Fantasio

Zeste de savoir

Recevoir des paquets très _très_ vite avec Linux

20 mars 2021

Table des matières

Bonjour à tous!

Si vous avez déjà programmé, en hobbyiste ou en pro, il est très probable que vous ayez déjà fait connaissance avec les *sockets*, ces objets qui vous donnent accès au réseau via votre OS. Il est aussi très probable qu'elles n'étaient pas le bottleneck de votre code, et que vous n'ayez jamais eu l'**immense plaisir** d'essayer de recevoir un maximum de paquets par seconde avec.

Petit rétex sur mon aventure avec elles 🍊

1. Au commencement, il fut…

Pour exploiter une socket, on procède généralement comme ceci:

1. On ouvre la socket, avec `socket`
2. On bind, avec `bind` la socket, pour lui dire quoi écouter (une adresse IP et un port TCP par exemple)
3. On utilise `send` et `recv` pour envoyer ou recevoir des messages.

Pour recevoir des paquets on aurait donc quelque chose du genre:

```
fd = socket(AF_PACKET, ...);
bind(fd, ma_carte_réseau);
while (true) {
    recv(buffer);
}
```

Avec cette approche sur un ordinateur domestique on obtient rapidement 300kpps (pps = paquets par seconde) en single thread. Pour des paquets d'une centaine d'octets cela donne 30Mo/s, ou 240Mbits/s. Pas mal, mais pas suffisant pour exploiter ma carte 1Gbits (voire 10Gbits sur un serveur !).

Dans mon cas j'utilise une socket AF_PACKET, qui permet d'écouter les paquets sur une carte réseau directement plutôt qu'un port TCP ou UDP, mais les performances sont similaires.

2. Le partage, c'est la base

Ce "code" souffre de deux gros défauts:

— On effectue un appel à `recv` par paquet, ce qui est coûteux car c'est un appel système,
— Le noyau effectue deux copies par paquets: une première depuis la mémoire de la carte réseau vers un *buffer* interne, et une deuxième vers le *buffer* fourni pour la réception. Pas idéal.

Pour réduire l'impact de ces deux problèmes, Linux propose une option avec les sockets AF_PACKET: PACKET_MMAP.

Dans ce mode, l'utilisateur crée une zone partagée avec le noyau. Elle fait office à la fois de *buffer* interne pour le noyau et de *buffer* de réception directement pour l'utilisateur.

On y gagne sur les deux points:

— Lors d'un appel système pour la réception (effectué avec `poll`), le noyau peut copier plusieurs paquets directement dans le buffer qui sera lu par l'utilisateur: l'impact de l'appel système est donc divisé par le nombre de paquets,
— On passe de deux copies en interne à une seule.

Grâce à ce mode, on arrive à recevoir environ 700kpps, soit 560Mbits (toujours en single thread et pour des paquets de 100 octets). C'est mieux! Mais toujours pas de Gbits, et encore moins de 10Gbits...

3. Contourner le noyau ??? Vous n'oseriez pas...

Il se trouve que dans les datacenters notamment, les ingénieurs se sont rendus compte de ça depuis un moment. Plutôt que rester dépendants de Linux, ou de devoir l'améliorer (ce qui n'est pas automatique, vu qu'il faut que tout autour continue de marcher), ils ont trouvé une astuce: plutôt que de laisser le noyau gérer les cartes, c'est une application qui va en prendre le contrôle.

On parle de *Kernel bypass*. C'est ce que fait DPDK par exemple. Typiquement, c'est l'application qui choisit où la carte réseau écrit ses paquets! On a donc éliminé les deux soucis qu'on avait avant: plus d'appel système ni de copie!

Et alors là, on va *très* ***très*** vite. En fait, on atteint plus de 20Mpps toujours en single thread (16Gbits+ pour nos paquets de 100 octets !).

Pour un processeur cadencé à 4Ghz, cela représente moins de 200 cycles par paquet! En comparaison, un appel système en consomme quelques milliers.

C'est bien beau tout ça mais c'est pas super pratique de monopoliser la carte réseau de son PC! C'est un peu chiant non?

4. La revanche du noyau

Il se trouve que le *kernel bypass* a ses contraintes, notamment d'éviter le noyau (donc on ne profite pas de ses éventuelles mesures de sécurité), et on monopolise une carte réseau par application, ce qui est une contrainte en terme d'infrastructures.

Depuis environ 2013, une nouvelle fonctionnalité est apparue dans le noyau, appelée XDP. Elle consiste à pouvoir exécuter dès la réception d'un paquet sur la carte réseau, un petit programme (BPF) directement dans le kernel. Les fonctionnalités de ce programme sont assez réduites (il doit s'exécuter vite pour ne pas surcharger le noyau), et aujourd'hui se résument à dropper un paquet, le transmettre à la stack régulière, le transmettre directement sur une autre carte, ou bien l'**écrire à une adresse mémoire donnée** (pour peu que le driver le permette).

Et là vous vous en doutez, c'est le jackpot en terme de performance et de sécurité: on élimine quasiment tout le parcours du paquet dans la stack réseau de Linux, et toute copie (encore une fois si le driver le permet).

Pour rendre tout ça utilisable pour le commun des mortels, un nouveau type de socket, AF_XDP, a été introduit. Il utilise la fonction de redirection de XDP pour recevoir les paquets directement dans un buffer circulaire en userspace (comme PACKET_MMAP donc, mais plus rapide).

Un certain nombre de benchmarks ont été réalisés, et montrent que AF_XDP rivalise avec la performance de DPDK!

Voilà comment aujourd'hui vous pouvez avoir une performance réseau délirante avec des sockets utilisables pour le commun des mortels 🍊 ! Bien sûr, c'est une description très superficielle, et il y a de nombreux détails à régler pour obtenir une performance absolument optimisée, mais c'est un bon début 🍊

Quelques ressources si vous voulez aller un peu plus loin que mes descriptions haut niveau 🍊 :

- PACKET_MMAP
- AF_XDP
- Un papier sur la perf d'AF_XDP (notamment en comparaison avec DPDK)
- DPDK
- Un excellent post de blog sur la stack réseau de Linux