***BACCAR Rostom 3MIC C.***

**Document Readme du projet de BE Réseau**

**I. Remarques**

-L’archive contient 4 codes différents :

1) Le code final (mictcp.c) qui se trouve dans le dossier src. Celui-ci est bien commenté et expliqué. Ce readme est un complément pour justifier davantage quelques éléments du code

2) Une version v3 sans gestion de perte de l’ACK (mictcp\_v3.c) qui se trouve dans le dossier « versions complémentaires », déposée à toutes fins utiles. La seule différence entre cette version et la version finale de la v3 (mictcp.c) est justement l’implémentation de la gestion de perte de l’acquittement.

3) Une version v3 beta (mictcp\_v3\_BETA.c) a également été déposée, et elle se trouve dans le dossier « versions complémentaires ». Il s'agit d'une piste que j'ai considérée (et surtout qui ne marche pas) avant de trouver la solution finale codée dans la version v3 finale (mictcp.c).

4) La version v2 a aussi été déposée à toutes fins utiles (mictcp\_v2.c), également dans le dossier « versions complémentaires »,

-Les versions déposées à l'issue de chaque TP ainsi que les commentaires laissés sur Moodle sous forme de bilan et de commentaires accompagnant les dépôts détaillent les erreurs qui sont survenues lors du développement du programme ainsi que l'avancement global du projet (à toutes fins utiles).

-La majorité des tests ont été faits avec tsock\_texte

-Commande utilisée pour compiler : make

-Commandes utilisées pour l’exécution :

PUITS : ./tsock\_texte -p et ./tsock\_video -p -t mictcp

SOURCE : ./tsock\_texte -s et ./tsock\_video -s -t mictcp

**II. Plan du Readme**

1. Les choses que j'ai faites et que j'ai supprimées, et pourquoi
2. Les int socket et leur utilisation : facultatifs mais gardés
3. L'identification de l'état des PDUs + le choix du type de fenêtre glissante
4. La synchronisation des PE et PA
   1. Petite vidéo d'explication
   2. Dans le cas d'un envoi normal sans perte de PDU ou de ACK :
   3. Dans le cas d'une perte de PDU :
      1. Dans le cas d'une perte non admissible de PDU
      2. Dans le cas d'une perte admissible de PDU
   4. Dans le cas d'une perte ACK :
      1. Problème initial
      2. La piste considérée et abandonnée (v3 beta)
      3. Solution finale au problème
5. Optimisation de lecture de la vidéo (tsock\_video)
6. Bilan : ce qui marche, ce qui a le plus posé problème et avancement général du projet
   1. Ce que j'ai fait et qui marche
   2. Ce que j’ai fait et qui ne marche pas
   3. Ce que j'ai fait et qui semble peu conventionnel
   4. Ce qui a le plus posé problème
   5. Ce que j'ai fait et que je trouve génial et que je ne veux surtout pas que vous ratiez dans votre revue du code pour la correction

**III. Contenu**

**1) Les choses que j'ai faites et que j'ai supprimées, et pourquoi**

-Vu que la phase d'établissement de connexion n'est pas à traiter cette année, j'ai enlevé du code la manipulation des states du socket (ESTABLISHED, IDLE...) car après tout, ces états-là serviraient à vérifier l'établissement de connexion.

-Même chose pour les header.syn et header.fin

-Même la variable globale sock n'est pas beaucoup utilisée finalement

**2) Les int socket et leur utilisation : facultatifs mais gardés**

-D'après ma compréhension du sujet et les réponses que j'ai pu avoir pendant les séances du BE, ces variables passées en paramètres serviraient aussi à établir la phase de connexion, chose qu'on n'a pas à faire cette année. Leur utilisation est alors facultative. Néanmoins, je les ai quand-même gardées simplement pour utiliser les variables passées en paramètres (cela évite des warnings selon le compilateur). Et leur utilisation consiste simplement à s'assurer que la description du socket correspond bien aux int socket.

**3) L'identification de l'état des PDUs + le choix du type de fenêtre glissante**

L'implémentation du concept de fiabilité partielle nécessitait un traçage continu de la réception ou non des PDUs envoyés. L'idée est donc de garder dans un tableau l'état de réceptions de ces PDUs envoyés.

La question qui vient tout de suite à l'esprit est, si on envoie un grand nombre de PDUs, est-ce qu'il faut gérer un tableau de taille "infinie" ou pas.

J'ai opté pour une solution avec un tableau de taille finie avec un parcours circulaire.

-Comment ça marche :

On définit la taille N du tableau. Ce tableau contiendra toujours N PDUs mais l'avancement de l'indice du tableau se fait PDU par PDU.

Le parcours circulaire veut simplement dire que l'échantillon de N PDUs analysés est pris en faisant un modulo de la taille N du tableau.

Si pour un certain échantillon le nombre de PDUs perdus respecte la perte admissible on accepte la perte de PDU. En revanche si cette perte de PDUs dépasse la perte admissible, on renvoie le(s) PDU(s) concernés afin de respecter la contrainte de perte admissible.

Le tableau contiendra 1 si le PDU a été reçu et 0 sinon.

Remarque : il n'est nécessaire de recalculer le pourcentage de perte que si on perd un PDU (timer expiré). En effet, si la perte est admissible, on ne renvoie pas le(s) PDU(s) mais si la perte est non admissible, on ne renvoie que le(s) PDU(s) qui posaient problème, de façon que quand on les renvoie, la perte admissible est tout juste respectée.

Cela signifie que si l'on reçoie bien un PDU, il est inutile de recalculer le pourcentage de perte puisque si on est passé à un autre PDU, c'est que les problèmes de perte précédents ont déjà été réglés. Le fait de ne pas recevoir un PDU peut changer le taux de perte mais recevoir un PDU ne change rien à la perte.

Ainsi, la mise à jour de la fenêtre (0 ou 1 pour le PDU) se fait pour chaque PDU qu'il soit perdu ou pas, mais le calcul des pertes sur l'échantillon de N PDUs ne se fait que lors d'une perte d'un PDU.

Il est important de se rappeler que le calcul des pertes ne concerne que l'échantillon des N PDUs analysés à un temps donné, échantillon qui évolue au fur et à mesure qu'on envoie des PDUs.

-Pourquoi avoir choisi un parcours circulaire avec une taille fixe pour le tableau et non pas un stockage infini de tous les PDUs envoyés :

Le parcours circulaire permet de ne pas avoir à prendre en compte les erreurs faites dans le passé et n'analyse l'état de perte des PDUs que localement (tableau de taille N).

Cela veut dire par exemple que si l'on commet plusieurs erreurs au début de l'exécution et que cette perte se stabilise dans le futur, avec un parcours circulaire il est possible d'atteindre à un moment donné une perte calculée de 0% alors que c'est impossible si on prend en compte tous les états de tous les PDUs envoyés.

D'une façon générale, le parcours circulaire permet de renvoyer moins de PDUs sans pour autant sacrifier la qualité du message à envoyer. Si on avait utilisé un tableau qui stocke les états de tous les PDUs (taille "infinie"), on serait toujours en train de payer pour les erreurs commises qui n'ont rien avoir avec les futurs PDUs envoyés et qui pourraient même gêner leur envoi. Cette technique et bien meilleure pour le contenu en streaming (Imaginez si la qualité de votre vidéo au milieu du visionnage était influencée par la qualité de la vidéo au début. On aurait pu avoir une meilleure définition localement mais du fait qu'on paye pour les erreurs du passé on se retrouve avec une qualité moindre).

Il est important d'analyser l'état de perte localement.

**4) La synchronisation des PE et PA : (PE et PA sont des variables globales et valent initialement 0)**

**4.1) J’ai réalisé une vidéo pour expliquer davantage le 4ème point "synchronisation des PE et PA" dans sa globalité dont voici le lien (privé) :**

**https://youtu.be/DTILYA85970**

**4.2) Dans le cas d'un envoi normal sans perte de PDU ou de ACK**

Dans ce cas, voilà comment se déroule l’exécution :

**CÔTE SOURCE**

-Le pdu.header.seq\_num reçoie la valeur de PE

-PE est mis à jour

-On envoie le PDU

-On reste en attente d'un ACK

-Lorsqu'on reçoie le ACK avec le timer et le numéro de séquence respectés, on met à jour l'état de réception de ce PDU dans le tableau

**CÔTE PUITS**

Si on reçoie quelque chose, la fonction process\_received\_PDU est automatiquement appelée :

-On vérifie que le numéro de séquence du PDU reçu correspond bien au PA

-Si c'est le cas (c'est le cas ici), on met à jour le PA

-Dans les deux cas (PE=PA et PE!=PA), on renvoie un ACK. Le cas ou PE!=PA est de toute façon traité par la fonction send qui affichera un message signalant le mauvais numéro de séquence (La fonction process\_received\_PDU aussi d'ailleurs). On aura donc le message « mauvais numéro de séquence » à la fois au niveau de la source et du puits si cette erreur survient.

**4.3) Dans le cas d'une perte de PDU :**

**4.3.1) Dans le cas d'une perte non admissible de PDU**

Dans ce cas, on se contente simplement de renvoyer le PDU perdu une deuxième fois. Cela revient donc à faire un envoi normal dont l'exécution a déjà été détaillée dans le point 4.2).

Remarque : une perte d'un PDU est normale et attendue car le programme est fait pour prendre en compte le fait que ces messages soient perdus et réagir d'une façon ou d'une autre selon la situation. Par contre, les pertes qui posent problème sont les pertes de ACK qui ne sont pas contrôlées comme les pertes de PDUs. Ceci sera d'avantage détaillé dans le point 4.4).

**4.3.2) Dans le cas d'une perte admissible de PDU**

Dans ce cas, voilà comment de déroule l’exécution :

**CÔTE SOURCE**

-Le pdu.header.seq\_num reçoie la valeur de PE

-PE est mis à jour

-On envoie le PDU

-On reste en attente d'un ACK

-Le timer est expiré. On ne sait pas s'il s'agit d'une perte de PDU ou de ACK mais dans ce cas on suppose que c'est une perte de PDU. (La source ne le sait pas mais la personne qui teste le programme avec 2 terminaux le sait puisqu'elle peut attester ou non de la réception du message au niveau du puits. Il y a aussi l'affichage des messages "Pertes de paquet" qui veut dire qu'il s'agit d'une perte du PDU si on est côté source et d'une perte de ACK si on est côté puits).

-Dans ce cas, on met à jour le PE pour faire en sorte de ne pas être décalé pour le PDU suivant puisque du côté du puits rien ne s'est passé. On fait comme si de rien n'était en quelque sorte et réattribue à PE la valeur qu'elle avait avant l'envoi du PDU.

**CÔTE PUITS**

-Rien ne se passe. La fonction process\_received\_PDU n'est pas appelée car le PDU ne parvient pas au puits.

**4.4) Dans le cas d'une perte ACK :**

**4.4.1) Problème initial**

Dans ce cas, voilà comment se déroule l’exécution :

**CÔTE SOURCE**

-Le pdu.header.seq\_num reçoie la valeur de PE

-PE est mis à jour

-On envoie le PDU

-On reste en attente d'un ACK

-Le timer est expiré. Cette fois-ci on suppose que c'est une perte de ACK.

-Dans ce cas, on met à jour le PE (En effet, le programme met à jour le PE en cas de timer expiré et ne distingue pas (coté source) le cas de perte PDU et perte de ACK).

**CÔTE PUITS** (c'est là que le problème survient)

Puisqu'on parle d'une perte de ACK, ça veut dire que le PDU est bel est bien parvenu au puits. La fonction process\_received\_PDU sera donc appelée et voilà comment se déroule l'exécution :

-On vérifie que le numéro de séquence du PDU reçu correspond bien au PA

-Si c'est le cas (c'est le cas ici), on met à jour le PA

-On envoie le ACK (qui sera perdu en cours de route).

Problème : on se retrouve donc dans une situation où, côté source le PE a été mis à jour car le timer a expiré, et côté puits le PA a aussi été mis à jour car le message est parvenu au puits. On aura donc un décalage entre PE et PA pour le prochain PDU qui sera envoyé.

**4.4.2) La piste considérée et abandonnée (v3 beta)**

Au vu de ce problème de décalage entre PE et PA, j’ai exploré plusieurs pistes. Mais dans tous les cas, le problème ne pouvait être réglé qu’en diminuant la probabilité de perte de l’ACK et non pas avec un code qui marche à 100% car le fait est qu’il reste impossible, du côté de la source, de savoir de quel type de perte il s’agit. La solution la plus efficace serait donc de jouer sur les probabilités et non pas de produire un code qui marche à tous les coups.

Avec cette idée en tête, j’ai initialement développé une version « beta » (v3 beta) qui envoie un ACK de l’ACK. Pour ce faire, j’ai implémenté exactement la même logique d’envoi de PDU et d’attente d’un ACK utilisée dans la fonction send, mais cette fois-ci dans la fonction process\_received\_PDU avec l’envoi d’un ACK et l’attente d’un ACK de l’ACK. Comme pour la fonction send, si l’ACK de l’ACK n’est pas reçu, on renvoie l’ACK.

Il s’agissait donc d’appliquer le même concept d’acquittement pour un message mais deux fois pour diminuer les chances de perte de l’ACK.

Cette version a malheureusement posé plus de problèmes qu’elle n’en a résolus. En effet, je n’ai pas pu corriger les erreurs liées à l’exécution. Néanmoins, je l’ai quand-même incluse parmi les fichiers déposés à toutes fins utiles.

**4.4.3) Solution finale au problème**

J’ai finalement pensé à envoyer plusieurs ACK en même temps. Cette technique peut sembler simple et banale mais elle est en effet très efficace selon les tests que j’ai faits : l’envoi de plusieurs ACK pour un seul et même message ne perturbe en effet pas le programme et aide à réduire les chances de perte de l’ACK. En effet, plus on envoie d’ACK pour un seul et même message, plus on augmente les chances qu’au moins un ACK parvient à la source. Et d’après les tests faits, le nombre d’ACK devant être envoyés pour s’assurer de la réception de l’ACK dépend (bien naturellement) du loss rate mis au début. C’est pour cela que j’ai effectué plusieurs tests avant de conclure qu’une relation générale de X ACK envoyés pour un loss rate de 10\*X convenait parfaitement au programme. J’aurais également pu envoyer directement N ACK et ne pas lier le nombre d’ACK envoyés au loss rate (N valant 10 dans mon programme et 10 ACK correspondant à un loss rate de 100 donc il y a très peu de chance que tous les ACK soient perdus d’un point de vue de probabilité) mais le fait de lier cela au loss rate semblait meilleur d’un point de vue logique (et ça marche surtout).

Remarque : l’envoi ne plusieurs ACK en même temps pour un seul message est possible notamment car on a un timer de 100ms dans le programme, ce qui est largement suffisant pour faire passer plusieurs ACK.

**5) Optimisation de lecture de la vidéo (tsock\_video)**

En ce qui concerne tsock\_video : pour que la lecture de la vidéo soit fluide, il faut bien évidemment diminuer le loss rate d'une part, et aussi le timer d'autre part.

En effet, du fait que le timer dure 100ms pour mon programme, cela a beaucoup ralenti l'avancement de la vidéo. En réalité cette valeur de 100ms n'est là que pour s'assurer que le PDU / ACK a bien été perdu et est très pratique pour les tests sur tsock\_texte. En revanche, pour une lecture fluide de la vidéo, nous n'avons pas besoin d'inspecter l'envoi et la réception de chaque PDU dans les terminaux (ce qui est impossible d'ailleurs vu le nombre important de PDUs envoyés par la vidéo). On peut donc diminuer ce timer jusqu'à 5ms et même moins. L'attente des PDU / ACK est en réalité très inférieure à la valeur de sécurité de 100ms.

Aussi, pour une lecture fluide, on peut enlever les printf qui prennent également du temps à s'afficher.

**6) Bilan : ce qui marche, ce qui a le plus posé problème et avancement général du projet**

**6.1) Ce que j'ai fait et qui marche**

-La v2 marche très bien (mictcp\_v2.c) : tous les messages envoyés sont reçus et ce même en cas de perte de PDU(s) puisqu'on renvoie le PDU tant qu'on ne reçoit pas de ACK.

-La v3 finale marche très bien aussi (mictcp.c) : le concept de fiabilité partielle est bien implémenté. A partir d'une certaine perte calculée sur l'échantillon de N PDUs analysés, on renvoie le PDU qui pose problème. Pour les pertes admissibles, on ne renvoie pas le PDU. La gestion de l’ACK perdu marche bien aussi.

-La fenêtre glissante est correctement mise à jour pour un échantillon N de PDUs analysés selon les tests.

**6.2) Ce que j’ai fait et qui ne marche pas**

-La version v3 (mictcp\_v3.c) ne marche pas car, comme vu précédemment, il y a le problème de décalage entre PE et PA : il manque la gestion de perte de l’ACK. Typiquement, pour un loss rate de 30% et une perte admissible de 30% également, en envoyant comme premier message ‘A’, et comme deuxième message ‘B’ etc, c’est au niveau du ‘G’ que l’ACK est perdu et que le timer expire du côté de la source. Ainsi, pour le message suivant envoyé (‘H’), il y aura un problème de décalage entre PE et PA.s

**6.3) Ce que j'ai fait et qui semble peu conventionnel**

L’idée d’envoyer un ACK d’un ACK semblait très peu conventionnelle. Après tout, il s’agissait de réitérer le processus mais une deuxième fois afin de réduire les chances de pertes. La question qui revenait à chaque fois est : pourquoi deux fois seulement, pourquoi pas 3 ou même X fois. Et c’est de là qu’est venue l’idée d’envoyer X ACK et de lier ça à la probabilité de perte (loss rate). C’est en effet une idée qui permet d’exécuter X fois l’idée citée précédemment mais sans compliquer le code (il s’agit d’une simple boucle for).

**6.4) Ce qui a le plus posé problème**

-Le début du projet a posé problème : je ne savais pas exactement ce qu’il fallait ne pas prendre en compte pour ne pas coder la phase d’établissement de connexion. Il y avait notamment des structures dans le fichier mictcp.h que je pensais devaient forcément être utilisées.

-La disjonction des cas entre le timer qui expire, le numéro de séquence qui n’est pas le bon etc.

-Le règlement du problème de décalage entre PE et PA (d’où la longueur du point 4) et la réalisation de la vidéo explicative).

**6.5) Ce que j'ai fait et que je trouve génial et que je ne veux surtout pas que vous ratiez dans votre revue du code pour la correction**

-Toute la partie 4) du Readme (et surtout la gestion de l’ACK et les pistes explorées).