Protocoles Réseaux VI

Juliusz Chroboczek

25 octobre 2022

1 La couche transport

Deux protocoles de couche transport sont déployés dans l'Internet global : TCP et UDP. TCP est une couche épaisse au dessus de IP, et fournit un service par flots, fiable, ordonné, avec contrôle de flot et de congestion. UDP, au contraire, est une couche très fine, qui fournit un service presque identique à celui fourni par IP, lui ajoutant seulement le multiplexage.

Il existe en principe d'autres protocoles de couche transport, notamment DCCP, UDP-Lite et SCTP. Ces protocoles ne traversent normalement pas les NAT, ce qui les rend inutilisables en pratique. Même en IPv6, où il n'y a pas de NAT, ils sont généralement bloqués par les *firewalls*. Cette impossibilité de déployer de nouveaux protocoles de couche transport s'appelle l'*ossification de l'Internet*.

Une solution consiste à coder les nouveaux protocoles de couche transport de façon à ce qu'ils traversent les NAT et les *firewalls*. Pour cela, on peut soit les implémenter comme des extensions à TCP (par exemple MP-TCP, utilisé par *Siri* d'Apple, ou *tcpcrypt*), ou à les encapsuler à l'intérieur d'UDP (RTP, utilisé pour la vidéoconférence, et QUIC, utilisé par HTTP/3 ; il existe même une variante de SCTP encapsulée dans UDP).

1.1 Multiplexage

Les deux protocoles de couche transport fournissent des services très différents. Le seul service qu'ils implémentent tous deux est le *multiplexage*, qui permet d'exécuter plusieurs applications sur une seule adresse IP. Ce multiplexage parfois appelé *interne* ne doit pas être confondu avec le multiplexage dit *externe*, qui permet de partager un seul lien entre plusieurs hôtes, et qui est réalisé aux couches 2 et 3.

À la couche transport, l'extrémité d'un flot de communication est identifiée par un entier de 16 bits nommé *numéro de port* (il n'y a pas de « ports » — le nom est purement conventionnel). L'extrémité d'un flot de communication est donc identifiée par une *adresse de socket*, une paire (IP, p), où IP est une adresse IP de 32 ou 128 bits, et p un numéro de port de 16 bits.

Un certain nombre de numéros de port sont affectés, soit formellement [1], soit par l'usage, pour être utilisés par des application spécifiques ; par exemple, le port 25 est utilisé par SMTP, le port 80 par HTTP, le port 443 par HTTPS (HTTP au-dessus de TLS). Les numéros de port supérieurs à

1. `https://www.iana.org/assignments/service-names-port-numbers/`

quelques milliers environ ne sont pas affectés, et peuvent être utilisés librement. Sous Unix (mais pas Windows), les ports inférieurs à 1024 sont réservés à l'utilisateur `root` [2].

2 Le protocole UDP

Le protocole UDP (RFC 768) est une couche très fine au dessus de IP : le seul service supplémentaire fourni par UDP est le multiplexage. De ce fait, le service fourni par UDP a les mêmes caractéristiques que celui fourni par IP : un service par paquets de taille limitée, non fiable, non ordonné, sans aucun contrôle de flot ou de congestion. Si une application a besoin de communication fiable ou ordonnée, il faudra soit implémenter ces caractéristiques dans l'application elle-même, soit utiliser un protocole de couche transport plus riche.

Un *datagramme* UDP consiste d'un entête UDP suivi des données de couche application. Le datagramme entier sera passé au protocole de couche inférieure (IP) où il sera stocké dans le champ « données de couche supérieure » d'un paquet IP. L'entête UDP a une longueur de 64 bits (8 octets) et sa structure est donnée à la figure 1.

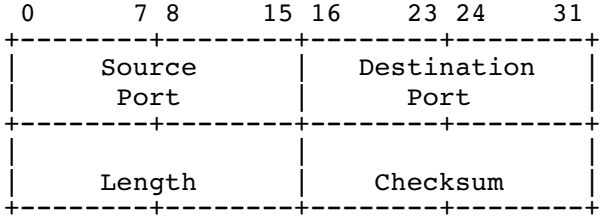

FIGURE 1 — Format de l'entête UDP

Les champs *Source Port* et *Destination Port* contiennent les ports source et destination ; concaténés respectivement aux champs *Source Address* et *Destination Address* de l'entête IP (couche réseau), ils forment les adresses de *socket* source et destination.

Le champ *Length* contient la longueur totale du datagramme, en octets. Le champ *Checksum* contient une somme de contrôle du datagramme, ce qui permet en principe d'éviter de remettre à l'application des datagrammes corrompus (il s'agit d'un mécanisme très faible ; en pratique il est généralement nécessaire de protéger les données à la couche application).

2.1 Service fourni

Le service fourni par UDP consiste de deux primitives :

- envoyer un datagramme à une destination (adresse de *socket*) donnée ;
- recevoir un datagramme.

Par ailleurs, UDP permet la *multi-diffusion* (*broadcast* et *multicast*) : un datagramme peut être envoyé à tous les hôtes du lien local, ou à un sous-ensemble arbitraire des hôtes de l'Internet. Nous en discuterons dans un cours subséquent.

2. Ce qui, soit dit entre nous, est complètement idiot. Les numéros de port devraient obéir aux permissions Unix, comme les fichiers et les périphériques.

2.2 Applications

Du fait de la pauvreté du service fourni, UDP est plus difficile à utiliser correctement que TCP. Historiquement, les principales applications d'UDP étaient les suivantes :

- les protocoles comme *DNS* et *Kademlia*, qui effectuent un grand nombre de *transactions* très simples (de type requête-réponse) avec un grand nombre de pairs ;
- les protocoles *temps réél* qui ne peuvent pas tolérer les délais imprévisibles introduits par le réordonnancement et la réémission de paquets, par exemple les protocoles de distribution de temps (NTP) ou les protocoles pour les jeux en ligne ;
- les protocoles multimédia, qui, outre le fait qu'ils ont besoin d'un service en temps réél, émettent un flux continu de paquets qui n'interagit pas toujours bien avec le contrôle de congestion de TCP (WebRTC, Skype) ;
- les protocoles de VPN, qui transportent des données de couche supérieure auxquelles le contrôle de congestion a déjà été appliqué, et qui interagissent donc mal avec le contrôle de congestion de TCP.

Plus récemment, certains développeurs ont commencé à contourner le manque de flexibilité de TCP en implémentant des protocoles de transport directement dans l'application au dessus d'UDP. On peut notamment citer le protocole *QUIC* qui vise à remplacer TCP comme support pour HTTP, et le protocole µTP utilisé par certaines implémentations de BitTorrent. (RTP, le format de paquets utilisé par certains protocoles de vidéoconférence, peut aussi être considéré comme étant un protocole de couche transport.)

3 Le protocole TCP

Le protocole TCP (RFC 793) est le principal protocole de couche transport de la suite TCP/IP. À la différence d'UDP, TCP est une couche épaisse au dessus d'IP : TCP fournit un service de communication :

- par *flots (streams)*, ou suites d'octets de longueur arbitraire sans frontières internes ;
- *fiable (reliable)* ;
- *ordonné* ;
- avec *contrôle de flot (flow control)* ;
- avec *contrôle de congestion (congestion control)*.

Ces caractéristiques du service sont très différentes de celles du service fourni par le protocole de couche inférieure (IP) ; elles sont donc implémentées au sein du protocole TCP par, respectivement, les mécanismes suivants :

- *segmentation* dans l'émetteur et *réassemblage* dans le récepteur ;
- *acquittements* émis par le récepteur et, si besoin, *réémissions* par l'émetteur *(acknowledgement and retransmission)* ;
- réordonnement des paquets reçus ;
- mécanisme de *fenêtre coulissante (sliding window)* ;
- mécanisme de *fenêtre de congestion (congestion window)*.

3.1 Sémantique par flots d'octets

Il est important d'insister sur le fait que TCP fournit une sémantique par flots d'octets : TCP ne préserve pas les frontières des *write*, seule est préservée la suite d'octets transmise. Lorsque l'émetteur fait un seul *write*, il est possible au récepteur de recevoir les données dans deux *read* ; inversément, si l'émetteur fait deux *write*, il est possible que TCP concatène les données et les remette au récepteur en un seul *read*.

Comme la grande majorité des applications manipulent des structures de données qui ne se codent pas en un seul octet, cette sémantique est généralement inadaptée à l'application, qui doit donc accumuler les données reçues dans un tampon avant de pouvoir les analyses. D'autres protocoles (SCTP, *WebSocket*) implémentent une sémantique plus utile, par messages peu coûteux. Ce n'est malheureusement pas le cas de QUIC, qui implémente des flots multiples (trop coûteux pour envoyer un message par flot, et non ordonnés entre eux).

3.2 Format des segments

Conceptuellement, il existe plusieurs types de messages TCP : les segments de données, les acquittements, les mises à jour de fenêtre etc. Tous ces messages sont codés dans des paquets de structure identique.

Le format d'un segment TCP est indiqué à la figure 2. Un segment TCP consiste d'un *entête*

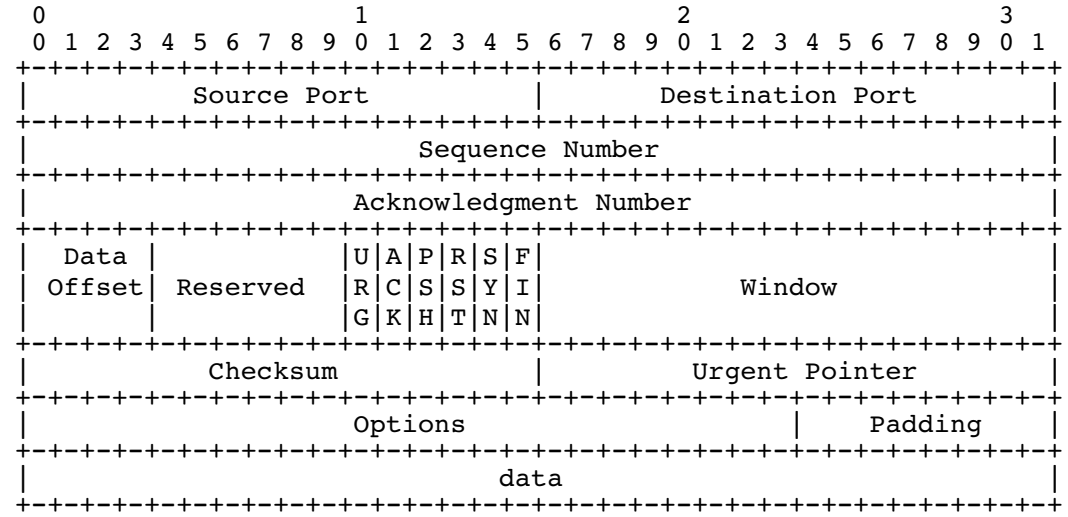

FIGURE 2 — Format d'un segment TCP

TCP suivi des données de couche supérieure (couche application), notées *data* sur la figure. Le segment (entête compris) est stocké dans la portion « données de couche supérieure » d'un paquet IP.

L'entête a une taille de 20 octets ou plus (qu'il faut ajouter aux 20 octets de l'entête IP). Parmi les nombreux champs qu'il contient, nous nous intéresserons particulièrement aux suivants :

– les port source et destination, qui jouent le même rôle que dans UDP ;

- les bits *ACK*, *RST*, *SYN* et *FIN* qui indiquent les différents types de messages ;
- le numéro de séquence, qui indique le numéro du premier octet de données contenu dans le segment et permet donc le réassemblage et le réordonnement des segments ;
- le numéro de l'octet acquitté ;
- le champ « *Window* » qui sert à mettre à jour la fenêtre.

3.3 Segmentation

TCP fournit un service de communication par flots. Comme la couche lien limite la taille des trames pouvant être transférées, et donc des paquets IP [3], un émetteur TCP *segmente* les données en *segments* suffisamment petits pour pouvoir être contenus dans des paquets IP. Inversement, le récepteur qui reçoit les données sous forme de segments discrets les *réassemble* pour les présenter à la couche application sous forme d'un flot, c'est à dire sans préserver les frontières entre les segments.

Afin de permettre le réassemblage même en présence de paquets perdus, dupliqués ou reordonnés, TCP numérote séquentiellement tous les octets d'un flot à partir d'une origine arbitraire (voir paragraphe 3.5.1 ci-dessous). Chaque segment de données contient dans le champ *numéro de séquence* de l'entête le numéro du premier octet qu'il transporte [4].

3.4 Fiabilité

TCP fournit un service *fiable*.

Intuitivement, on aurait envie de dire qu'un service fiable garantit la remise des donneés au destinataire. Cette propriété est évidemment impossible à satisfaire : si quelqu'un débranche le câble du réseau, les données n'arriveront clairement pas à destination.

On voudrait au moins pouvoir demander qu'un service fiable retourne une indication d'erreur si et seulement si les données ne sont pas remises au destinataire. Cette propriété est encore impossible à satisfaire : si une panne survient après que le paquet a été reçu mais avant qu'il soit acquitté, l'application recevra une indication d'erreur alors que les données ont bien été reçues.

On dit qu'un service est *fiable* lorsqu'il garantit que soit les données sont remises au destinataire ou une indication d'erreur est retournée à l'émetteur *ou les deux*. En particulier, un service qui jette toutes les données émises et retourne systématiquement une erreur à l'émetteur est fiable (mais pas particulièrement utile).

TCP fournit un service fiable. Pour cela, TCP demande que tout transfert soit *acquitté* par le récepteur. Un émetteur conserve les données émises en mémoire tant qu'il n'a pas reçu un acquittement ; si l'acquittement n'arrive pas au bout d'un certain temps (parce que soit les données soit l'acquittement a été perdu), les données sont *réémises*. Au bout d'un certain nombre de réémissions, TCP suppose que la connexion ne fonctionne plus et retourne une indication d'erreur à la couche application de l'émetteur.

3. Si vous êtes paumés, revoyez le modèle OSI simplifié du poly 1.
4. Oui, ça peut déborder, et oui, on gère ce cas.

3.5 Établissement et fermeture de connexion

Les mécanismes de segmentation, d'acquittement et de réémission demandent de l'état dans l'émetteur et le récepteur. Un transfert de données TCP sera donc précédé d'une phase qui établit cet état, l'*établissement de connexion*, et suivi d'une phase qui le libère, la *fermeture de connexion*.

3.5.1 Établissement de connexion

L'établissement de connexion est une phase asymétrique, qui se déroule entre un pair jouant un rôle actif, conventionnellement appelé *client* et son partenaire qui joue un rôle passif, conventionnellement appelé *serveur* [5].

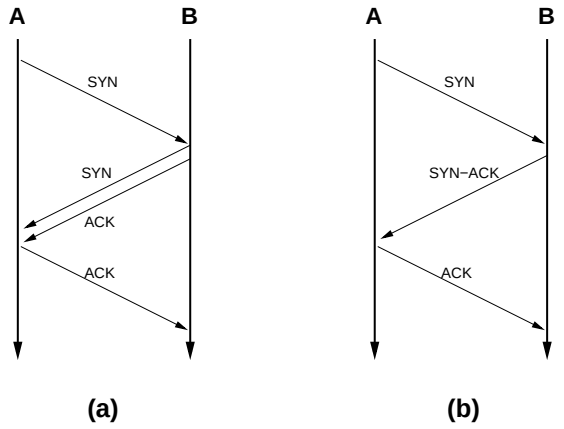

FIGURE 3 — Établissement de connexion

L'échange de paquets qui a lieu lors d'un établissement de connexion est illustré à la figure 3(a). Il consiste des messages suivants :

- le client envoie un segment SYN au serveur (« je voudrais te parler ») ;
- le serveur acquitte ce segment (« bien reçu ») ;
- le serveur envoie un segment SYN au client (« moi aussi, je veux te parler ») ;
- le client acquitte le SYN du serveur.

Si le serveur décide de refuser l'établissement de la connexion, il l'indique en envoyant un segment RST à la place du segment SYN (voir paragraphe 3.5.3 ci-dessous).

Le champ *numéro de séquence* d'un segment SYN s'appelle le *numéro de séquence initial* (ISN). Il indique l'origine de la numérotation des octets du flot : si l'ISN vaut n, le premier octet de données émis aura le numéro $n + 1$.

Optimisation : paquets combinés Comme un segment SYN et un segment ACK sont indiqués par des bits distincts dans l'entête TCP, il est possible de les combiner en un seul segment. Avec

5. Les rôles de client et serveur à la couche transport ne coïncident pas forcément avec les rôles de client et serveur à la couche application : le client de couche transport peut très bien répondre aux requêtes de couche d'application.

cette optimisation, l'échange initial consiste des trois segments illustrés à la figure 3(b). On parle parfois de « poignée de mains à trois branches » *(three-way handshake)*. L'envoi de tels paquets combinés (*piggybacked*, littéralement « à dada ») est utilisé dans toutes les phases du protocole TCP.

3.5.2 Fin de connexion

Il est possible d'arrêter de parler, mais il n'est pas possible de cesser de recevoir des segments qui ont pu déjà avoir été émis par le pair. De ce fait, la fin de connexion utilise un protocole où chaque pair déclare qu'il n'a plus rien à dire.

Lorsqu'il désire fermer la connexion [6], un pair envoie un segment FIN ; après avoir envoyé un tel segment, il n'a plus le droit d'envoyer de nouveaux segments de données [7]. Lorsque chacun des deux pairs a envoyé un segment FIN et ces segments ont tous deux été acquittés, la connexion est fermée.

En principe, ce protocole permet une situation où l'un des pairs a envoyé un segment FIN tandis que l'autre continue à envoyer des segments de données ; une telle connexion est dite *semi-fermée (half-closed)*. En pratique, cependant, les protocoles d'application sont conçus de façon à ce qu'un pair ferme la connexion dès qu'il reçoit un FIN de la part de son partenaire.

3.5.3 Interruption de connexion

Un segment RST émis par l'un des deux pairs a pour effet d'interrompre immédiatement la connexion TCP (une erreur est retournée à l'application). Ce mécanisme est utilisé en réponse à un segment SYN lorsqu'un pair ne désire pas établir une connexion (par exemple parce qu'aucune application n'écoute sur ce port), mais aussi à tout moment pour signaler une erreur, par exemple un numéro de séquence impossible [8].

3.6 Transfert de données

TCP utilise lors du transfert de données un certain nombre de techniques qui permettent d'obtenir des débits élevés sans mettre en danger l'intégrité du réseau.

3.6.1 Protocole synchrone

Une version très primitive de TCP, par exemple un *firmware* de système embarqué, peut envoyer les données de façon *synchrone*. L'émetteur commence par envoyer le premier segment de données, puis attend un acquittement ; lorsque celui-ci arrive, il envoie le second segment, et ainsi de suite (figure 4(a)).

Si un paquet est perdu, l'émetteur reémet ce paquet après un *timeout* (figure 4(b)). Comme l'émetteur ne peut pas distinguer entre la perte d'un segment de données et celle d'un acquittement (figure 4(c)), le récepteur doit être capable d'acquitter puis d'ignorer les segments de données dupliqués.

6. Par exemple après que l'application a fait `shutdown(1)`.
7. Sauf pour des réémissions, bien-sûr.
8. Ça vous donne une idée d'attaque ?

La valeur du *timeout* n'est pas fixée : elle est calculée à partir du RTT mesuré des acquittement précédents, et de la variation de celui-ci.

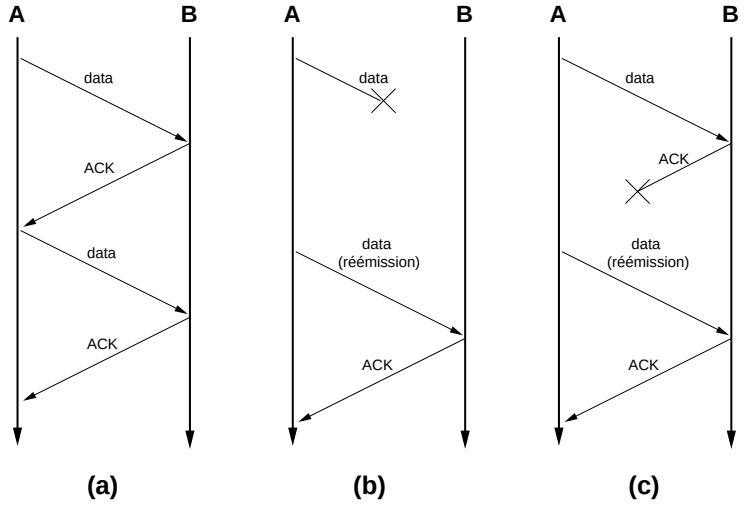

FIGURE 4 — TCP synchrone naïf

3.6.2 Protocole avec *pipelining*

Digression : débit et latence Il y a deux mesures indépendantes de la performance d'un canal de communication : son *débit* (*throughput*, parfois incorrectement appelé *bande passante* ou *bandwidth*), c'est-à-dire la quantité de données qu'on peut transmettre par unité de temps, et sa *latence*, qui est le temps nécessaire aux données pour transiter à travers la liaison.

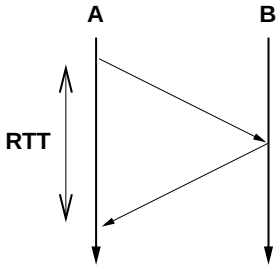

FIGURE 5 — Le *round-trip time* (RTT)

La mesure habituelle de la latence d'une liaison réseau est le *temps d'aller-retour* (*round-trip time, RTT* ou *ping*), qui est défini comme le le temps nécessaire pour envoyer un message de taille négligeable et recevoir une réponse (figure 5). Des RTT typiques de liaisons de bonne qualité sont en deçà de la milliseconde pour des communications sur le lien local, et de plusieurs dizaines de millisecondes pour des liaisons transatlantiques. Des RTT de l'ordre d'une seconde sont constatés

sur les liaisons par satellite en orbite géostationnaire (à la différence des satellites en orbite basse, dont la latence est excellente).

À la différence du débit, qui augmente avec le progrès des technologies de couche physique, les améliorations de la latence sont limitées par la vitesse de propagation du signal, et donc par la vitesse de la lumière. Or, à l'échelle d'un ordinateur, la vitesse de la lumière est glacialement lente : 30 cm/ns. Sur un réseau moderne, c'est donc la latence, qui est incompressible au delà d'un certain point, et non le débit, qui augmente avec les nouvelles technologies de couche physique, qui limite les performances.

Pipelining Le protocole synchrone présenté ci-dessus envoie au plus un segment par RTT. Avec un RTT de 30 ms et une taille maximale de segment de 1500 octets, ce protocole ne peut pas émettre avec un débit supérieur à 50 ko/s, et cela quel que soit le débit du lien [9].

Si on accepte que la latence est incompressible, il y a trois solutions à ce problème :

- augmenter la taille maximale de segment, ce qui pose d'autres problèmes (de multiplexage du lien, de capacité mémoire des routeurs, et de compatibilité avec les technologies déployées, notamment Ethernet) ;
- ouvrir plusieurs connexions TCP en parallèle, ce qui complique l'application ;
- autoriser l'envoi de plusieurs paquets avant que le premier soit acquitté ; on dit alors que ces paquets sont *pipelined*.

Dans TCP, l'émetteur envoie plusieurs segments de données ; le récepteur envoie des acquittement au fur et à mesure que les segments de données lui arrivent.

3.6.3 Contrôle de flot et fenête coulissante

Il n'est pas raisonnable d'émettre une grande quantité de données sans s'assurer que le récepteur est capable de les stocker dans ses tampons (pensez à une imprimante). Il est donc nécessaire d'utiliser un protocole qui modère le débit de l'émetteur. Comme c'est le récepteur qui sait de quelle quantité de tampons il dispose, c'est le récepteur qui impose le débit au récepteur.

TCP utilise un algorithme dit à *fenêtre coulissante*. Par définition, la *fenêtre* est l'ensemble des numéros de séquence que l'émetteur a le droit d'émettre à un moment donné ; l'émetteur maintient deux variables, le *bord gauche* et le *bord droit*, qui définissent les bornes de la fenêtre.

Le bord gauche est défini comme le plus grand numéro de séquence qui a été acquitté ; il se déplace donc vers la droite à chaque nouvel acquittement (la fenêtre se réduit). Le bord droit, par contre, est augmenté par le récepteur à chaque fois qu'il libère de l'espace dans ses tampons (par exemple parce que l'application a fait un appel à *read* ; il est défini explicitement par l'émetteur qui envoie un segment *Win* (pour *window*). Le bord droit se déplace donc vers la droite à chaque réception d'un segment *Win* (la fenêtre croît).

Naturellement, les segments *Win* sont envoyés dans des paquets combinés : le récepteur envoie un paquet *Ack-Win* pour chaque segment de données reçu.

9. Refaites le calcul, et assurez-vous d'avoir compris pourquoi.

3.6.4 Contrôle de congestion

Si le mécanisme de fenêtre coulissante permet au récepteur de modérer la vitesse d'émission, il ne permet rien de semblable aux routeurs qui se trouvent sur le chemin entre les deux pairs. Or, un routeur peut facilement se trouver dans une situation où il est débordé par le traffic qu'on lui demande de faire suivre.

Considérons par exemple la figure 6, où le lien entre les routeurs R_1 et R_2 a un débit plus faible que les deux autres liens. Le routeur R_1 doit faire suivre le traffic émis par A vers B à travers le lien lent ; or, ni A ni B ne connaissent l'existence de ce lien, et rien ne leur indique donc qu'il n'est pas souhaitable d'envoyer du traffic à la vitesse des liens rapides. Si A envoie des données à la vitesse à laquelle B est prêt à les accepter, les tampons d'entrées-sorties du routeur vont se trouver débordés, et, une fois sa capacité mémoire dépassée, celui-ci n'aura d'autre choix que de jeter des paquets. Ces paquets jetés vont mener à des reémissions, qui elles-mêmes ne feront qu'empirer la situation.

Un routeur dans cette situation est dit *congestionné* (*congested)*, et la situation dans laquelle un routeur ne fait plus suivre de données utiles du fait des reémissions dues à la congestion s'appelle *congestion collapse*.

Figure 6 — Un routeur qui risque d'être congestionné

La solution à ce problème retenue dans TCP (RFC 896) consiste à interpréter toute perte de segment comme une indication de congestion, et à y réagir en diminuant le débit dans l'émetteur. Pour cela on définit une deuxième fenêtre, la *fenêtre de congestion* ou *fenêtre de l'émetteur*, et le débit est régi par le minimum de la fenêtre de contrôle de flot et de la fenêtre de congestion.

Initialisation La fenêtre de congestion est initialisée à deux fois la taille maximale d'un segment.

Slow start Initialement, la fenêtre de congestion est régie par l'algorithme *slow start* qui, contrairement à ce qu'indique son nom, sert à faire croître la fenêtre rapidement. À chaque fois qu'un acquittement est reçu, la fenêtre de congestion est augmentée d'un segment ; lorsqu'un paquet est perdu (il n'est pas acquitté), la fenêtre de congestion est divisée par deux, mais pas plus souvent qu'une fois par RTT.

Contrairement à ce qu'il pourrait sembler, la fenêtre de congestion croît de façon exponentielle en l'absence de pertes. En effet, le nombre d'octets acquittés pendant un RTT est égal à la fenêtre de congestion, qui double donc tous les RTT.

Congestion avoidance L'algorithme *slow start* est très agressif, et mène à la perte d'un paquet tous les RTT. Lorsque la fenêtre est suffisamment grande, TCP transitionne à un algorithme moins agressif, nommé *congestion avoidance*.

Pour cela, TCP maintient une variable *ssthresh* qui estime la quantité de données qu'on peut maintenir en vol ; cette variable est mise à jour à chaque fois que TCP détecte une perte. Lorsque la taille de la fenêtre de congestion dépasse *ssthresh*, TCP passe à *congestion avoidance*.

Durant *congestion avoidance*, la fenêtre de congestion est augmentée d'un segment tous les RTT en l'absence de pertes. En présence de pertes, la fenêtre est divisée par deux une fois par RTT, comme dans *slow start*. *Congestion avoidance* fait donc croître la fenêtre de façon linéaire, d'un segment par RTT en l'absence de pertes.

3.6.5 *Bufferbloat*, AQM et sensibilité au délai

L'algorithme de contrôle de congestion décrit ci-dessus, nommé parfois TCP-Reno, repose sur l'hypothèse qu'un routeur congestionné jette des paquets au lieu de construire une queue de taille excessive. Cette hypothèse était vraie lorsque les routeurs avaient peu de mémoire ; de nos jours, cependant, la plupart des routeurs sont surprovisionnés : ils ont plus de tampons que nécessaire, et ne jettent donc plus les paquets suffisamment tôt. On dit que ces routeurs souffrent de *bufferbloat*.

La meilleure solution au problème de *bufferbloat* est de jeter des paquets avant que les tampons ne soient remplis. Une AQM (*active queue management policy*, politique active de gestion de queue) est un algorithme qui décide, pour chaque paquet, si ce paquet doit être jeté ou s'il faut l'autoriser à entrer dans la queue. Les AQM sont généralement stochastiques, elles jettent les paquets avec une certaine probabilité qui dépend par exemple de la longueur de la queue ou du débit d'un flot, ce qui leur permet d'éviter les discontinuités dans le traffic qui résultent d'un tampon qui déborde.

Si une AQM résoud effectivement le problème, on ne peut pas compter sur le fait que le routeur à l'entrée du goulot d'étranglement déploie une AQM ; des solutions de bout-en-bout sont donc nécessaires en plus des AQM. On peut remarquer qu'en présence de *bufferbloat*, la congestion se traduit par une augmentation du délai. On a donc développé des algorithmes de contrôle de congestion qui réagissent non seulement aux pertes de paquets mais aussi aux variations du délai. On peut citer TCP-Vegas, TCP-LP, LEDBAT, BRR et GCC.

Le problème de ces techniques, c'est qu'elles ont tendance à être moins agressives que TCP-Reno, et donc à se faire dominer par celui-ci : lorsque le goulot d'étranglement est partagé entre un flot TCP-Reno et un flot TCP-Vegas, le flot TCP-Vegas réduit son débit plus tôt que le flot TCP-Reno, qui occuppe donc la majorité du débit disponible. Ces techniques se trouvent donc réléguées au ghetto des transmissions de très basse priorité *(bandwidth scavenging)* [10].

3.6.6 Optimisations

Algorithme de Nagle Une implémentation naïve de TCP envoie un segment de données pour chaque appel système fait par l'application. Dans le cas d'une application interactive, par exemple un émulateur de terminal, il s'agit souvent d'écritures d'un seul octet. Comme chaque segment contient plus de 40 octets d'entêtes (20 octets d'entête TCP, 20 octets d'entête IP, plus les entêtes de couche lien), une telle application a typiquement une efficacité de 1/40, ou, inversément, une perte de 4000%.

10. Les auteurs de GCC soutiennent que leur algorithme ne souffre pas de ce problème. Je n'ai pas compris pourquoi.

La solution consiste a retarder de quelques dizaines ou centaines de millisecondes l'envoi de petits segments dans l'espoir que l'application ajoute des données qui pourront être combinées avec les données déjà présentes dans le tampon d'émission du noyau. Plutôt qu'utiliser une stratégie basée sur des *timeouts*, les implémentations de TCP utilisent une approche basée sur les acquittements, dite *algorithme de Nagle* (RFC 896). Celui-ci est défini par les règles suivantes :

- s'il n'y a pas de données en vol (toutes les données envoyées ont été acquittées), les données passées par l'application sont émises tout de suite ;
- si l'application passe suffisamment de données pour remplir un segment de taille maximale (1460 octets dans le cas d'Ethernet [11]), ces données sont envoyées tout de suite ;
- sinon, l'émission des données est retardée jusqu'à ce qu'une des conditions ci-dessus soit satisfaite.

En résumé, l'algorithme de Nagle retarde l'envoi de petites quantités de données lorsqu'il y a déjà des données en vol, ce qui fait que l'émetteur envoie au moins un segment par RTT, et plus si le débit de l'application le justifie.

Lorsque l'application est capable de prévoir son traffic et adopte une stratégie de gestion de tampons d'émission optimale, l'algorithme de Nagle ne fait que retarder inutilement l'envoi des données. Pour cette raison, il est possible de l'éteindre [12].

Acquittements retardés Les acquittement TCP sont interprétés de façon cumulative : un acquittement pour l'octet n indique la bonne réception de tous les octets $m \leq n$. Pour cette raison, il n'est pas nécessaire d'envoyer un acquittement pour chaque segment : il suffit d'envoyer des acquittement « suffisamment souvent ». Cette optimisation réduit la quantité de traffic du récepteur vers l'émetteur, ce qui est important sur les lignes à débit asymétrique, où le débit « descendant » est beaucoup plus élevé que le débit « montant ».

L'algorithme d'acquittements retardés utilisé par TCP est incroyablement primitif : un acquittement est retardé de 500 ms au plus (un *timeout* fixé, indépendant du réseau !), et on envoie au moins un acquittement pour deux segments reçus. Cet algorithme interagit très mal avec Nagle, et il est malheureusement impossible de l'éteindre.

Autres raffinements Les implémentations de TCP utilisent un certain nombre d'autres algorithmes pour rendre le traffic plus efficace. On peut citer :

- l'algorithme des *acquittements dupliqués* « dupack », qui consiste à effectuer une réémission avant qu'elle soit déclenchée par un *timeout* lorsqu'on reçoit trois acquittements dupliqués ;
- le *silly window syndrome avoidance* (RFC 896), qui évite d'envoyer de petits segments quand la fenêtre est presque pleine ;
- des extensions qui permettent d'utiliser une fenêtre de plus de 64 ko (RFC 1323) ;
- les *acquittement sélectifs* (SACK) qui permettent au récepteur d'informer explicitement l'émetteur des « trous » dans la fenêtre ;
- d'autres algorithmes de contrôle de congestion qui permettent d'éviter de réduire la fenêtre trop drastiquement en cas de pertes de paquets qui ne sont pas dues à la congestion, qui per-

11. Pourquoi pas 1500 ?
12. Sous Unix, `setsockopt(SO_NDELAY)`.

mettent de l'augmenter de façon plus agressive, ou qui permettent de minimiser les pertes de paquets (TCP Reno, NewReno, BIC, CUBIC, BBR, etc.).

4 Autres protocoles de couche transport

UDP-Lite UDP-Lite est une extension à UDP qui permet de remettre au récepteur des datagrammes même lorsqu'ils ont été partiellement corrompus. Il a été conçu pour la transmission des flux vidéo sur les liens avec pertes (les laisons radio).

UDP-Lite n'est pas très utile sur les technologies de lien modernes (WiFi, 4G), qui implémentent de la détection et de la correction d'erreur à la couche lien, et ne remettent donc jamais à la couche réseau des trames partiellement corrompues. Je ne l'ai jamais vu utilisé en production.

DCCP Les applications d'audio et de vidéoconférence ont besoin que les paquets soient remis rapidement à l'application, et sont capables de gérer les pertes de paquets (par exemple en interpolant un échantillon d'audio perdu). Pour cette raison, elles utilisent généralement UDP, ce qui les force à implémenter leur propre algorithme de contrôle de congestion. DCCP est un protocole non-fiable, basé sur des datagrammes, qui inclut un algorithme de contrôle de congestion adapté à la vidéoconférence.

Les applications de vidéoconférence modernes implémentent leurs propres algorithmes de contrôle de congestion, souvent plus performants que celui que propose DCCP, et fortement couplés au *codec* vidéo. DCCP n'a, a ma connaissance, jamais été déployé en production.

SCTP SCTP est un peu semblable à TCP : il implémente une couche épaisse au-dessus de IP. À la différence de TCP, il implémente une sémantique par messages, et permet à l'application de choisir si elle désire une sémantique ordonnée et fiable ou non. La sémantique de SCTP est proche de celle dont on typiquement besoin les applications.

La version d'origine de SCTP a été déployée, notamment pour supporter les protocoles de signalisation dans les réseaux téléphoniques et pour les protocoles d'accès à distance aux disques. Cependant, elle est incapable de traverser la plupart des NAT, et de ce fait son utilité est limitée aux réseaux privés (par exemples les réseaux de contrôle des opérateurs de téléphonie).

Depuis, SCTP a été redéfini pour être encapsulé dans UDP (un segment SCTP est stocké dans un datagramme UDP qui est lui-même stocké dans un paquet IP), ce qui lui permet de traverser les NAT et les *firewalls*. Cette nouvelle version de SCTP fait partie de la pile *WebRTC* de protocoles pour la vidéoconférence dans les navigateurs *web*, et est donc très largement déployée.

Websocket *Websocket* est techniquement un protocole qui se trouve tout en bas de la couche application, mais on peut aussi le voir comme étant en haut de la couche transport. Il s'agit d'une couche fine au-dessus de TCP, qui implémente la communication par flots de messages, et qui adapte donc TCP aux besoins typiques des applications. *Websocket* est implémenté dans tous les navigateurs modernes, ce qui évite au programmeur *Javascript* d'implémenter lui-même l'assemblage et le désassemblage des messages. C'est le protocole de communication bidirectionnelle symétrique préféré pour les applications *web* (HTTP est un protocole requête-réponse asymétrique).

QUIC QUIC est un protocole qui vise à remplacer TCP pour les besoins du protocole HTTP ; HTTP/3 est basé sur QUIC. QUIC est encapsulé dans UDP, ce qui lui permet de traverser les NAT. À la différence de TCP, qui délègue la sécurité à la couche application, QUIC intègre le protocole de sécurité TLS : il n'existe pas de version non-sécurisée de QUIC.

QUIC apporte plusieurs améliorations utiles par rapport à TCP :

- le *handshake* QUIC est combiné au *handshake* TLS, ce qui permet de gagner un RTT au début de chaque nouvelle connexion (ce qui est important pour HTTP, qui a tendance à faire de nombreuses connexions de courte durée) ;
- QUIC permet d'avoir plusieurs *flots* dans une seule connexion, ce qui permet de transmettre simultanément plusieurs objets HTTP sans perdre du temps à établir plusieurs connexions ;
- la structure des acquittements est plus explicite.

QUIC est conçu pour HTTP, et n'implémente donc pas les fonctionnalités qui ne sont pas utiles pour HTTP : il n'implémente pas la sémantique par messages, il n'implémente pas de sémantique non-fiable ou non-ordonnée. Bien qu'on puisse simuler plusieurs de ces fonctionnalités à l'aide des flots multiples, SCTP reste un protocole plus adapté aux applications qui n'ont pas la même structure que HTTP.

Le protocole *WebSocket* n'est pas défini au dessus de QUIC. Le groupe de travail *WebTransport* travaille à un successeur de *WebSocket* qui sera basé sur QUIC.