UCLouvain

École polytechnique de Louvain

LINFO1341 – Réseaux informatiques

Projet : Truncated Reliable Transport Protocol

Professeur : Bonaventure Olivier

Roman Hardy
Robin Paquet
Group 10

Avril 2022

1 Introduction

Pour le cours de réseaux informatiques, il nous a été demandé de mettre en place un protocole de transport fiable pour transférer des données entre deux machines distantes. Ce protocole de transport est basé sur UDP et utilise la stratégie du *selective repeat*. Pour cela, nous avons implémenté un émetteur (sender) et un récepteur (receiver) en langage C.

Nous détaillerons dans ce rapport quelques points clefs de l'implémentation, nos tests, les performances observées et les limites de nos programmes.

2 Implementation

2.1 Fenêtre de réception

Le mécanisme de fenêtre de réception permet le stockage des paquets arrivant hors séquence. L'émetteur part du principe que la fenêtre du récepteur est de taille 1, et suit l'émetteur sur base des paquets reçus. L'émetteur dicte donc la taille de la fenêtre et commence avec une fenêtre de taille 1.

Nous nous sommes inspiré de l'algorithme *additive increase / multiplicative decrease*. Lorsqu'un paquet est tronqué, la taille est divisée par 2 et lorsque plusieurs paquets sont reçus en séquence cette taille est incrémentée de 1 (sans dépasser la limite de 31).

2.2 Acquittements

Le récepteur lit les paquets entrants et tant qu'ils sont en séquence, il n'envoie pas de paquet de type ACK. Si un paquet hors séquence est reçu, un paquet de type ACK avec le numéro de séquence attendu est envoyé. Lorsque tous les paquets ont étés lus, un paquet de type ACK est envoyé indiquant le prochain numéro de séquence attendu. Tant que le récepteur ne reçoit pas le paquet attendu, il va renvoyer le même paquet ACK.

2.3 NACK

Une réception de paquet NACK (à l'émetteur) indique que le réseau est surchargé, et qu'un routeur a volontairement tronqué un paquet afin de se décongestionner. Le récepteur adapte la taille de sa fenêtre en conséquence, sur base de celle renseignée par le paquet NACK. Un paquet ACK peut aussi diminuer la taille de la fenêtre, car nous n'avons aucune garantie que le paquet NACK ne soit pas perdu en chemin. Cela peut donc mener à des instabilités dans la taille de la fenêtre si les paquets n'arrivent pas dans l'ordre d'émission.

2.4 FEC

Les 4 derniers paquets DATA envoyé sont gardés en mémoire afin de générer un paquet FEC. Une fois envoyé, nous ne retenons pas le paquet FEC ni les paquets DATAutilisés pour sa création et continuons avec le prochain. Lorsque la fenêtre est modifiée et qu'un des paquets DATA utilisé dans le FEC est supprimé (acquitté ou jeté suite à une réduction de fenêtre), nous supprimons les références gardées pour générer le FEC.

Le récepteur ne gère pas la réception des paquets FEC. Nous avons commencé à l'implémenter mais au vu du temps qu'il nous restait pour finir le projet, nous avons préféré nous concentrer sur les tests et la robustesse de ce qui était déjà en place. Nous n'avons donc pas pu tester l'envoi des FEC (aucun groupe avec lequel nous avons fait des tests d'interopérabilité n'avait implémenté les FEC).

2.5 Fermeture de la connexion

Lorsque l'émetteur arrive à la fin du fichier, il envoie un dernier paquet DATA avec le champ length mis à 0. Le récepteur voyant ce paquet sait qu'il marque la fin du fichier. Il continue normalement jusqu'à pouvoir l'acquitter (i.e. il attend d'avoir reçu tout les paquets précédent).

Si ce paquet est acquitté, la connexion peut se fermer. Après avoir renvoyé ce dernier paquet quatre fois et sans nouvelle du récepteur, l'émetteur part du principe que le récepteur a bien reçu le paquet et que la connexion est terminée de son côté. Le dernier paquet ACK a dû se perdre ou être corrompu (et donc ignoré).

2.6 Timestamp

Le champ timestamp des paquets DATA envoyés correspond au nombre de millisecondes écoulées depuis l'Epoch. Dans les paquets ACK, le récepteur copie ce champ du dernier paquet DATA reçu.

Ce champ sert à deux choses (pour l'émetteur) :

1. Savoir à quel moment le paquet a été envoyé pour décider si il faut retransmettre le paquet à nouveau
2. Estimer le temps de parcours d'un paquet (rtt), grâce aux paquets ACK envoyés par le récepteur

2.7 Retransmission

La latence du réseau indiqué dans les consignes variant dans l'intervalle $[0, 2000]$, nous avons choisi un timer de retransmission de deux secondes, soit la moitié du temps maximal aller-retour qu'un paquet peut avoir. Pour être sur de ne pas retransmettre des paquets inutilement, nous aurions du prendre un temps plus élevé et tenir compte des acquittement cumulatif (chaque paquet DATA reçu ne provoque pas nécessairement l'envoi d'un paquet ACK) dans le calcul du timer.

Au contraire, diminuer le temps de retransmission permet de compenser les paquet perdus en les renvoyant plus rapidement, mais cette solution n'est pas optimale (les paquets FEC sont destiné à cet usage).

2.8 Ouverture de connexion

Pour initier la connexion, l'émetteur envoie un premier paquet de données. Lorsque le récepteur reçoit ce paquet, il peut se connecter et commencer à renvoyer des paquets. Si le récepteur ne reçoit aucune information au bout de 10 secondes, il va s'arrêter.

La fermeture de connexion n'a pas été implémentée correctement. Nous n'avons pas assez poussé la robustesse de nos programme au cas ou l'un des deux s'arrête brusquement.

3 Partie critique

Voici quelques points qui nous semble critiques dans notre implémentation.

— Notre manque d'implémentation des paquet FEC. Une redondance dans les données aurait pu permettre d'améliorer grandement l'efficacité du transfert et aurait évité l'envoi intempestif de paquet ACK et DATA.

— Sur base des rtt calculé au cours du transfert, il aurait été intéressant de mettre à jour le timer de retransmission, pour qu'il s'adapte au réseau. La connexion aurait été beaucoup plus réactive et un réseau sans délai important aurait été utlisé a son plein potentiel.

— La retransmission des paquets ne semble pas bien implémentée. Nous avons pu observer le *call-graph* (voir Annexe) que cette fonction utilisait beaucoup de ressources, pour quelque chose d'assez trivial. Cela n'affecte peut-être pas la vitesse en fin de compte, mais il nous semble important de ne pas gaspiller des ressources.

4 Performances

Nous avons testé nos programmes avec différentes valeurs pour les paramètres *jitter, loss, error,* et *cut* du simulateur de lien, afin de voir quel impact à chacun d'eux sur le transfert. Les transferts ont tous été réalisés avec un même fichier de 188Ko et ont tous été réussis. La figure 1 montre ces résultats.

On observe que le *jitter* n'a pas beaucoup d'impact sur la transmission. Les paramètres *cut, error* et *loss* font augmenter linéairement la vitesse de transmission, mais pas avec le même facteur. Nous sommes surpris de voir une telle différence entre ces trois courbes. Un paquet perdu n'est au final pas si différent qu'un paquet corrompu, si ce n'est qu'il ne faut pas le décoder à l'arrivée (cf. CRC32 ci dessous). Un paquet tronqué va faire diminuer la taille de la fenêtre de réception, ce qui peut expliquer une différence avec des paquets simplement perdus, mais une telle augmentation du temps reste surprenante.

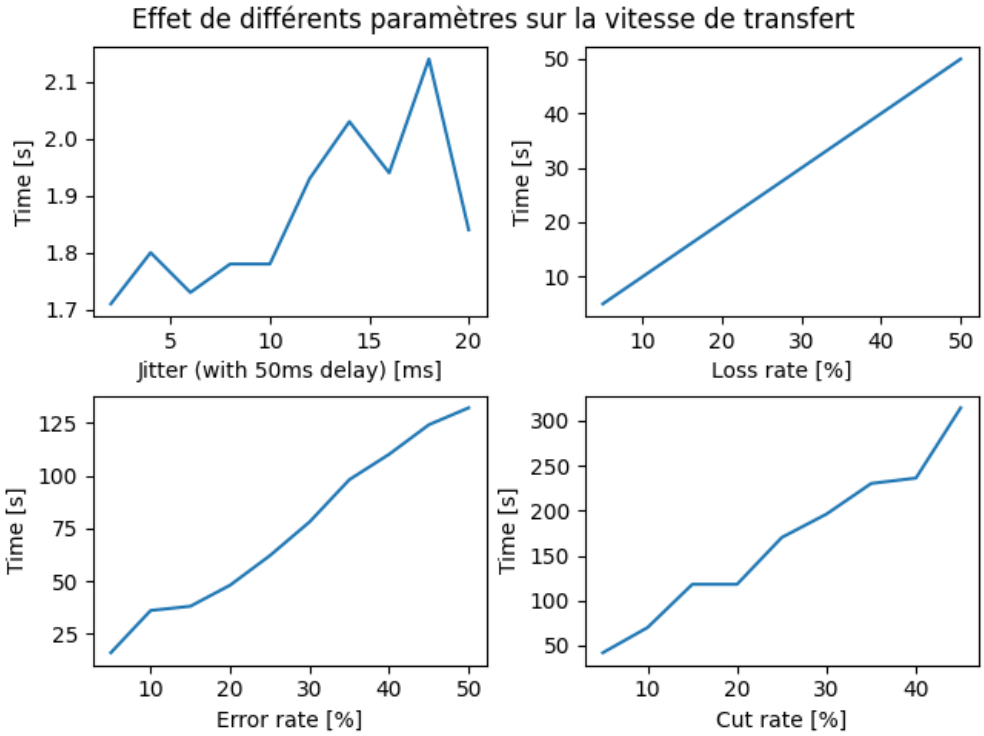

FIGURE 1 – Performances sur un fichier texte de 188Ko (note : les axes des ordonnées ne sont pas égaux)

Nous avons aussi effectué des tests sur un réseau parfait, avec des fichiers aléatoire allant jusqu'à 100Mo afin de connaître les limite de notre implémentation. Nous avons une vitesse moyenne de 13.7 Mo/s et qui reste assez stable (déviation standard de 1.29) même si l'on devine une décroissance avec la taille des fichiers.

5 Stratégie de tests

Nous avons implémentés quelques tests CUNIT pour vérifier le bon fonctionnement de notre liste chaînées. Les programme d'émetteur et récepteur ont été soumis a des *black-box* tests, basé sur le test de

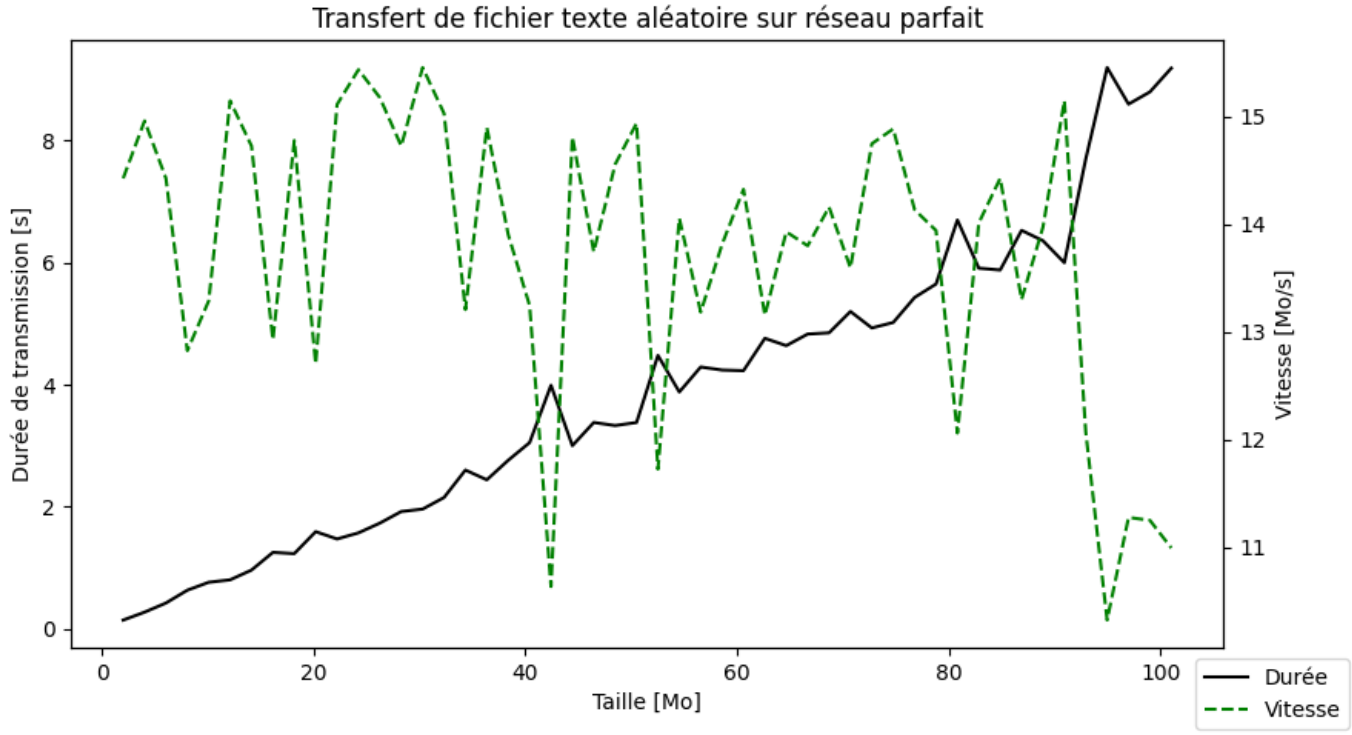

FIGURE 2 – Mesures de vitesses en cas idéal

base fournit avec le squelette du projet. Seul le résultat importe : nous comparons les fichiers envoyés et reçu avec la fonction sha256sum. Nous avons fait une suite de 8 tests avec différents fichiers et paramètres. Un test avec Valgrind nous a permis de vérifier que nos programmes ne comportaient aucune fuite de mémoire.

6 Conclusion

Ce projet nous a permis d'appliquer des concepts vu au cours et de voir l'impact de la qualité du réseau sur la transmission. Nous avons pu nous familiariser avec l'API socket et le multiplexage de fichier en C. Nous aurions aimé aller plus loin dans le projet, surtout avec l'implémentation des paquets FEC.

7 Annexes

7.1 Tests d'interopérabilité

Au cours de ce projet, des séances d'interopérabilités ont été proposées pour nous laisser la possibilité de tester l'implémentation et la compatibilité de nos programmes avec les groupes. Nous nous sommes alors entretenu avec quelques groupes pour pouvoir les améliorer et corriger nos erreurs. Ces tests nous ont été utiles car effectivement nous avions quelques erreurs dans notre code et cela s'est remarqué dès le premier test.

Pour ce premier test, nous nous sommes rendu compte que les noms de domaine ne fonctionnaient pas et donc nous n'arrivions pas à nous connecter en salle Intel avec une machine voisine. Lorsque nous avons réussi à résoudre ce problème, nous arrivions à nous connecter correctement avec l'implémentation de l'autre groupe. Notre *sender* réussissait à envoyer des paquets correctement sur un réseau parfait mais nous ne gérions pas encore l'envoi du dernier paquet de taille 0 permettant de préciser la fin de l'envoi du fichier.

Nous avons effectué une deuxième interopérabilité un peu plus tard pour tester nos corrections et nos améliorations. Lorsqu'on voulait renvoyer un paquet de type ACK après la première réception de paquet, nous avions une erreur de CRC qui était mal calculé et donc le *sender* de l'autre groupe déclenchait une erreur. En ce qui concerne notre *sender*, l'interopérabilité fonctionnait correctement.

Dernièrement, nous avons réalisé des tests supplémentaires pour confirmer la bonne implémentation de notre code. Pendant ces tests nous avons supposé que leur programme n'utilisait pas les ACK cumulatifs, car nous avons reçu autant de paquet de données que de ACK. Cependant notre code semblait correctement répondre à cet effet. Nous avons seulement rencontré un soucis dans certaines situations où le *sender* envoyait un paquet et que le *receiver* envoyait un paquet ACK pour ce même numéro de séquence au lieu d'envoyé un paquet ACK pour le paquet en attente. Lors d'un autre test, nous avons remarqué que le *receiver* envoyait un paquet ACK avec le numéro de séquence attendu 255 mais qu'il ne recevait jamais ce dernier paquet. Au lieu de ça, le *sender* envoyait des paquets en redémarrant le numéro de séquence à 0.

7.2 *Callgraph*

Note : ceci étant la première fois que j'utilise l'outil Callgraph, j'espère ne pas m'être trompé dans son interprétation. Si c'est le cas, cette section n'a plus aucun sens.

Nous avons utilisé l'outil *callgrind* de Valgrind pour essayer d'avoir une meilleure compréhension de nos programmes. Nous avons utilisé le simulateur de lien avec 50ms de délai, 5ms de jitter, 3% d'erreur, 2% de troncature et 10% de pertes, dans les deux sens. Le fichier était une image png de 280Ko. Le transfert a été réalisé deux fois, afin de ne pas influencer les résultats du *sender* (resp. *receiver*) par le fait que le *receiver* (resp. *sender*) tournait dans la machine virtuelle de Valgrind.

Il nous est apparu que le récepteur passait beaucoup de temps à attendre des paquets en provenance de l'émetteur (voir Figure 3). Cela provient sans doute d'un temps de retransmission trop élevé. La fonction traitant les données du récepteur (receive_data, voire Figure 5) montre que la calcul du CRC32 lors du décodage des paquet est conséquence (par rapport à 3).

Pour l'émetteur, on constate qu'il passe le plus clair de son temps à regarder si il peut renvoyer les paquets, ce qui met en avant l'impact de clock_gettime. Ces ressources sont à notre sens gâchées et nous aurions du implémenter plus efficacement cette partie de l'émetteur. Il existe aussi peut-être d'autres appels systèmes plus performant pour cet usage.

Enfin, nous remarquons que les appels à printf ne sont pas sans un certain coût. Cela confirme notre idée d'avoir créé une cible make—nolog qui désactives les macro de logs dans le *Makefile*.

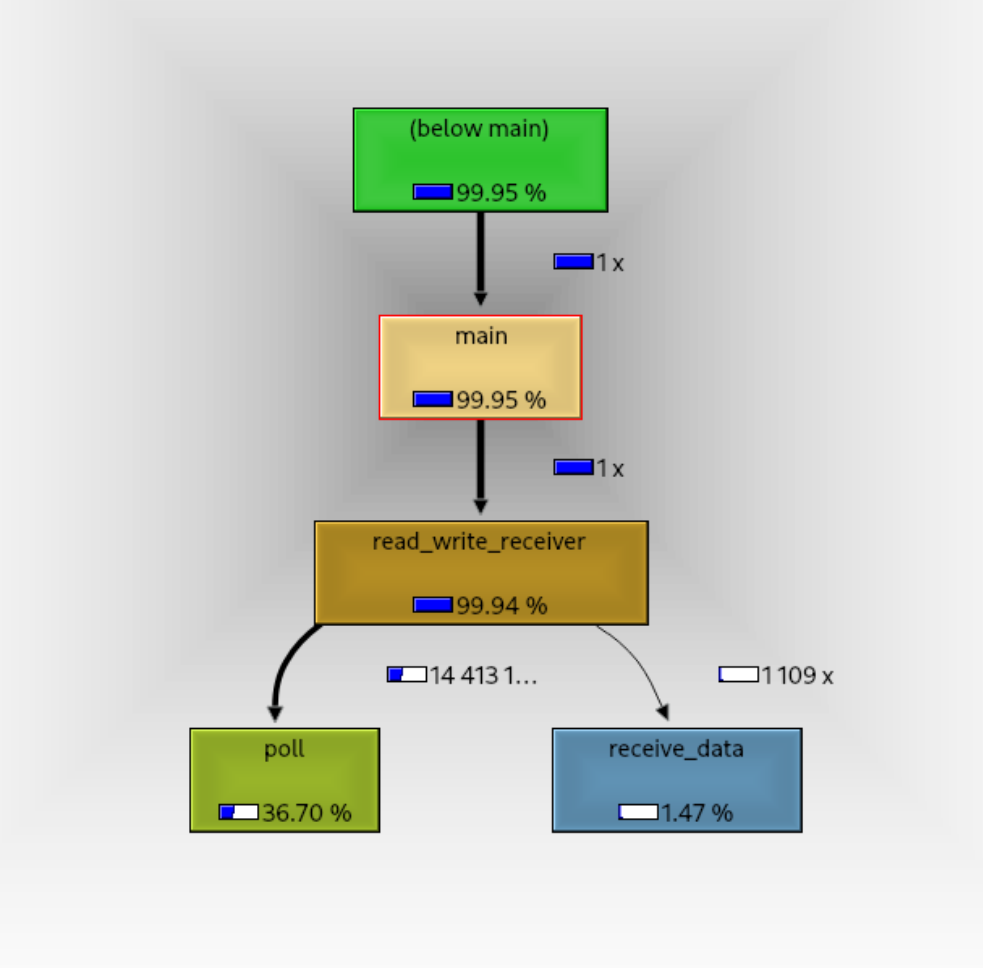

FIGURE 3 – *Callgraph* du récepteur

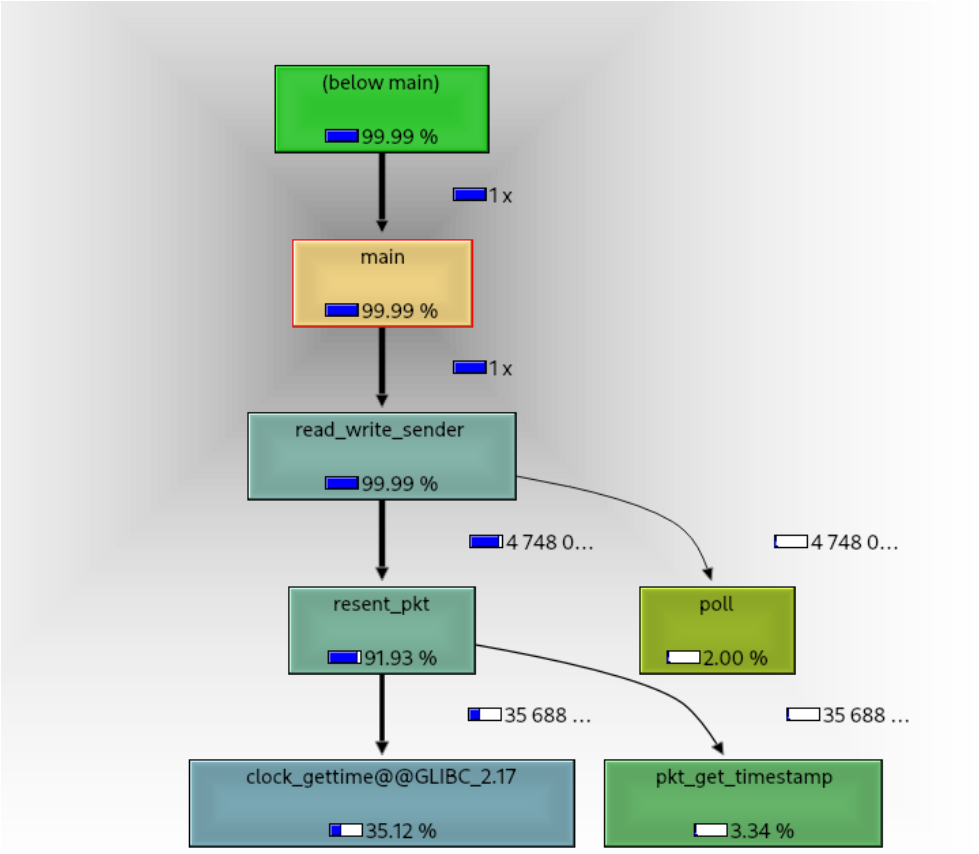

FIGURE 4 – *Callgraph* de l'émetteur

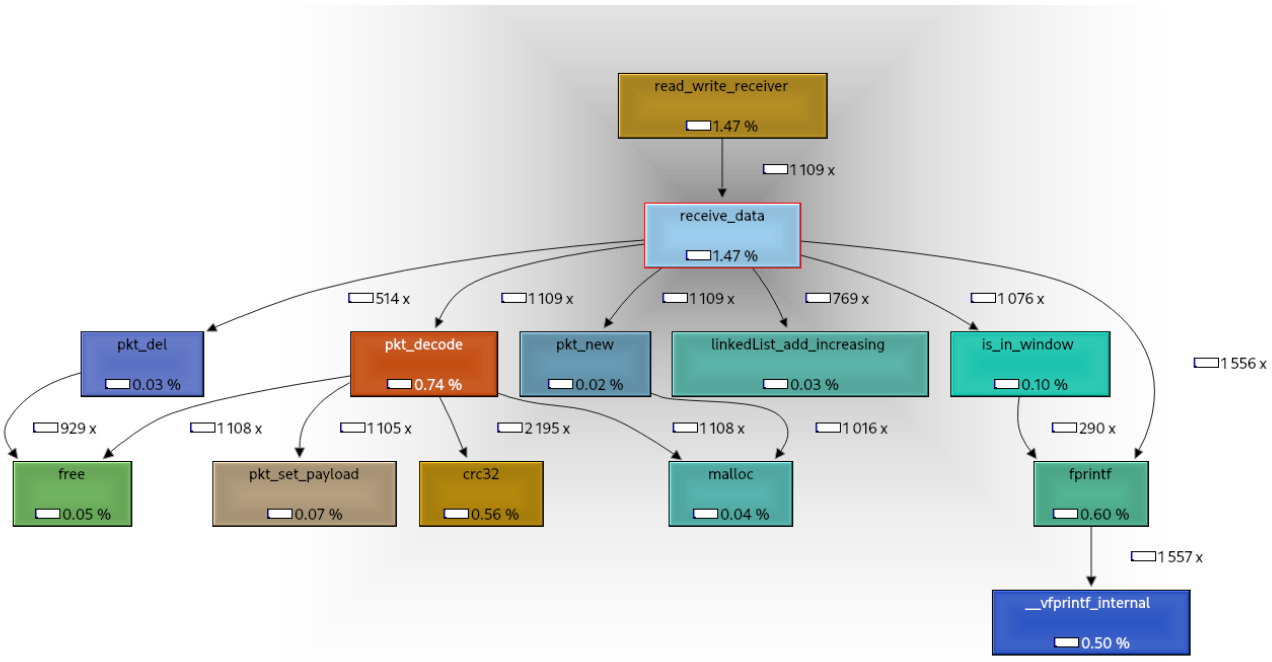

FIGURE 5 – *Callgraph* de la fonction receive_data