Sébastien Richer (0837433)
Gevrai Jodoin-Tremblay (0989650)

IFT3325
Devoir 2: Rapport
Paquets, classes, interfaces, méthodes, paramètres, utilités et outputs

Compilation et tests du projet

Nous avons inclus un *makefile* à la racine du dossier tp2 s'occupant de toutes les tâches de compilations et de tests JUnit. Un simple *make* dans la ligne de commande effectue toutes ces opérations automatiquement, indiquant des erreurs si les tests ne passent pas. Par la suite, pour exécuter les programmes *Receiver* et *Sender* sur le serveur du DIRO, on a deux choix possibles :

1. Simplement exécuter les scripts
    - ./Receiver <Numéro de port> [<Ratio d'erreur>] [<Nom fichier de sortie>]
    - ./Sender <Nom de machine> <Numéro de port> <Nom du fichier à envoyer> 0
2. Pour la syntaxe spécifié dans l'énoncé, ajout du dossier courant au PATH :
    - Dans la ligne de commande exécuter : export PATH=$PATH:$(pwd)
    - Receiver <Numéro de port> [<Ratio d'erreur>] [<Nom fichier de sortie>]
    - Sender <Nom de machine> <Numéro de port> <Nom du fichier à envoyer> 0

Il est à noter que les arguments entre crochets carrés de *Receiver* sont optionnels et permettent un contrôle plus avancé sur celui-ci :

- [<Ratio d'erreur>] : Entre 0.0 et moins que 1.0, permet de simuler des erreurs de communication.
- [<Nom de fichier de sortie>] : Permet de spécifier le nom du fichier de sortie de la communication reçue.

Il est bien évidemment nécessaire d'exécuter Receiver en premier et d'utiliser le nom de machine 'localhost' avec le même numéro de port pour que la connexion s'effectue.

Finalement, nous avons un simple programme ./SendAndReceiveTest qui automatise l'envoi du fichier *testInput.txt* et reçoit les données dans le fichier *output.txt*, affichant un détail de toutes les frames envoyées et reçues dans le terminal.

1. Paquet: frames

1.1 Classe: Frame

Cette classe abstraite représente (bien sûr) toute frame envoyée de l'émetteur au récepteur ou vice-versa. Elle contient des variables contenant les types de champs envoyés dans une frame (deux de type byte, une de type byte[]), ainsi que des constantes de type int qui indiquent la taille de ces champs.

1.2 Classes: AckFrame, FinalFrame, InformationFrame, PollFrame, RejFrame

Ces classes étendent toutes la même classe parent, Frame. Elles ont chacune une constante de type byte déterminant le type de la frame. Nous avons décidé de séparer chacune des différentes frames, histoire de rendre la création et la manipulation de celles-ci plus facile.

1.3 Classe: ConnectionFrame

Une instance de ConnectionFrame est la première frame à être envoyée de l'émetteur au récepteur lors d'un échange. Elle ades constantes de type *byte* pour le genre de session (go back n, selective reject, stop and wait) qu'on tente d'ouvrir au moment de la connection.

Nous avons, dans ce projet implémenter les versions *Stop-and-Wait* et *Go-Back-N,* et ainsi une tentative de connection de type *SelectiveReject* retournera une exception.

1.4 Classe: MalformedFrameException

Cette classe donne simplement la possibilité d'instancier une exception plus précise, qui signale un problème avec la frame reçue.

1.5 Classe: FrameFactory

Cette classe comporte une seule méthode qui prend un tableau de bytes en paramètre et instancie une frame d'un certain type selon ce tableau. Cela, évidemment, si le tableau en question représente une frame valide; sinon, FrameFactory jette une MalformedFrameException.

2. Paquet: network

2.1 Classe: CRCCalculator

Cette classe, comme son nom l'indique, calcule le Cyclical Redundancy Code (CRC). Elle a une variable, generator, de type byte[], qui est passée en paramètre au constructeur lors de l'instanciation de la classe. Le générateur de CRC n'est valide que si byte[] generator commence par '1'.

CRCCalculator comporte une méthode, shiftLeft, qui prend un tableau de bytes en paramètre et déplace tous ses bits vers la gauche, bit par bit, par une série d'opérations bitwise.
Le calcul du CRC est effectué selon l'algorithme du shift-register que l'on a vu dans le devoir 1, mais implémenté sur des bytes plutôt que des bits. Ainsi, après la boucle principale, le reste de la division se trouve dans le registre et est retourné.

2.3 Classe: ReceiveFrameBackgroundTask

Cette classe étend la classe Thread et a deux variables initialisées dans son constructeur, l'une de type NetworkAbstraction et l'autre de type IFrameReceiver.

La méthode run de cette classe, qui supplante celle de Thread, attend incessamment la réception d'une frame et notifie un IFrameReceiver.

2.4 Interface: IFrameReceiver

Cette interface doit être implémenter par toute classe voulant communiquer avec un ReceiceFrameBackgroundTask, en lui permettant d'être notifié lors de la réception d'une nouvelle Frame, et ce de manière asynchrone. La première, notifyFrameReceived, permet l'envoie de la Frame en question, et la deuxième, notifyIOException, permet de notifier une IOException dans le NetworkAbstraction de réception.

2.5 Classe: NetworkAbstraction

Cette classe représente, comme son nom l'indique, le réseau par lequel communiquent l'émetteur et le récepteur de données; le réseau en question est ici simulé au niveau software plutôt que hardware.

NetworkAbstraction comporte plusieurs constantes: la constante flag, de type byte, contient les flags de début et de fin de frame; la constante GX16, de type byte[], contient le polynôme générateur utilisé pour calculer le CRC; quant à la constante crcCalculator de type CRCCalculator, elle prend GX16 en paramètre pour, bien sûr, calculer le CRC.

La classe NetworkAbstraction a aussi deux variables: une variable socket de type Socket, ainsi qu'une variable errorRatio de type double, qui représente la proportion d'erreurs simulés suite à la transmission de données entre l'émetteur et le récepteur.

Le socket susmentionné peut être fermé par la méthode close. À part cette méthode, la classe présente contient les méthodes sendFrame et receiveFrame, dont l'utilité est évidemment d'envoyer et de recevoir des frames.

La première est de type void et prend en paramètre une frame et fait tout le nécessaire pour l'envoie de cette frame sur le réseau. Elle y ajoute le CRC, écrit le flag de départ, fait le bit stuffing pour s'assurer qu'on puisse distinguer les données des flags, écrit le flag de fin dans le output stream et, finalement, envoie les données (les écrit dans le output stream).

La deuxième est de type Frame et, une fois appelée, “bloque” (cesse toute activité) jusqu’à recevoir une frame valide. Quand une suite de bytes potentiellement correcte est reçue, le CRC est vérifié, et si cette vérification est concluante et qu’il n’y a pas eu de problème de transmission, une frame valide est retournée. Cette méthode garantit que la Frame reçue est valide.

2.6 Classe: ErrorGenerator

L’utilité de cette classe et de sa seule méthode apply sur une liste de byte, est de simuler des bytes erronés dans la communication émetteur-récepteur, donc des bytes erronés dans le tableau passé en paramètre. Puisque Java garantit déjà une communication sans erreurs, il nous fallait une façon détournée de simuler les erreurs de communication.

Le fait d’appliquer des erreurs aléatoirement sur les frames envoyées et reçues nous permet aussi de s’assurer, après plusieurs tests, de la validité de nos algorithmes sur toute erreurs de communication.

3. Paquet: receiver

3.1 Classe: Receiver

Cette classe contient le main() du côté serveur, et ne fait qu’instancier un ReceiverWorker à la connection d’un client.

Cette classe a deux variables: serverSocket de type ServerSocket et errorRatio de type double, aux noms assez explicites. Tout comme la variable errorRatio de la classe NetworkAbstraction, errorRatio contient ici la proportion d’erreurs suite à la transmission de données entre l’émetteur et le récepteur.

La méthode acceptConnections, de type void, ne cesse de tourner une fois appelée et crée un socket client utilisant le même port que le socket serveur. De plus, elle crée et démarre un thread de type ReceiverWorker, qui est la classe que nous allons décrire à l’instant.

3.2 Classe: ReceiverWorker

ReceiverWorker implémente l’interface IFrameReceiver. Cela signifie que cette classe peut être notifié de la réception de frames et des exceptions IO. Cette classe comporte d’ailleurs plusieurs variables utiles pour cela, incluant une LinkedList (receptionQueue), de type Queue<Frame>, pour stocker les frames reçues.

La méthode run de cette classe attend incessamment d’être notifiée de la réception d’une frame ou d’une exception IO.

La méthode disconnect, de type void, initialise puis envoie une frame ACK, puis ferme l’instance de NetworkAbstraction.

La méthode processPoll, de type void, envoie le num courant que l’on attend.

La méthode processConnectionFrame, de type void, renvoi simplement un ACK si le type de connection est supporté (Go-Back-N ou Stop-and-Wait) et affirme que la connection est établie.

La méthode processInformationFrame, de type void, prend une frame d'information en paramètre et, si la connection est établie et que la frame reçue est effectivement celle attendue, écrit ses données dans le output stream et envoie une frame ACK pour confirmer la réception d'une frame d'information. Sinon, elle envoi REJ avec le numéro de frame auquel on s'attendait.

La méthode notifyFrameReceived, de type void, prend une frame en paramètre et l'ajoute à la LinkedList receptionQueue.

Quant à la méthode notifyIOException, de type void, elle prend une exception IO en paramètre et signale au thread principal qu'il y a eu une exception IO.

4. Paquet: sender

4.1 Classe: Sender

Cette classe ne comporte qu'une méthode main qui sert à envoyer un fichier après avoir pris en argument le nom de la machine, le numéro de port, le nom du fichier à envoyer et le type de session (0 pour go back n, 1 pour selective reject, 2 pour stop and wait). L'appeler sans arguments affiche une aide.

5. Paquet: sessions

5.1 Classe: Session

Cette classe abstraite implémente l'interface IFrameReceiver et représente toute session de connection concrète d'un Sender, qu'elle soit de type GoBackN, SelectiveReject ou StopAndWait. Elle comporte deux constantes qui déterminent le nombre maximum de tentatives de connection et le temps d'attente entre chaque tentative.

Étant abstraite, cette classe a une méthode abstraite send, de type boolean, qui prend un InputStream en paramètre, et qui doit être implémentée par toute sous-classe.

La méthode statique connect prend un nom de machine, un numéro de port et un type de connection en paramètre; elle initialise une session, vérifie si la connection est de type valide, puis appelle une autre méthode pour tenter une connection. Si la connection s'effectue avec succès, un objet de type Session est retourné.

La méthode attemptConnection, de type boolean, prend en paramètre un type de connection (de type byte) et un nombre de tentatives de connection (de type int); elle essaie d'envoyer une frame de connection et de recevoir en réponse une frame ACK, soit jusqu'à ce qu'elle réussisse, soit jusqu'à ce que le nombre maximum de tentatives ait été atteint. Elle retourne vrai en cas de connection et faux en cas d'échec de connection.

Les méthodes notifyFrameReceived et notifyIOException ont les même utilités que décrites précédemment.

Quant à la méthode close, elle envoi une frame de déconnection au le Receiver et libère les ressources.

5.2 Classe: GoBackNSession

Cette classe étend la classe Session et supplante la méthode send, de type boolean, qui prend un input stream en paramètre. Elle implémente l'envoi de frames selon le protocole Go-Back-N. Essentiellement, grâce à la structure de données NumWindow, on remplit la fenêtre d'anticipation avec les prochaines frames à envoyer, pour ensuite pop et envoyer la prochaine frame dans la fenêtre. On vérifie ensuite si le temporisateur du premier élément à accepter a expiré, dans quel cas on effectue un go back n sur celui-ci, et finalement on regarde les frames reçues.

5.3 Classe: SelectiveRejectSession

Cette classe étend la classe Session et supplante la méthode send, de type boolean, qui prend un input stream en paramètre. Nous n'avons pas implémenté cette version dans ce projet car ceci n'était pas demandé, mais l'avons inclus pour amélioration ultérieure.

5.4 Classe: StopAndWaitSession

Cette classe étend la classe Session et supplante la méthode send, de type boolean, qui prend un input stream en paramètre. Essentiellement, cette méthode lis des bytes du input stream, puis initialise et envoie une frame d'information au récepteur, qui sera renvoyée si une frame ACK n'est pas reçue dans le délai prescrit. Notons que la valeur NUM de la frame ACK reçue doit aussi être du même type que la valeur NUM de la frame d'information envoyée, sinon cette dernière sera également renvoyée.

Nous avons implémenté ce protocole simplement car il est très facile et nous permettait de tester le reste du projet sans embûches.

6. Paquet: utils

6.1 Classe: BitInputStream

Cette classe simule la réception de données, bit par bit, provenant d'un byte stream. Elle a comme variables un input stream contenant des données à lire; une variable current, de type byte, qui représente bien sûr le byte qu'on est en train de lire; et une variable pos, de type int, qui représente notre position présente à l'intérieur du byte.

La méthode readBit, de type byte, fait exactement ce qu'on son nom indique à l'aide d'opérations bitwise.

6.2 Classe: BitOutputStream

Cette classe simule l'écriture de données, bit par bit, dans un byte stream. Elle a trois variables: un output stream prêt à accueillir des données, et deux variables, nommées current et pos, qui fonctionnent exactement comme les variables de la classe BitInputStream.

La méthode writeBit, de type void, prend en paramètre un byte à écrire dans le stream. Cependant, ce byte n'est pas pris d'un coup; on simule la réception d'un bit à la fois à l'aide d'opération bitwise et, lorsque la valeur de pos est égale à 8 (bits), on a un byte complet, et c'est alors qu'on écrit ce byte dans le output stream.

La classe présente dispose aussi d'une méthode flush, de type void, qui sert à écrire, dans le stream, un dernier byte dans le cas où ce dernier aurait un nombre de bits qui ne serait pas un multiple de 8 (autrement dit, si on arrête de construire notre dernier byte alors que la valeur de pos est plus grande que 0, mais plus petite que 8. Bien sûr, une fois que la méthode flush a fini d'écrire le dernier byte, elle vide le output stream par précaution.

6.3 Classe: Log

Nous avons ici une constante isVerbose, de type boolean, qui sert à distinguer entre les communications détaillées et standard du programme suite à son exécution. Correspondant à ces deux types de communication, la classe a deux méthodes, verbose et println, qui sont chacune de type void et prennent un String en paramètre.

6.4 Classe: NumWindow

Cette classe implémente une fenêtre d'anticipation de frames à envoyer, utile pour le protocole Go-Back-N.

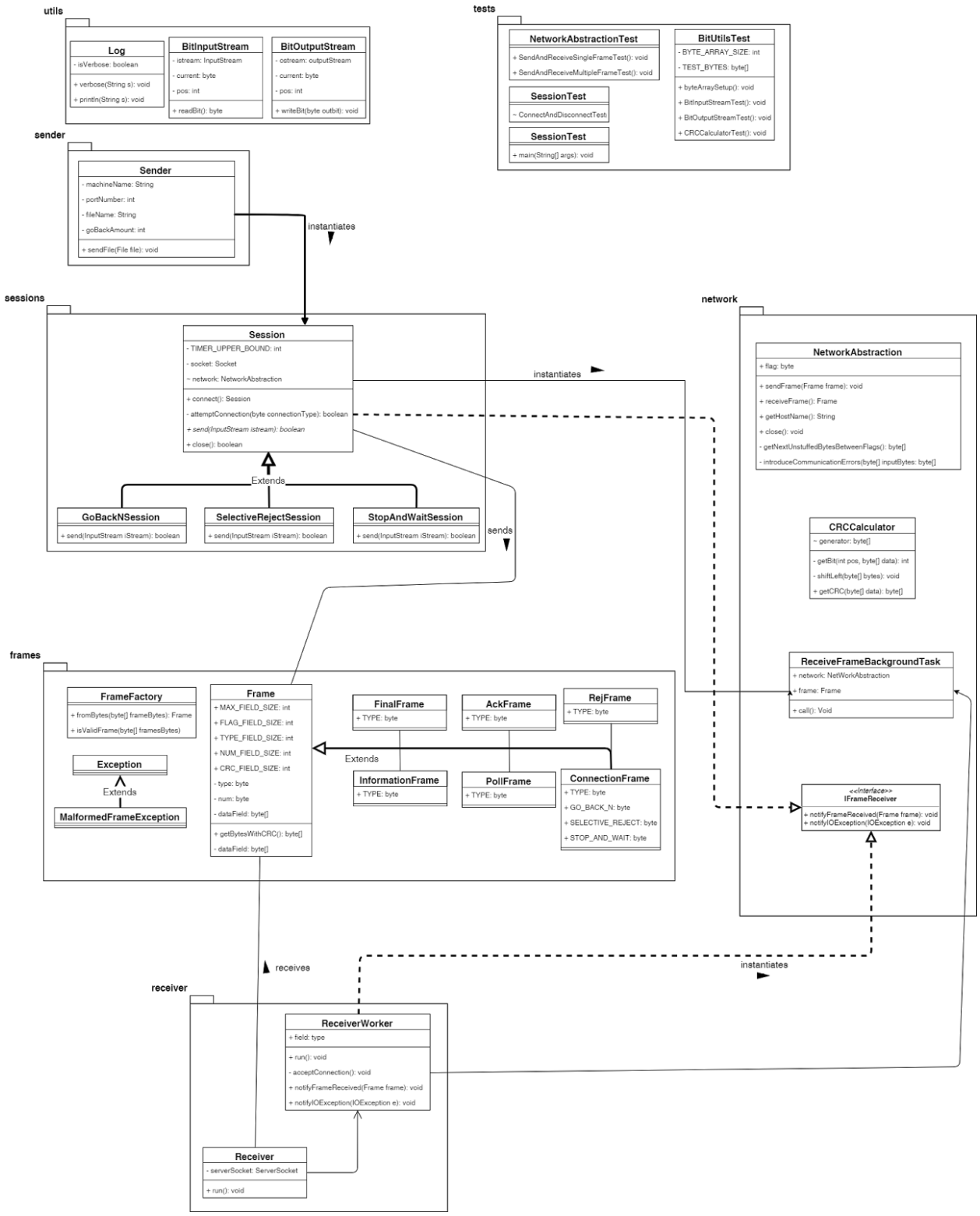
Les noms des méthodes sont aussi indicateurs et chaque méthode est assez simple. Le principe est que la variable currentAck pointe sur la dernière frame acceptée et currentToSend pointe sur la prochaine frame à envoyer. Faire un go-back-n revient à remettre ces deux pointeurs égaux et à recommencer.

7. Paquet: tests

Nous avons testé le plus possible notre implémentation à chaque étape du développement pour s'assurer de la validité de nos objets et méthodes. Ce paquet est en quelque sorte un artéfact de notre conception.

Ce paquet contient évidemment les classes de test JUnit, qui formes aussi un exemple d'utilisation pour les classes implémentées. Par exemple, le test GoBackNSendTextTest démontre comment envoyer un Stream de données d'un Sender vers un Receiver.

8. Diagramme de classes


utils
Log
- isVerbose: boolean
+ verbose(String s): void
+ println(String s): void
BitInputStream
- istream: InputStream
- current: byte
- pos: int
+ readBit(): byte
BitOutputStream
- ostream: outputStream
- current: byte
- pos: int
+ writeBit(byte outbit): void
tests
NetworkAbstractionTest
+ SendAndReceiveSingleFrameTest(): void
+ SendAndReceiveMultipleFrameTest(): void
SessionTest
~ ConnectAndDisconnectTest
SessionTest
+ main(String[] args): void
BitUtilsTest
- BYTE_ARRAY_SIZE: int
- TEST_BYTES: byte[]
+ byteArraySetup(): void
+ BitInputStreamTest(): void
+ BitOutputStreamTest(): void
+ CRCCalculatorTest(): void
sender
Sender
- machineName: String
- portNumber: int
- fileName: String
- goBackAmount: int
+ sendFile(File file): void
instantiates
sessions
Session
- TIMER_UPPER_BOUND: int
- socket: Socket
~ network: NetworkAbstraction
+ connect(): Session
- attemptConnection(byte connectionType): boolean
+ send(InputStream istream): boolean
+ close(): boolean
Extends
GoBackNSession
+ send(InputStream iStream): boolean
SelectiveRejectSession
+ send(InputStream iStream): boolean
StopAndWaitSession
+ send(InputStream iStream): boolean
sends
instantiates
network
NetworkAbstraction
+ flag: byte
+ sendFrame(Frame frame): void
+ receiveFrame(): Frame
+ getHostName(): String
+ close(): void
- getNextUnstuffedBytesBetweenFlags(): byte[]
- introduceCommunicationErrors(byte[] inputBytes: byte[]
CRCCalculator
~ generator: byte[]
- getBit(int pos, byte[] data): int
- shiftLeft(byte[] bytes): void
+ getCRC(byte[] data): byte[]
ReceiveFrameBackgroundTask
+ network: NetWorkAbstraction
+ frame: Frame
+ call(): Void
«Interface»
IFrameReceiver
+ notifyFrameReceived(Frame frame): void
+ notifyIOException(IOException e): void
frames
FrameFactory
+ fromBytes(byte[] frameBytes): Frame
+ isValidFrame(byte[] framesBytes)
Exception
Extends
MalformedFrameException
Frame
+ MAX_FIELD_SIZE: int
+ FLAG_FIELD_SIZE: int
+ TYPE_FIELD_SIZE: int
+ NUM_FIELD_SIZE: int
+ CRC_FIELD_SIZE: int
- type: byte
- num: byte
- dataField: byte[]
+ getBytesWithCRC(): byte[]
- dataField: byte[]
FinalFrame
+ TYPE: byte
AckFrame
+ TYPE: byte
RejFrame
+ TYPE: byte
Extends
InformationFrame
+ TYPE: byte
PollFrame
+ TYPE: byte
ConnectionFrame
+ TYPE: byte
+ GO_BACK_N: byte
+ SELECTIVE_REJECT: byte
+ STOP_AND_WAIT: byte
receives
instantiates
receiver
ReceiverWorker
+ field: type
+ run(): void
- acceptConnection(): void
+ notifyFrameReceived(Frame frame): void
+ notifyIOException(IOException e): void
Receiver
- serverSocket: ServerSocket
+ run(): void