Devoir 2 - Rapport

présenté à

M. Loïc Tsobdjou Dongmo

M. Abdelhakim Hafid

IFT3325

Steve Lévesque
Bojan Odobasic

2 mai 2022

Table des matières

1 Introduction

Le but de ce TP était d'implémenter une version simplifiée du protocole HDLC en Java. Il s'agissait de coder la communication entre deux entités sur un réseau, soit un émetteur de données et un récepteur. Les données à envoyer se résumaient à du texte contenu dans un fichier type « txt » ordinaire. Les pertes ou erreurs dans les trames et dans les accusés de réception devaient pouvoir être récupérées. Enfin, il fallait créer des procédures de tester afin de valider le bon fonctionnement de l'application. Le présent rapport a pour but de guider dans la compréhension des fonctionnalités et de la structure de notre application. Toutes les fonctionnalités demandées ont été implémentées.

Tel que demandé, l'application a été conçue en Java en visant la version disponible au DIRO (OpenJDK 1.8 sur Linux). Les applications d'envoi et de réception devraient fonctionner sur toutes les versions supérieures de Java ainsi que toutes les plateformes. Cependant, les procédures de tests automatisées ne fonctionnent pas sur Windows. Il est toutefois possible de lancer les tests sur Windows en procédant un par un quand même. Ceci sera expliqué plus tard.

2 Architecture générale

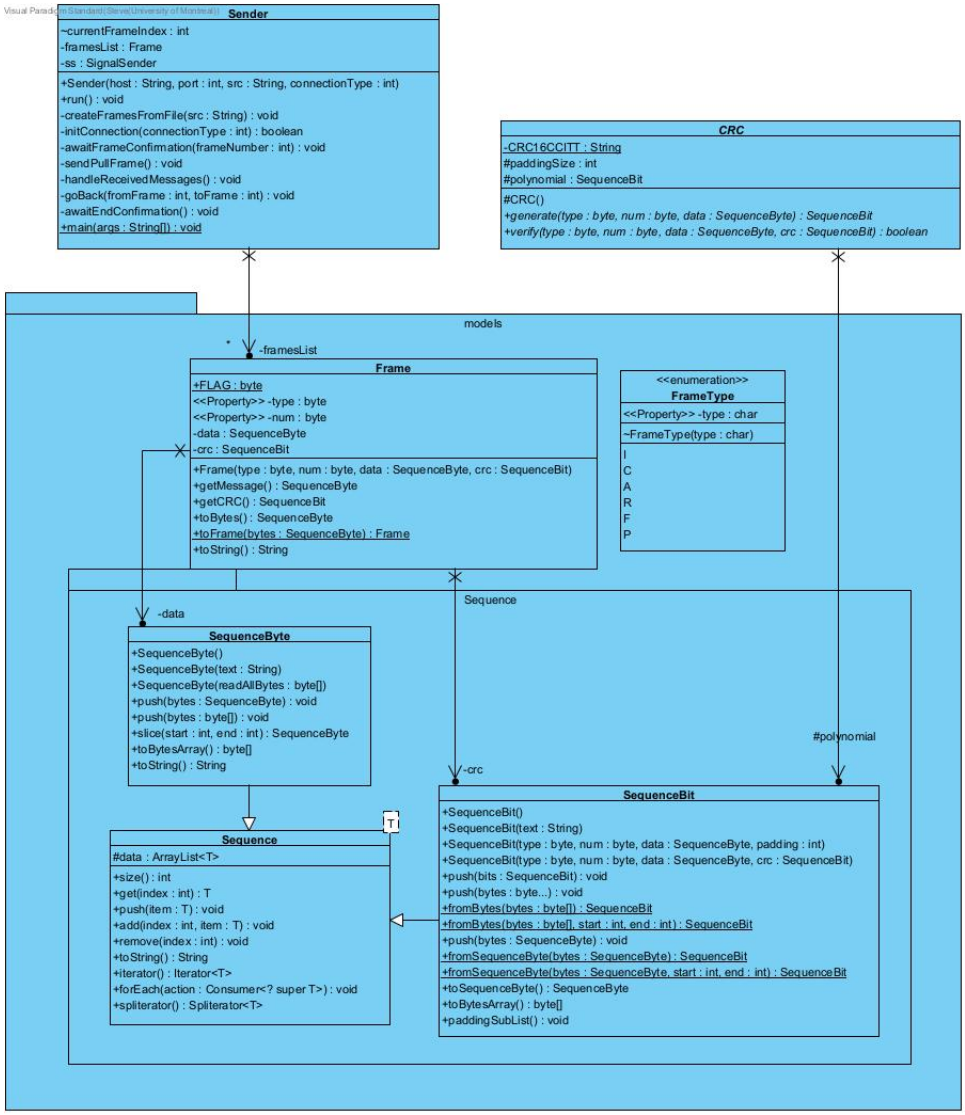
Notre application suit fortement le paradigme de la programmation orientée objet. Pour aider à la réutilisation du code, la plupart des classes héritent d'une classe abstraite. De plus, plusieurs classes se conforment à au moins une interface ce qui aide le développement.

Nous avons pris la décision d'abstraire les données plus rudimentaires (comme les bits et les bytes) en les intégrant dans une classe de séquence, plutôt que d'utiliser des tableaux. Ceci limite le nombre de classes nécessaires et assure un traitement plus uniforme de ces types de données.

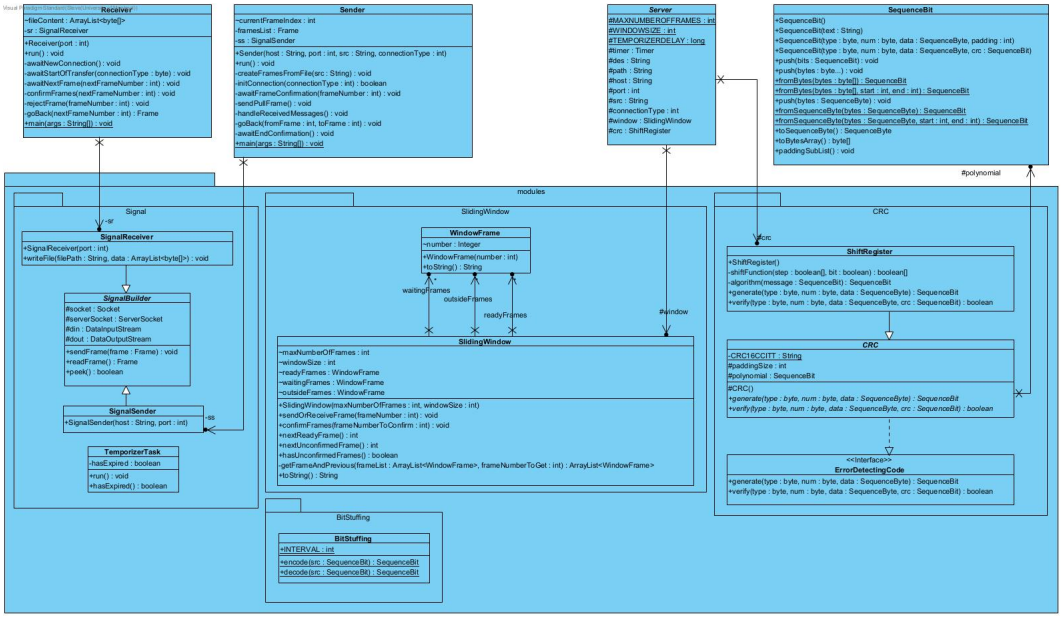
2.1 Diagrammes de classes

Nous présentons ci-dessous les diagrammes de classes. Les fichiers d'origine, en format Visual Paradigm, sont joints au présent rapport. Les diagrammes sont séparés par paquet (« package »), représentés par un large dossier bleu. Les sous-paquets s'y retrouvent directement. Les classes extérieures au paquet, mais dont il y a des dépendances dans le paquet, sont visibles à l'extérieurs des dossiers. La section suivante décrira plus en détails les différentes classes.

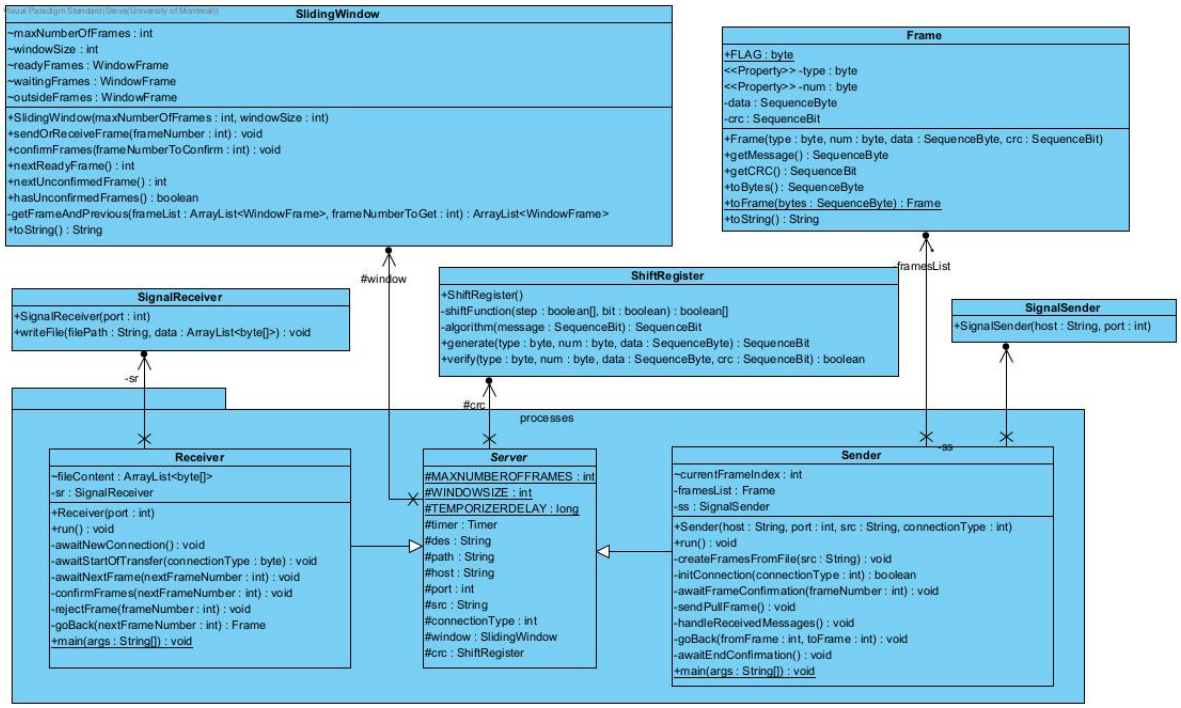
2.1.1 Paquet models



2.1.2 Paquet module



2.1.3 Paquet processes



2.2 Description des classes

Cette section détaillera les classes importantes de l'application. Les classes utilisées pour les tests seront présentées dans la section suivante.

2.2.1 Classes d'application

- `Server.java (classe)`
- `Sender.java (classe)`
- `Receiver.java (classe)`

Ce sont les points d'entrée pour l'application (contiennent la fonction main). La classe Sender s'occupe de lire le fichier texte, de créer les trames, et de faire l'envoi. La classe Receiver, quant à elle, attend une connection, analyse les traimes reçues et enregistre le tout dans un fichier texte. La classe Server est une classe abstraite qui contient, entre autres, des constantes (par exemple, largeur de la fenêtre coulissante et délai du temporisateur).

2.2.2 Classes de données

- `Sequence.java (classe abstraite)`
- `SequenceBit.java (classe)`
- `SequenceByte.java (classe)`
- `Frame.java (classe)`
- `FrameType.java (énumération)`

La classe Sequence est abstraite et représente une séquence ordonnée d'un type de données. SequenceBit est une réalisation de Sequence dont le type est un bit (concrètement un booléen). SequenceByte, quant à elle, est une réalisation de type byte. La classe Frame permet de stocker une trame de communication. À chaque trame est associé un

type qui doit être contenu dans FrameType.

2.2.3 Classes de modules

Les classes de modules permettent de faire des opérations en lien avec la communication. On y retrouve le calcul d'erreur (CRC), le bourrage de bits (bit stuffing), le traitement du signal et la fenêtre coulissante.

Module BitStuffing

- `BitStuffing.java (classe)`

Le fanion proposé est « 01111110 ». Afin d'éviter de reconnaître indûment cette séquence en plein milieu d'un trame, nous ajoutons un zéro après détection d'une suite de cinq un. La classe BitStuffing permet d'encoder et de décoder une séquence de bits en respectant cette règle.

Module CRC

- `ErrorDetectingCode.java (interface)`
- `CRC.java (classe abstraite)`
- `ShiftRegister.java`

Ces classes permettent de faire la détection d'erreur par un calcul de « cyclic redundancy check » (CRC). L'interface ErrorDectectingCode précise les deux opérations nécessaires (générer un code et vérifier des données). La classe CRC implémente cette interface. Elle contient le polynôme utilisé (CRC16-CCITT). La classe ShiftRegister est une réalisation de CRC qui applique la technique du « shift-register » tel que dans le premier devoir.

Module Signal

- `SignalBuilder.java (classe abstraite)`
- `SignalSender.java (classe)`
- `SignalReceiver.java (classe)`
- `TemporizerTask.java (classe)`

Ce module est une abstraction des fonctionnalités fournies par les « sockets » de Java. Elles font également le lien avec les « streams » de données qui permettent la lecture et l'écriture des « sockets ». Ceci facilite grandement le traitement du signal. La classe SignalBuilder est une classe abstraite qui implémente la lecture et l'écritue de trames par les « streams ». La classe SignalSender est une réalisation du SignalBuilder et initialise un « socket ». La classe SignalReceiver est aussi une réalisation du SignalBuilder. Elle initialise un « ServerSocket » ainsi qu'un « Socket » lors d'une demande de connection. Également, elle permet l'écriture dans un fichier. Enfin, la classe TemporizerTask hérite du TimerTask de Java et ne permet que de vérifier l'expiration du temporisateur qui est déclenché par un Timer standard.

Module SlidingWindow

- `SlidingWindow.java (classe)`

Ce module permet d'avoir une fenêtre coulissante dynamique. Elle garde en mémoire les numéros de trame envoyées, confirmées ainsi que celles qui ne sont pas actuellement dans la fenêtre. Les numéros de trames sont stockés dans une sous-classe privée nommée WindowFrame.

3 Tests

3.1 Classes de tests

Classes d'application

- `Channel.java (classe)`
- `Tests.java (classe)`

Ces classes permettent de lancer les tests. La classe Channel est un intermédiaire entre les classes Sender et Receiver. Elle contient une sous-classe nommée DirtyChannel qui analyse les trames qui circulent et selon le test choisi, détruit une trame spécifique ou l'altère. Il y a deux DirtyChannel qui sont initialisés, soit un du côté de l'émetteur et un du côté du récepteur. Le lancement du Channel permet d'effectuer un seul test à la fois. La classe Tests, quant à elle, permet d'effectuer des tests à la chaîne. Pour chaque test, elle initialise dans un fil (« Thread ») un Receiver, un Channel et un Sender. Lorsque les fils ont terminé leur exécution, elle compare les caractères des fichiers textes pour décider de la réussite du test.

Classes de module (signal)

- `SignalTester.java (classe)`

Cette classe contient la définition des tests. Elle reçoit une trame et si celle-ci répond aux conditions, elle est éliminés ou altérée. Chaque test ne peut être exécuté qu'une seule fois par communication.

3.2 Description des tests

Nous avons identifié dix tests qui nous apparaissaient pertinents et qui permettent de couvrir les cas extrêmes. Notre fichier de tests contient dix lignes. Puisque seulement trois des huits bits de numérotation sont utilisés, nous pouvons numéroter les trames de 0 à 7. La fenêtre contient, quant à elle, sept trames. Les dix lignes du fichier permettent donc de remplir la fenêtre au complet une fois. Les tests effecutées sont les suivants :

1. S : Perte de la demande de connexion de l'émetteur
2. S : Perte de la première trame de donnée envoyée
3. S : Perte de la dernière trame de la première fenêtre
4. S : Perte de la première trame de la deuxième fenêtre
5. S : Perte de la dernière trame de donnée envoyée
6. S : Corruption de la quatrième trame de donnée
7. S : Perte de la trame de fin de connexion de l'émetteur
8. R : Perte de la confirmation de connexion du récepteur
9. R : Perte d'une trame de confirmation de fenêtre du récepteur
10. B : Perte de la quatrième trame de donnée envoyée suivi d'une perte de la trame de rejet du récepteur

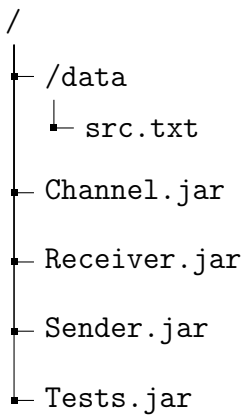
La lettre à côté du numéro du test correspond au canal qui est testé ('S' pour émetteur, 'R' pour receveur et 'B' pour les deux). La majorité des tests sont effectués pour les pertes de trame. Nous n'effectuons qu'un seul test de corruption. Il ne nous apparaît pas nécessaire de tester davantage de cas pour la corruption car les séquences d'événements qui en résultent sont les mêmes.

4 Déploiement

Le code source est distribué avec le présent rapport. Pour faciliter la tâche, nous avons déjà compilé le projet et nous avons inclus également des fichiers Jar qui permettent de lancer l'émetteur (Sender.jar), le récepteur (Receiver.jar), l'intermédiaire (Channel.jar) et les tests automatisés (Tests.jar). Tel que mentionné en introduction, il est recommandé d'utiliser OpenJDK 1.8 sur une machine Linux telles qu'on retrouve au DIRO. Néanmoins, l'application devrait fonctionner sur les versions supérieures de Java et sur toutes les plate-formes, hormis les tests automatisés qui ne fonctionnent pas sur Windows.

4.1 Lancement de l'application

Voici la structure de répertoire attendue pour que l'application fonctionne :



À la fin du traitement, on devrait retrouver normalement un fichier `des.txt` dans le dossier `/data`.

Transfert simple sans erreur

Pour ce faire, on lance dans l'ordre, Receiver puis Sender. Les arguments nécessaires sont les mêmes que dans l'énoncé du devoir. Par exemple :

- `java -jar ./Receiver.jar 3325`
- `java -jar ./Sender.jar localhost 3325 src.txt 0`

Transfert avec erreur (test unique)

On doit lancer dans l'ordre, le Receiver, le Channel, puis le Sender. Cette fois-ci, le port du Receiver et celui du Sender seront différents. C'est le Channel qui fait le relai. Les arguments du Channel sont les suivants :

`<sender port> <receiver host> <receiver port> <test> <channel>`

Le numéro du test (`test`) et le canal à tester (`channel`) sont disponibles à la section 3.2.

Par exemple, on peut lancer :

- `java -jar ./Receiver.jar 3324`
- `java -jar ./Channel.jar 3325 localhost 3324 1 S`
- `java -jar ./Sender.jar localhost 3325 src.txt 0`

Tests en lots

On lance simplement le fichier Tests. Les arguments sont :

`<sender port> <receiver host> <receiver port>`

Par exemple, on peut lancer :

- `java -jar ./Tests.jar 3325 localhost 3324`

La conclusion des tests s'affichera à l'écran après moment.