UNIVERSITÉ DE SHERBROOKE
DÉPARTEMENT D'INFORMATIQUE

IFT 585 – Télématique

Travail pratique #1

Travail présenté à
M. Bessam ABDULRAZAK

par

Raphaël Cayouette (15 064 430)
Jean-Philippe Croteau (15 072 957)
Jonathan Martineau (15 059 124)
Félix Vigneault (15 079 576)

Le 9 juin 2017

Table des matières

Contenu de la remise

Notre ZIP de remise contient la solution Visual Studio de notre travail. Dans cette solution, deux projets sont présents: Le projet de code et le projet des tests unitaires. Le projet *TransmissionSimulation* contient tous les fichiers de code principaux du travail. Dans le dossier *Components* se trouvent les classes de composants, soit *Transmitter* et son interface publique *ITransmitter*, ainsi que *Station*, la classe des deux stations qui communiquent via le transmetteur. Le dossier *Helpers* contient la classe *HammingHelper*, qui se charge de l'encryption, la décryption et les autres fonctions reliées au code de Hamming. Le dossier *Models*, quant à lui, contient des fichiers de configuration pour la station: *Frame* et *StationParameters*. Enfin, le dossier *Ressources* contient un fichier de constantes utilisé par la plupart des classes de notre programme et l'icône de notre application pour notre équipe Kappa. À la racine de notre projet se trouvent le fichier *Program* qui lance le programme, *MainForm* qui est l'interface graphique du logiciel, deux fichiers que nous lisons pour transférer des données d'une station à l'autre ainsi que le fichier *App.config* qui est le fichier de configuration utilisé pour déterminer les paramètres des stations (taille des fenêtres, délai de temporisation, etc.). L'autre projet, *TransmissionSimulationTests* contient, comme son nom l'indique, les tests unitaires que nous avons créés pour valider nos implémentations.

Procédure d'exécution

Le programme charge les configurations du programme à partir du fichier « TransmissionSimulation.exe.config ». Une fois que tous les paramètres sont vérifiés, le programme est démarré en lançant le transmetteur et les deux stations. Chaque station reçoit ses paramètres respectifs, dont les flux vers les fichiers d'entrées et sorties.

L'utilisateur du programme peut décider d'insérer des erreurs dans la prochaine trame transférée avec un des deux boutons d'insertion d'erreurs.

La station, une fois démarrée, prend les données fournies dans le fichier d'entrée pour l'envoyer au transmetteur lorsqu'il est prêt. L'envoi et la réception suivent le protocole 6 de Tanenbaum, et la trame est encodée avec le code de Hamming avant d'être envoyée sur le fil.

Lorsque la station envoie une trame au transmetteur, ce dernier vérifie quelques conditions supplémentaires, applique les erreurs, si nécessaire, sur la trame, attend un délai arbitraire représentant une latence réseau, et transmet la trame vers la réception de l'autre station. Celle-ci vérifie alors auprès du transmetteur si des données ont été reçues, puis récupère les données.

Ensuite, la trame est décodée avec le code de Hamming et, selon le mode de correction/détection actuel, la trame peut être corrigée ou bien abandonnée s'il y a une erreur. Les données reçues sont envoyées dans le fichier de sortie dans l'ordre.

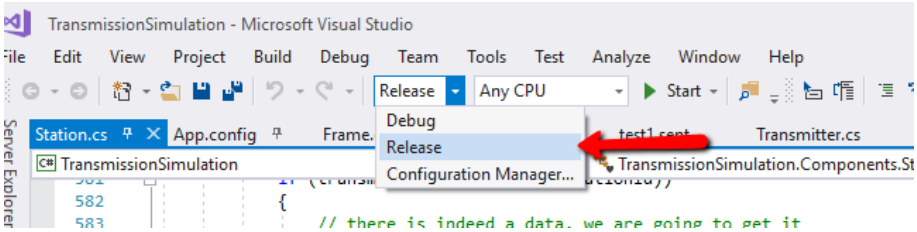
À chaque fois qu'une trame est confirmée comme étant reçue, la barre de progression avance proportionnellement à la grosse de la trame par rapport à la taille totale du fichier. Une fois que la barre est pleine, une boîte dialogue va s'afficher pour informer l'utilisateur de la fin du transfert.

Guide de compilation et exécution

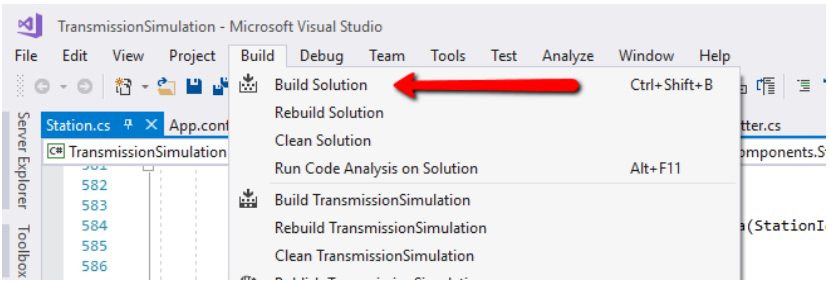
Compilation

La compilation du programme se fait à l'aide de Visual Studio 2017. Il est important d'utiliser la dernière version de l'éditeur puisque le code est en C# 7.0, qui est seulement disponible dans Visual Studio 2017. Visual Studio doit avoir accès aux librairies .Net Windows Form, mais l'ouverture du projet devrait automatiquement proposer d'installer les librairies manquantes, advenant le cas où elles ne seraient pas présentes.

Pour compiler, il suffit d'ouvrir le fichier *TransmissionSimulation*\ *TransmissionSimulation.sln* avec Visual Studio 2017. Ensuite, dans VS, il faut changer le profil de compilation pour mettre Release au lieu de Debug.



Enfin, dans l'onglet *Build*, il faut cliquer sur Build solution.



Voilà! Le programme compilé sera placé dans le projet à l'emplacement relatif suivant : *TransmissionSimulation\TransmissionSimulation\bin\Release*

Exécution du programme compilé

Pour exécuter le programme, il faut être sur une machine Windows et utiliser l'exécutable *TransmissionSimulation.exe* pour partir le programme. Les paramètres par défaut sont mis dans le fichier *TransmissionSimulation.exe.config* et peuvent être édités directement dans ce fichier avant l'exécution du programme.

Par défaut, le programme utilise les fichiers test1.sent et test2.sent, mais il est possible de choisir les fichiers à envoyer en écrivant leur chemin d'accès complet dans le fichier de configuration (ex: C:\Users\JohnDoe\example.txt). Un fichier vide signifie que la station n'enverra pas de données (attention à l'encodage du fichier, car certains fichiers sont parfois vides, mais contiennent quand même certains caractères, par exemple UTF-8 avec signature). Le fichier test2.sent est un exemple de fichier vide valide.

Pour faire un test avec les configurations par défaut, il faut seulement ouvrir l'exécutable. Le transfert sera affiché à l'écran, puis une fois le transfert complété, il faut fermer le programme, puis une fois le programme fermé, le fichier test2.received sera mis à jour avec le contenu reçu. À ce moment-là, le contenu de test1.sent doit être égal au contenu de test2.received.

Guide d'usager et Analyse de conception

Interface utilisateur

L'interface contient quatre zones principales de contenus : l'affichage des données envoyées par les deux stations, l'affichage des données dans les deux stations, l'insertion d'erreurs et une barre de progression du transfert.

Les deux zones d'affichages présentent l'identifiant, le type de trame, le numéro de « Ack », ainsi que la longueur en octets des données de la trame. La zone d'affichage de réception contient également une colonne pour indiquer le plus haut niveau de problème rencontré (correction ou détection d'erreur avec le code de Hamming).

Pour l'insertion d'erreurs dans les trames, deux options sont disponibles : inversement de bit à des positions données et insertion selon le niveau de problème. Quand le bouton « Injecter Erreur(s) » est appuyé, toutes les positions données qui sont différentes de -1 vont engendrer une inversion de bit dans la prochaine trame envoyée. L'insertion aléatoire permet d'injecter un nombre quelconque d'erreurs corrigibles (1 bit), détectables (2 bits), et indéterminées (3 bits). Chaque erreur demandée est insérée dans une zone de bits différente (grosseur déterminée avec Hamming) et le choix des bits à inverser dans chaque zone est aléatoire.

Finalement, la barre de progression est calculée selon la taille totale du fichier et le nombre de trames reçues correctement.

La conception de l'interface utilisateur a beaucoup évolué lors du développement. Initialement le programme devait être en console, mais pour améliorer la qualité de l'affichage nous avons transféré vers le Windows Form. La version finale fonctionne bien à l'exception du défilement des zones d'affichages qui n'est pas totalement automatique.

Stations

La classe Station peut recevoir sa configuration (taille de la fenêtre, timeout, etc.), son lien vers le Transmitter (qui représente la couche physique) ainsi que la référence vers les fichiers d'entrées et de sorties (qui représentent la couche Réseau). Elle offre uniquement une méthode Start qui part la station.

Le Protocole 6 a été implémenté suivant l'ébauche donnée par Tanenbaum. La gestion des événements est faite par attente active, donc la station boucle jusqu'à ce qu'un événement survienne. De cette façon, on peut choisir l'ordre des événements lorsqu'il y en a plusieurs simultanément. L'envoi des Nak, des trames visées par un Nak et des trames en timeout sont considérées plus prioritaires, suivi par l'envoi d'une trame de donnée, l'envoi d'un Ack dont le délai est expiré et finalement la lecture des trames qui arrivent.

L'implémentation actuelle est assez stable et remplit les exigences du Protocole 6 :

- Envoi de plusieurs trames avant réception d'un acquittement (via les fenêtres et les tampons) ;
- Les données sont envoyées à la couche supérieure (représentée par les fichiers) dans l'ordre ;
- Le nombre de trames autorisées pour les tampons est configurable pour chaque station ;
- La communication est bidirectionnelle, donc les deux stations peuvent envoyer des données ;
- Les trames dont le délai de temporisation expire sont réenvoyées ;
- Le « piggybacking » est utilisé pour envoyer des Ack via une trame de données s'il y en a une avant la fin du délai imparti.

La procédure de Checksum n'a pas été implémentée, mais il y a quand même une vérification faite pour s'assurer que l'entête de la trame est au moins dans les valeurs permises. Étant donné que cette validation n'est pas suffisante, l'implémentation courante du protocole peut mener à des états non sûrs lors de la réception d'une trame corrompue qui n'a pas été détectée par le code de Hamming (3 erreurs ou plus dans un seul octet). Par exemple, le id dans le champ Ack de la trame pourrait avoir été changé sans qu'on s'en rende compte, ce qui a pour effet de désynchroniser l'échange.

Ce problème crée de l'instabilité dans l'implémentation, donc l'envoi de multiples erreurs indéterminées (3 bits et plus) peut parfois causer des erreurs, mais c'est un cas qui arrive rarement. La plupart du temps, le programme réussit à récupérer, mais le fichier de destination peut contenir des différences avec le fichier source. Ces problèmes ne peuvent pas être réglés dans l'implémentation actuelle.

Transmetteur

Le transmetteur fonctionne sur la base de deux stations, qui représentent deux postes qui seraient connectés via le réseau. Le transmetteur fournit quelques méthodes aux autres classes via son interface. Deux des méthodes permettent de vérifier l'état courant du transmetteur, soit: *TransmitterReady*, qui indique à la station si elle a le OK pour envoyer des nouvelles données, et *DataReceived*, qui permet à la station de savoir si de nouvelles données prêtes à être lues sont arrivées. Par la suite, deux autres méthodes sont disponibles afin d'interagir avec le transmetteur. La première, *SendData*, permet, après avoir obtenu l'approbation d'envoi par la méthode *TransmitterReady*, d'effectuer un transfert de données vers l'autre station. La seconde méthode, *GetData*, permet, après avoir vérifié auprès de *DataReceived*, d'obtenir les données qui ont été reçues.

Une méthode *InsertRandomErrors* permet à l'interface usager d'insérer des erreurs aléatoires dans la prochaine trame envoyée entre les deux stations et une propriété *IndicesInversions* permet d'insérer des erreurs à des endroits précis dans les bits de la prochaine trame.

Enfin, lorsque le transmetteur détecte des nouvelles trames à transférer vers une station, la méthode *TransferFrame* s'active dans un fil d'exécution parallèle (thread). C'est dans celle-ci que l'injection d'erreur se produit sur la trame à envoyer, après un délai qui simule une latence réelle. La conception du transmetteur a débuté avec le simple envoi et réception de trames, avant d'ajouter la possibilité d'injecter plusieurs types d'erreurs.

Quelques embûches se sont présentées durant la conception, mais elles ont rapidement été détectées et corrigées grâce à la mise en place de tests unitaires sur les différentes fonctionnalités du transmetteur. Tout est fonctionnel dans le transmetteur.

Code de Hamming

L'implémentation de notre code de Hamming est un Hamming(12, 8) avec un bit de parité supplémentaire. C'est-à-dire un Hamming(13, 8) qui donne 8 bits de données, 4 bits de parité et 1 bit de parité global. La lecture du livre de Tanenbaum nous a permis de choisir ce type de conception et l'implémentation de nombreux tests unitaires ont permis d'avancer rapidement dans l'élaboration du code de Hamming tout en sachant qu'il fonctionne à la perfection.

Le *HammingHelper* fournit principalement deux interfaces publiques : *EncryptManager* et *DecryptManager*. Celles-ci appellent le *HammingManager* en prenant soin d'enlever ou de mettre du « padding » sur la totalité des données à modifier. Le *HammingManager,* quant à lui, divise la totalité des bits reçus en plusieurs petits groupes de bits (8 bits pour le mode d'encodage, 13 bits pour le mode de décodage) et appelle la bonne méthode selon le mode (*Encrypt* ou *Decrypt*) pour chaque groupe.

Encrypt fait tout simplement ajouter des bits de parité à toutes les puissances de 2. Il ajoute également un bit de parité global au tout début des bits de données pour la détection des erreurs doubles (SECDED). *Encrypt* retourne les données encryptées ainsi que le statut OK.

Decrypt, quant à lui, calcule le syndrome d'erreur avec les bits de parité. S'il y a une erreur de syndrome et que le bit de parité global n'est pas bon, il y a une erreur d'un bit et on peut corriger le bit en erreur en inversant le bit à la position de l'erreur de syndrome dans le tableau de bits. Sinon, s'il y a une erreur de syndrome et que le bit de parité global est bon, il y a une erreur de 2 bits et cela veut dire que l'on ne peut pas corriger les données. Dans le pire des cas, il y a une erreur de 3 bits ou plus et l'implémentation de notre code de Hamming ne peut rien y faire. *Decrypt* retourne les données décryptées ainsi que le statut OK, CORRECTED ou DETECTED selon le mode et le cas.