**

*UNIVERSITÉ DE SHERBROOKE*

*DÉPARTEMENT D’INFORMATIQUE*

*Rapport du TP1*

*Dans le cadre du cours:*

*IFT585 - Télématique*

*Travail présenté*

*à*

*M. Bessam Abdulrazak*

*Par*

*Guillaume Cléroux - 20 076 819*

*Ala Antabli - 20 012 727*

*Aida Behrouzi- 21 003 129*

*Le 11 juin 2022*

## 

## Instructions sur la construction du projet

Pour avoir le projet complet, vous pouvez accéder à toute la banque de code à ce lien GitHub: <https://github.com/gcleroux/IFT585-TP1/>

Prérequis pour la construction:

* Un environnement Linux
* Cmake 3.12+
* Make 4.3
* C++ 11 et plus

La construction du projet se fait facilement dans un environnement linux moderne. Premièrement, pour obtenir les fichiers du projet, vous devez entrer dans le terminal la commande suivante:

$ git clone <https://github.com/gcleroux/IFT585-TP1.git>

Puis vous devez aller dans le répertoire du projet:

$ cd IFT585-TP1

Lorsque vous êtes dans le projet, vous pouvez construire l’exécutable avec les commandes

suivantes:

$ cmake .

$ make

L'exécutable se trouvera dans le répertoire Output/ avec les fichiers de configurations fournis. Nous avons deux fichiers à transmettre entre les machines, soit cat.jpg et alice.txt. Vous pouvez modifier et ajouter vos propres fichiers dans le répertoire Output/ et les transmettre en remplaçant les fichiers transmis dans les fichiers ComputerXFiles.txt. Plus d'informations sur cette procédure sont disponibles dans le fichier de documentation suivant: <https://github.com/gcleroux/IFT585-TP1/blob/main/docs/docs-simulateur.pdf>.

Pour faire un test des fonctionnalités, nous recommandons de ne pas changer de fichier dans la configuration, et d’utiliser la commande suivante pour exécuter le test:

$ ./Simulateur -c 2 -f 1 -g conf.txt

Le transfert devrait s'exécuter de façon adéquate. SI un problème survient lors de l'exécution de cette commande, vous pouvez réessayer (les résultats sont un peu inconsistants).

Une vidéo démonstrative de notre projet peut être consulté au lien suivant: [2022-06-11 22-30-48.mp4](https://usherbrooke-my.sharepoint.com/:v:/g/personal/cleg1805_usherbrooke_ca/Ed8BAzQi4KBCn1gMUeqtJ1sBbumFZ41sZAY-ZMoJpPcBlA?e=5XptkZ)

## I- Explications du Code des fenêtres

#### Ce qui fonctionne bien:

##### Gestion de la plupart des événements

Le timeout de transmission fonctionne très bien ainsi que la gestion des timers ACK. Lorsque nous n’envoyons pas un ACK en piggybacking, notre implémentation ne semble pas inclure de bugs.

##### Liaison bidirectionnelle entre les ordinateurs

Nous sommes capables d’envoyer avec des fenêtres de réception et d’émission de façon fiable et stable entre deux ordinateurs. Les fichiers de taille différentes ne semblent également pas causer de problème lors de la transmission puisque nous avons testé avec un fichier de 35 ko et 145 ko.

##### Implémentation d’un protocole à fenêtres de taille 1 sans piggybacking

Bien que notre soumission finale implémente un protocole de taille n avec du piggybacking, nous avons atteint un niveau de stabilité et de fiabilité maximale avec cette implémentation. Notre code produit semble sans bugs et n’a aucune limite de taille de fichier. Le code de ce protocole peut être obtenu dans notre historique de projet au lien suivant: <https://github.com/gcleroux/IFT585-TP1/tree/a24b282bd5e602220fdc971c561b9227ca3cb307>.

#### Ce qui fonctionne moins bien:

##### Envoi de fichiers de moins de plus 145 ko

Le protocole est capable de transmettre avec succès de façon constante des fichiers d’une taille de moins de 145 ko. Par contre, lorsque nous dépassons cette taille, il semblerait que notre implémentation contient certains bugs qui nous empêchent de transmettre le reste du fichier. En conséquence, seulement les premiers 145 ko sont disponibles dans le fichier transmis et le reste est pris dans la boucle d’exécution.

##### Piggybacking pour l’envoie de ACK

Le piggybacking implémenté dans le protocole a fenêtres de taille n semble bien fonctionner. Du moins pour le début de la transmission, tous les échanges se produisent facilement et de façon fiable. Le problème se produit encore une fois sur des fichiers volumineux laissant supposer que notre implémentation n’est pas complètement réussie.

#### Ce qui fonctionne ne fonctionne pas:

##### Quitter l’exécution de façon propre

Bien que notre protocole soit capable d’envoyer des fichiers de 145 ko, nous ne sommes pas en mesure de quitter l’exécution de façon propre. En fait, nous obtenons un segfault lorsque nous tentons de couper la communication entre les machines. Ce résultat est survenu lorsque nous avons tenté d’ajouter les fenêtres de taille n et le piggybacking.

Analyse du code

L'un des inconvénients du protocole d'arrêt et d'attente était que, comme l'expéditeur attendait de recevoir le message de confirmation de trame le plus récent avant d'envoyer la trame suivante, il était inactif tout le temps. Mais dans le protocole de fenêtre coulissante, le temps d'attente est également utilisé pour envoyer des données. Dans le protocole de fenêtre coulissante, "fenêtre" fait référence à la mémoire tampon utilisée pour stocker les images. Dans ce protocole, deux fenêtres ou deux mémoires tampons sont définies : l'une est la fenêtre émettrice et l'autre est la fenêtre réceptrice. Le nombre de trames que l'expéditeur doit envoyer est stocké dans la fenêtre de l'expéditeur. Si la taille de la fenêtre de l'expéditeur est supposée être n, les trames sont numérotées de 0 à n-1, et l'expéditeur envoie n trames à chaque fois au lieu d'une. La fenêtre du récepteur contient également le nombre de trames à recevoir.

Contrairement au protocole stop-and-wait, dans lequel la méthode de communication était semi-duplex, dans le protocole de fenêtre coulissante, la communication expéditeur-récepteur est complètement duplex, ce qui signifie qu'ils peuvent s'envoyer des données en même temps. Il s'agit d'une caractéristique importante du protocole de fenêtre coulissante.

Dans le protocole à fenêtre coulissante, l'émetteur stocke un certain nombre de trames en fonction de la taille de sa mémoire tampon ou fenêtre dite. Les trames de la mémoire tampon sont envoyées consécutivement. C'est-à-dire que l'expéditeur envoie la première trame, mais n'attend pas le message de confirmation du récepteur, mais envoie également les trames suivantes. Lorsque le récepteur reçoit le premier message de confirmation de trame du récepteur, l'expéditeur, qui est sûr qu'il a atteint sa destination, efface cette trame de sa mémoire tampon (fenêtre). Par conséquent, un espace est ouvert pour enregistrer une nouvelle trame et l'émetteur enregistre une nouvelle trame dans sa mémoire tampon. Dans ce cas, la fenêtre dite de l'émetteur (mémoire tampon de l'émetteur) glisse vers l'avant pour ajuster le nouveau cadre. De même, lorsque le deuxième message de confirmation de trame atteint l'expéditeur, l'expéditeur efface également la deuxième trame de sa mémoire, et la fenêtre glisse d'un pas de plus pour enregistrer la nouvelle trame. L'expéditeur envoie également d'autres trames au destinataire. Ce processus se poursuit jusqu'à ce que toutes les trames soient envoyées à la destination.

Ainsi, dans le protocole de fenêtre coulissante, l'expéditeur et le destinataire ne passent aucun temps d'attente et échangent des données tout le temps.

Procédure d’exécution

Dans notre code, les informations sont transférées à l'aide de l'algorithme de fenêtre de taille n à rejet sélectif. L'ordinateur A récupère les paquets de la couche réseau et produit des trames afin de les transférer à l’ordinateur B. Lorsque l’ordinateur B reçoit les trames, il les envoie directement (supposant que la trame reçue est celle qu’il attendait) à sa couche réseau.

L’ordinateur A garde les trames qu’il envoie dans une fenêtre tampon et initialise un minuteur pour obtenir une réponse de l’ordinateur B. Si ce minuteur arrive à échéance, l’ordinateur A peut conclure que la trame a été perdue dans la transmission et qu’il devrait procéder au renvoi.

Après l’envoi d’une trame, l’ordinateur A n’attendra pas activement le retour de l’ordinateur B. Il va plutôt envoyer les trames subséquentes jusqu’au remplissage de son tampon d’envoi. Lorsque son tampon est plein, il ne peut pas envoyer d’autre trame et il attend un ACK afin de libérer des ressources dans son tampon et envoyer à nouveau une trame.

Lorsque l’ordinateur B reçoit une trame qui est différente de celle qu’il attendait, deux scénarios peuvent se produire. Si aucun NAK n’a été reçu préalablement, cela signifie hors de tout doute que nous avons une erreur de communication entre les machines et qu’une trame a été perdue. Dans ce scénario, l’ordinateur B signale qu’on doit envoyer un NAK à l’ordinateur A pour renvoyer la trame. Or, si un NAK était déjà présent, la trame qu’on vient de recevoir est une trame envoyée avant que l’ordinateur A ait reçu notre message d’erreur. Comme nous avons du rejet sélectif, nous devons garder cette trame dans notre tampon puisqu’elle demeure valide. Cependant, nous ne pouvons pas envoyer cette trame directement à la couche réseau comme nous devons garantir la livraison des trames dans l’ordre.

Le NAK sera levé lorsque l’ordinateur B recevra la trame manquante originale (la trame avec le numéro de séquence circulaire le plus bas). Quand le NAK est levé, l’ordinateur B va alors parcourir sa fenêtre tampon et transférera toutes les trames reçues qu’il a emmagasiné à sa couche réseau.

De plus, l’envoie de ACK dans notre code se fait avec du piggybacking. Cela nous permet d’avoir une meilleure utilisation du médium plutôt que d'envoyer des ACK séparément.

## 

## II- Explications du Code de Hamming

#### Ce qui fonctionne bien:

Le codage fonctionne très bien. La fonction du codage transforme bien le message reçu en code de Hamming, en ajoutant les bits de redondance dans les places désignées (ayant les indices 2n). Le décodage fonctionne bien en absence de bruit. La fonction prend le code et on génère le message avec un booléen qui indique la présence ou l'absence d' erreurs de transmission.

#### Ce qui fonctionne moins bien:

Le décodage ne marche pas bien dans le cas du présence d’un bruit. Plus le pourcentage du bruit augmente, plus le message corrigé contient des erreurs.

#### Ce qui fonctionne ne fonctionne pas:

Dans le cas d’un pourcentage de bruit assez élevé, on ne peut plus compter sur le décodeur pour la correction du message. Beaucoup de codes reçus seront mal corrigés.

Analyse du code

Dans le code de Hamming, la relation 2m = n + 1 est établie :

n = nombre de bits dans un bloc

m = nombre de bits de contrôle par bloc (m = n-k)

k = nombre de bits d'information dans le bloc

* Codage:

1- Dans un premier temps, on obtient le nombre de bits de contrôle en fonction de ladite longueur en fonction de la longueur du bloc et du nombre de bits d'information. Les bits du bloc sont numérotés à partir de la droite d'un nombre.

2- Si le nombre de bits (total de contrôle et d'information) est devenu individuel. Ajouter un zéro à droite. Il est à noter que ce bit ne participe pas à la numérotation.

3- On remplit les bits qui sont des puissances de deux (... et 1,2,4,4,8) avec des bits de contrôle et le reste des bits (... et 3,5,6,6,7) avec des bits d'information. Chaque bit de contrôle est rempli de manière à ce que la parité du bit de contrôle et d'un ensemble de bits soit pair. Pour vérifier le solde, nous devons procéder comme suit :

A - On vérifie le solde des bits dont le reste est divisé par 2.

B- On vérifie le solde des bits dont le reste est divisé par 4, deux ou trois.

C- On vérifie le solde des bits dont le reste est divisé par 8, quatre, cinq, six ou sept.

D- Nous vérifions le solde des bits dont le reste est divisé par 16, huit, neuf, dix, onze, douze, treize, quatorze ou quinze. Etc.

* Décodage:

1- Calculer le solde des bits:

A- On calcule le solde des bits dont le reste est divisé par 2.

B- On calcule le solde des bits dont le reste est divisé par 4, deux ou trois.

C- On calcule le solde des bits dont le reste est divisé par 8, quatre, cinq, six ou sept.

D- On calcule le solde des bits dont le reste est divisé par 16, huit, neuf, dix, onze, douze, treize, quatorze ou quinze. Etc.

2- Ensuite, nous écrivons les équilibres obtenus sous forme de nombre binaire, respectivement. La valeur du solde obtenu dans la première étape est de 20, la deuxième étape est de 21, la troisième étape est de 22 et ainsi de suite.

3- Si le nombre obtenu à l'étape précédente est zéro, cela signifie qu'il n'y a pas d'erreur, mais s'il n'est pas zéro, le nombre résultant indique le numéro de bit qui a une erreur en binaire. Le bit souhaité doit être inversé.

4- Dans la dernière étape, nous supprimons les bits de contrôle, et si dans la deuxième étape de codage est ajouté un bit, nous le supprimons également. Enfin, les données originales sont obtenues.

Procédure d’exécution

Dans notre code, pour l'envoi, nous prenons le message à envoyer comme entrée. Nous calculons tout d'abord le nombre de bits de redondance, ce qui nous permet de déduire la longueur du code de Hamming. Par la suite nous marquons les emplacements des bits de redondance dans le code à générer (initialisation avec -1). Nous insérons les bits du message dans les emplacements désignés pour le message, puis nous calculons les parités des bits pour les écrire dans les bits de redondance (pair = 0, impair = 1). Nous remplissons finalement le buffer avec le code calculé et nous le retournons pour être envoyé.

Pour le décodage, nous prenons le code reçu comme entrée de la fonction. A partir du code reçu nous faisons une extraction du message et des bits de redondance. Pour détecter la présence d'une erreur nous faisons une boucle qui parcourt tout le code reçu. Si une erreur est détectée, nous marquons sa position pour la corriger par la suite. Nous pouvons corriger une seule erreur, cependant si plusieurs erreurs ont lieu le message corrigé sera erroné. Dans le cas de présence d'une erreur, nous retournons un booléen False avec le code corrigé. S'il n y a pas d'erreur, un booléen True sera retourné avec le code reçu.