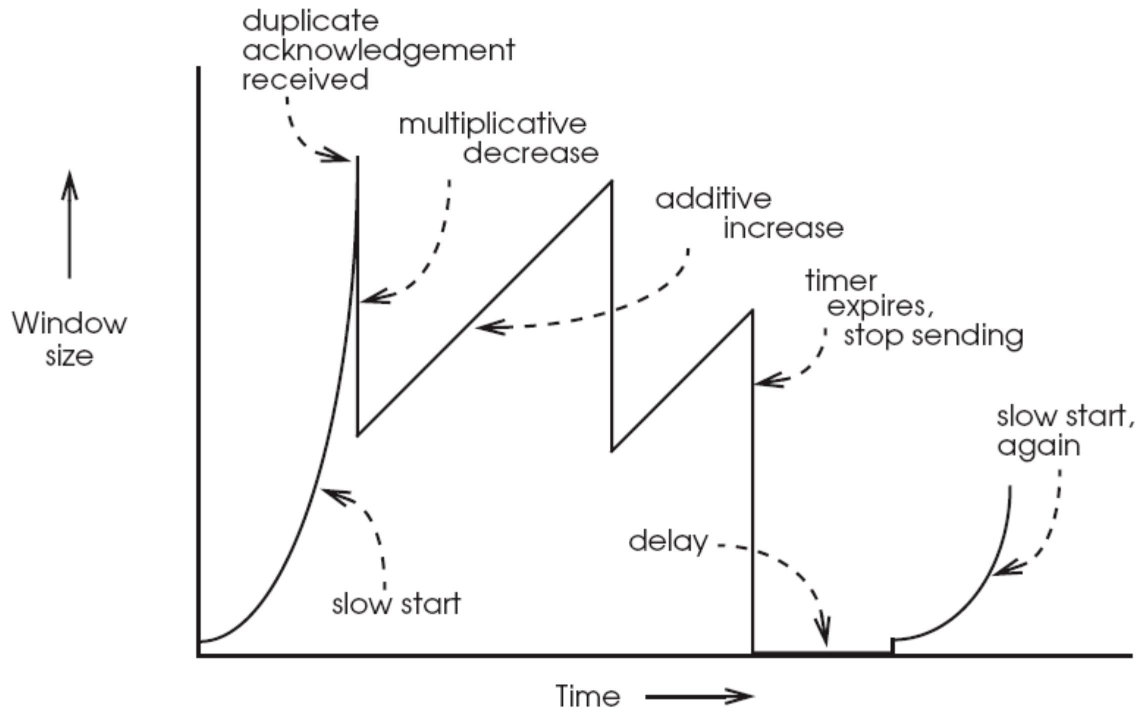
IV.4) Comportement de TCP

- 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/`