

# Rapport Projet SEOC Réseau

## Client Bittorrent

Equipe

Superviseur

William GOULOIS  
Souha TIBI  
Aiman Wahdi MOHD IMRAN

Olivier ALPHAND

<b>1) Implémentation</b>	<b>2</b>
1.1) Tableau de fonctionnalités	2
1.2) Diagramme de classe UML	3
1.3) Traitement des messages	5
1.4) Gestion connexion/buffer	6
1.5) Algorithme de sélection des pièces + Endgame	7
<b>2) Performances</b>	<b>9</b>
2.1) Scénario de test	9
2.2) Test de performance	9
2.2.1) Caractéristiques du PC/Réseau et du torrent	9
2.2.2) Calcul du débit	10
2.2.3) Tableau des performances	11
<b>3) Tests et validation</b>	<b>11</b>
3.1) Fonctionnalités testées et méthodes de validation	11
3.2) Bugs résolus, difficultés/subtilités qui ont coûté du temps	12
<b>4) Bonnes pratiques</b>	<b>13</b>
4.1) Etapes du travail d'architecture/factorisation	13
4.2) Design Pattern, Clean Code	14
4.2.1) 2 Visiteurs (réception et envoi des messages)	14
4.2.2) Fabrique (réception des messages)	14
4.2.3) Adaptateur (ProgressBarArray)	15
4.2.4) Clean Code	16
4.3) Scrum, Gitlab et Git	17
4.3.1) Méthode Scrum	17
4.3.2) Utilisation de git	17
<b>5) Organisation du travail</b>	<b>18</b>
5.1) Répartition des tâches	18
5.2) Enchaînement des événements	18
5.3) Difficultés d'organisation rencontrées	18
5.4) Retours personnels de chaque membre	19

# 1) Implémentation

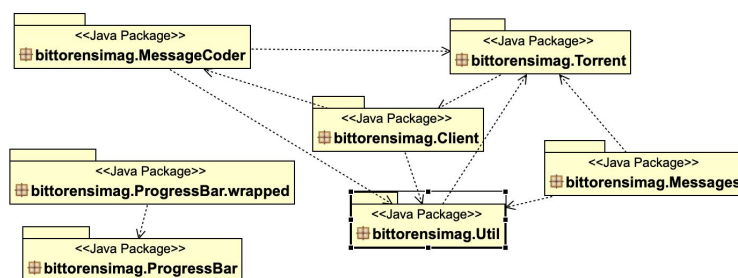
## 1.1) Tableau de fonctionnalités

Sprint	Fonctionnalité	Avancement (précision)
<b>Sprint 1</b>	Tracker (HTTP Req URLEncodé/Réponse de-béncodée)	OK
Leecher 0%	Socket bloquante + java.io	OK
Leecher 0%	Ecriture pièce sur disque	OK
Leecher 0%	Plusieurs messages bittorrent au sein d'un segment TCP	KO
Leecher 0%	Message inconnu traité	OK
Leecher 0%	Sélection de pièces	Séq
Seeder 100%	bitfield créé en f° du torrent	OK
Seeder 100%	gestion pièces et blocs	OK
Seeder 100%	Clients supportés	Vuze, aria2c, transmission,q bittorent
<b>Sprint 2</b>	support Multi-leecher (socket non bloquante OP_ACCEPT )	OK
	support Multi-seeder	OK
	ByteBuffer (~1 par msg ou ~1 par peer)	1/message
	Sélection de pièces : Conditions Sprint 2 respectées	OK
	Eval. de perf	local,perso +remote,ensim ag
	Resume calculé à partir du fichier partiel	OK
	Scénario multi-leecher supporté	OK
	Gestion concurrence Thread	KO
<b>Sprint 3</b>	Design Patterns	2 Visiteurs Adaptateur Fabrique
	Sélection de pièces	Rarest first
	Message supporté HAVE, CANCEL, KEEPALIVE	Have, Cancel
	Clean Code, Factorisation, Archi. Objet,	Fact, Arch Obj
<b>Avancé</b>	Tests automatisés	~OK lancement

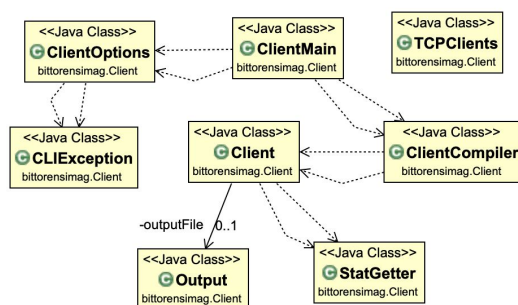
		manuel mais automatique
	Calcul débit, Barre progression	Progression, Temps restant
	Compilation automatique	Maven
	Taille de pièce (32K et +)	OK
	Tracker contacté à intervalle régulier	OK
	Extensions	Endgame 1%

## 1.2) Diagramme de classe UML

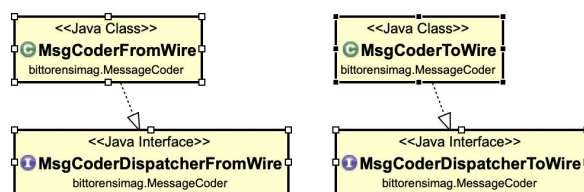
Vue Globale :



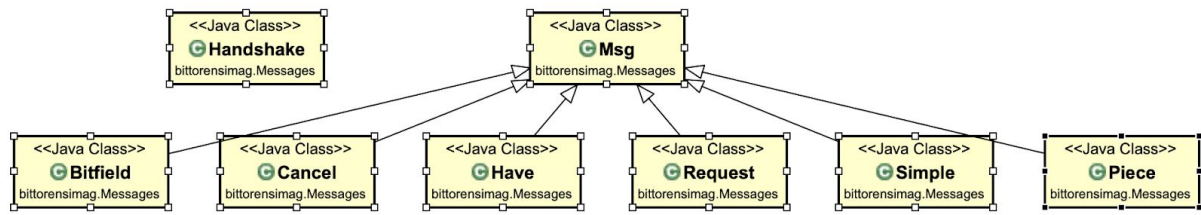
Package Client :



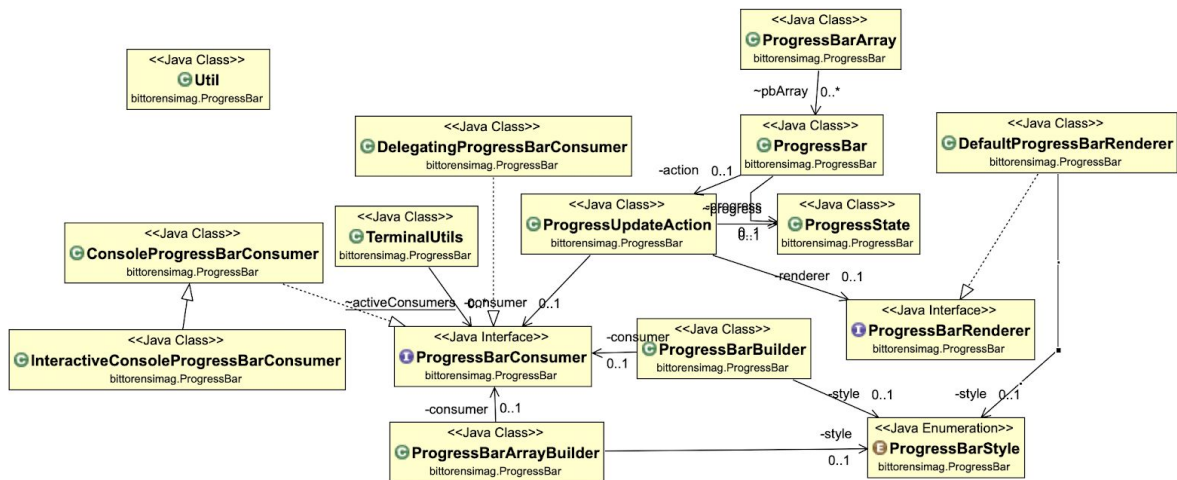
Package MessageCoder :



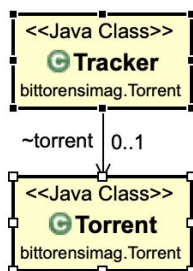
Package Messages :



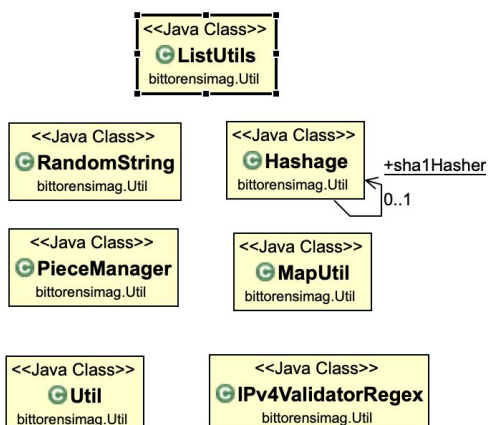
## Package ProgressBar



## Package Torrent :



## Package Util :



## 1.3) Traitement des messages

Pour traiter les messages nous avons choisi d'utiliser un switch (machine à états) qui se trouve dans la fonction `receivedMessage` de la classe `Client` :

```
switch (msgType) {  
    case Simple.CHOKE: ...  
    case Simple.UNCHOKE: ...  
    case Simple.INTERESTED: ...  
    case Simple.NOTINTERESTED: ...  
    case Have.HAVE_TYPE: ...  
    case Bitfield.BITFIELD_TYPE: ...  
    case Request.REQUEST_TYPE: ...  
    case Piece.PIECE_TYPE: ...  
    default: ...  
}
```

Dans chaque "case" nous avons une fonction (handler) qui permet selon le contenu du message de choisir l'action correspondante. Seul le handshake est traité avant car il n'a pas la même construction que les autres messages. Exemple du handler lors de la réception d'un handshake :

```
private boolean handleHandshake(Handshake handshake, SocketChannel clntChan, ProgressBarArray torrentProgressBars)  
    throws IOException {  
    LOG.debug("Handling Handshake message");  
    if (handshake.getSha1Hash().compareTo(this.torrent.info_hash) != 0) {  
        LOG.error("Sha1 hash received different from torrent file");  
        return false;  
    }  
    String peerId = new String(handshake.getPeerId());  
    this.socketToPeerIdMap.put(clntChan, peerId);  
  
    if (Logger.getRootLogger().getLevel() == Level.INFO) {  
        torrentProgressBars.getByIPPort(clntChan).setExtraMessage(peerId);  
    }  
  
    if (!this.handshakeSent.contains(clntChan)) {  
        Handshake.sendMessage(this.torrent.info_hash, clntChan);  
        this.handshakeSent.add(clntChan);  
    }  
    Bitfield.sendMessage(Bitfield.ourBitfieldData, clntChan);  
    return true;  
}
```

Explication de ce handler :

Après avoir vérifié que le handshake concerne bien le même fichier torrent, on ajoute le `peerId` sur sa barre de progression et dans une hashmap qui permet de faire le lien entre les channels et les noms de `peerId`. Il suffit ensuite de vérifier si nous avons déjà envoyé le handshake à ce pair et de lui envoyer notre bitfield.

## 1.4) Gestion connexion/buffer

La connexion est gérée par la fonction `startCommunication` de la classe `Client` :

Cette fonction tente d'abord d'établir les connexions avec tous les pairs trouvés depuis le tracker (via la fonction `connectToAllClients`) qui récupère la première entrée de la hashmap qui crée avec la réponse du tracker. Cette hashmap met en relation IP et PORT des pairs. Tous les pairs sont donc regroupés avec la même adresse IP et des ports différents.

Puis elle va créer une `ServerSocket` sur le port 6881 afin d'accepter les nouvelles demandes de pairs qui rejoignent le swarm après le lancement de notre propre client.

```
ServerSocketChannel ssc = ServerSocketChannel.open();

// Set it to non-blocking, so we can use select
ssc.configureBlocking(false);

// Get the Socket connected to this channel, and bind it to the
// listening port
ServerSocket ss = ssc.socket();
InetSocketAddress isa = new InetSocketAddress(PORT);
ss.bind(isa);

// Register the ServerSocketChannel, so we can listen for incoming
// connections
ssc.register(selector, SelectionKey.OP_ACCEPT);
LOG.debug("Listening on port " + PORT + " for incoming connections");
```

Puis on rentre dans le traitement des différentes clés :

```
while (keyIter.hasNext()) {
    SelectionKey key = keyIter.next();
    // if we receive a new connection from a new client
    if (key.isAcceptable()) { ...
    if (key.isReadable()) {
        // LOG.debug("Readable key");
        // Client socket channel is available for writing
        if (key.isWritable()) {
            // LOG.debug("Writable key");
            try {
                SocketChannel cIntChan = (SocketChannel) key.channel();
                if (!this.receiveMsg(this.coderToWire, this.coderFromWire, cIntChan, torrentProgressBars,
                    pbCPU,
                    pbMemory)) {
                    this.closeConnection((SocketChannel) key.channel(), torrentProgressBars);
                }
            } catch (IOException ioe) {
                LOG.error("Error handling client: " + ioe.getMessage());
            }
        }
    }
}
```

Selon l'opération que la clé supporte, la connexion sera acceptée ou le message envoyé au switch vu plus haut pour traiter le message.

## 1.5) Algorithme de sélection des pièces + Endgame

L'algorithme de sélection des pièces est géré par la classe `PieceManager`. Cette classe contient les informations des pièces des clients tels que les pièces manquantes, les pièces demandées et la disponibilité des pièces dans tel client.

Après avoir implémenté l'algorithme séquentiel lors du sprint 2, nous avons choisi de passer à un algorithme `rarest first`. Lorsqu'un client nous envoie son `bitfield`, nous ajoutons ses pièces dans une `HashMap` qui fait le lien entre numéro de pièce et clients qui ont cette pièce. La fonction `sortBySize` dans la classe `MapUtil` permet de classer dans cette `HashMap` les pièces selon le nombre de clients. Nous nous en servons comme référence pour ensuite classer une `ArrayList` des pièces manquantes dans la fonction `sortPiecesNeeded` de la classe `PieceManager`.



```

public void sortPiecesNeeded() {
    Set<Integer> sortedPieces = pieceMap.keySet();
    List<Integer> sortedList = new ArrayList<Integer>();
    sortedList.addAll(sortedPieces);
    Collections.sort(needed, Comparator.comparing(item -> sortedList.indexOf(item)));
}

```

Nous avons aussi choisi d'implémenter l'extension Endgame. Le déclenchement se fait lorsqu'il reste moins de 1% des pièces à télécharger. Toutes les pièces sont pendant l'Endgame demandées à tous les clients connectés et lorsqu'une pièce n'est plus nécessaire un message cancel est broadcasté à tous les autres pairs.

```

// send have for this piece to all other peers
for (SocketChannel channel : this.peersConnected) {
    Have.sendMessage(pieceIndex, channel);
    // send cancel to all other channels
    if (this.pieceManager.getEndgameStatus() && clntChan != channel) {
        Cancel.sendMessageForIndex(pieceIndex, channel);
    }
}
// set this piece downloaded
this.pieceManager.pieceNoMoreNeeded(pieceIndex);

```

L'algorithme ci-dessous détermine la prochaine pièce à demander à un pair lorsque notre client reçoit un message unchoke ou qu'une pièce complète vient d'être reçue.

```

public int nextPieceToRequest(Socket currentPeer) {
    // Rule for endgame ? 1 last percent here
    if (needed.size() < Torrent.numberOfPieces * 0.01) {
        this.beginEndgame();
    }
    boolean peerHasPieceWeNeed = false;
    int nextPiece = -1;
    for (int i = 0; i < needed.size(); i++) {
        nextPiece = needed.get(i);
        // if request has already been sent or peer does not have nextPiece
        if (pieceMap.containsKey(nextPiece)) {
            if (isRequested(nextPiece) || !pieceMap.get(nextPiece).contains(currentPeer)) {
                continue;
            } else {
                peerHasPieceWeNeed = true;
                break;
            }
        }
    }
    if (!peerHasPieceWeNeed) { // if peer has no piece we need, return -1
        return -1;
    } else {
        return nextPiece;
    }
}

```

## 2) Performances

### 2.1) Scénario de test

Nous avons réalisé des tests en local et en remote. L'implémentation des tests en remote sur les machines de l'Ensimag était assez complexe, d'autant plus qu'il fallait avoir une fenêtre graphique pour pouvoir mesurer les performances avec wireshark.

Nous avons donc 2 sets de tests différents : en local nous avons écrit des tests fonctionnels avec plusieurs scénarios et en remote deux tests (monoleecher et monoseeder).

Voir le [README.md](#) pour plus de détails et une marche à suivre pour tester sur sa propre machine/ une machine de l'ensimag. Nous détaillons ici seulement ce qui a été réalisé et le déroulement général.

#### 2.1.1) Tests locaux

Pour un test en local, voici le scénario typique de test :

- initEnv.sh démarre l'environnement pour les tests (nettoyage des anciens lancements, démarre opentracker, lancement webserveur aria2c, lancement de wireshark, lancement de firefox pour monitorer)
- démarrage du scénario, qui peut prendre la forme de mutli-seeders.sh, multi-leechers.sh, scenario\_c1%\_c2%\_c3%\_fichier.sh.
- on démarre notre client (depuis le jar ou le script de lancement sous bin)

Lors de l'exécution d'un scénario, un dossier de logs contient les logs de : notre client, chaque client aria2c lancé, la capture wireshark, le webserver aria2.

#### 2.1.2) Tests remote

Pour le test en remote, voici le scénario typique de test :

- On copie le fichier de test\_remote sur une machine de l'ensimag (qui sera le seeder)
- On change dans le script les numéros de l'ensipc pour le seeder et leecher
- On ssh -X (pour pouvoir lancer des applications graphiques) dans les deux machines choisies et on lance wireshark pour monitorer
- On démarre le script en choisissant d'installer les dépendances sur les deux machines

Le test par défaut sera avec notre client en leecher et aria2c en seeder. Pour changer ce comportement il faut déplacer le fichier complet de la machine seeder vers la machine leecher.

## 2.2) Test de performance

### 2.2.1) Caractéristiques du PC/Réseau et du torrent

Local : Lenovo Ideapad Flex 5 14ARE05 sous Manjaro (basé sur Arch Kernel 5.10.7)



- AMD Ryzen 7 4700U (8C / 8T, Base:2.0 Boost:4.1GHz, 4MB L2 / 8MB L3)
- 16 Go DDR4-3200
- Wifi Realtek RTL8822CE 2x2 ac
- 1TB SSD Samsung PM981a M.2 PCIe 2280

Réseau local (localhost)

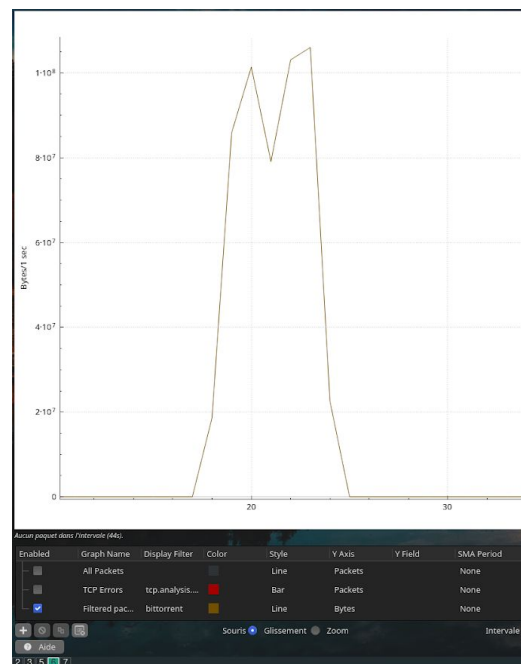
Remote : 2 PCs de l'ensimag connectés via ethernet 100Mb/s

## 2.2.2) Calcul du débit

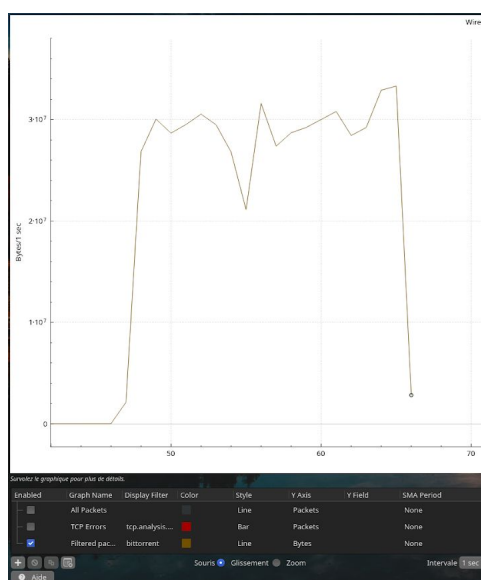
Local :



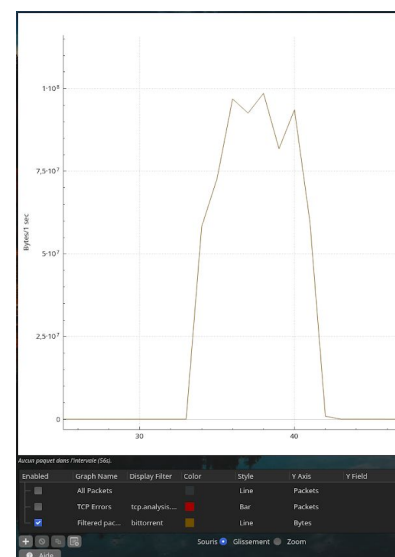
Leecher 0% avec 1 Aria2c 100% Seeder



Seeder 100% avec 1 Aria2c Leecher 0%

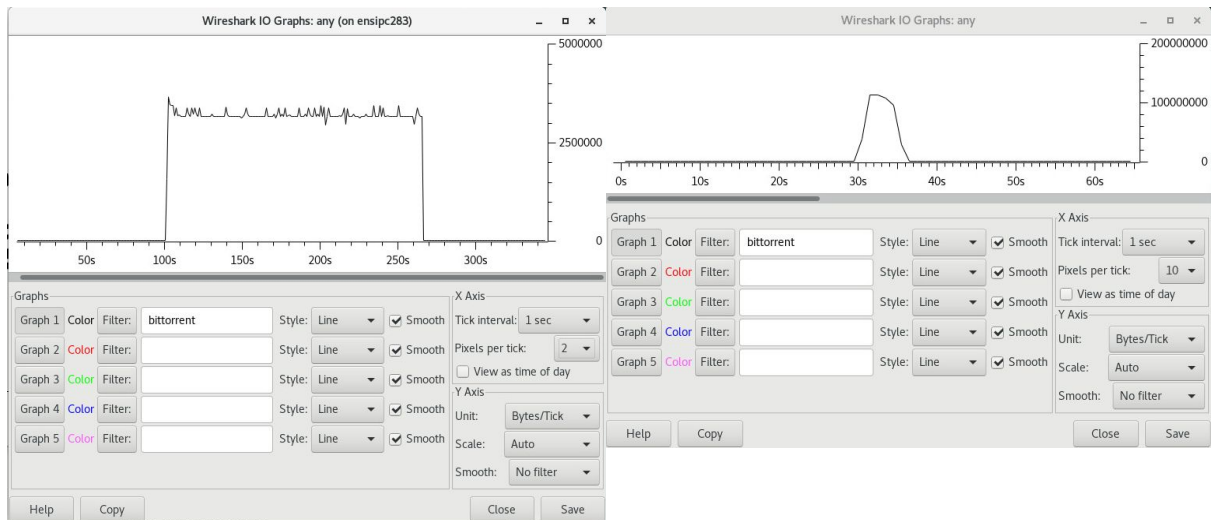


Leecher 0% avec 3 Aria2c Seeder 100%



Seeder 100% avec 3 Aria2c Leecher 0%

Remote :



Leecher 0% avec 1 Aria2c 100% Seeder

Seeder 100% avec 1 Aria2c Leecher 0%  
(faux selon nous)

### 2.2.3) Tableau des performances

Description de l'expérience	Débit
Votre client (0%) Vs 1 aria2c (100%) en remote (500M)	3,2 Mb/s
Votre client (100%) Vs 1 aria2c (0%) en remote (500M)	9,3 Mb/s
Votre client (0%) Vs 3 aria2c (100%) en local (500M)	30 Mb/s
Votre client (100%) Vs 3 aria2c (0%) en local (500M)	85 Mb/s

## 3) Tests et validation

### 3.1) Fonctionnalités testées et méthodes de validation

Le fonctionnement global du client peut être testé manuellement via des scripts shells dans le dossier src/test/script. Un test de script permet de vérifier le bon fonctionnement de l'interface CLI du client et les autres permettent d'automatiser au maximum le test des fonctionnalités :

- Les scripts qui commencent par clean permettent de nettoyer l'environnement, arrêter les clients en tâche de fond, fermer wireshark...
- InitEnv permet d'initialiser tout l'environnement de test (opentracker, wireshark, aria2c webserver)
- cobertura-report qui permet de vérifier la couverture des tests du client (inutile car les tests ne sont pas inclus dans maven)

- multi-leechers et multi-seeders qui permet de lancer le nombre souhaité de clients aria2c en leechers ou seeders respectivement.
- 3 scripts qui permettent de démarrer des scénarios particuliers (leechers 50%)

La vérification du bon transfert des fichiers via le protocole bittorrent se faisait avec l'interface web d'aria2c pour voir que les clients leechers sont bien passés en seeder. Pour notre client nous pouvions le relancer afin qu'il vérifie que tous les hash des parties du fichier téléchargé sont bien identiques aux hash stockés dans le torrent.

### 3.2) Bugs résolus, difficultés/subtilités qui ont coûté du temps

De nombreux bugs ont été rencontrés pendant le développement et il est difficile de se souvenir de l'ensemble des difficultés rencontrées sur le projet. Voici les parties qui ont posé le plus de problèmes :

- vérifications des hash des pièces du torrent
- l'Urlencode du hash de info pour le dialogue au tracker
- le support de la dernière pièce du fichier de taille plus petite/ en moins de parties qui oblige à rajouter ce cas lors de la requête, l'envoi la vérification...
- le remplissage du bitfield de façon dynamique
- adaptation des barres de progression sur plusieurs lignes
- les readByte et readUnsignedByte
- trailing 0 pour les total length
- les conversions bytes en hexa en byteArray
- littleEndian et bigEndian
- scripts shells pour les tests
- implémentation du logger dans tout le projet
- restructuration pour implémenter des design patterns
- parsing des messages bittorrents inconnus (continuous data) envoyés par les clients (readSafe method)
- changement de transmission vers aria2c

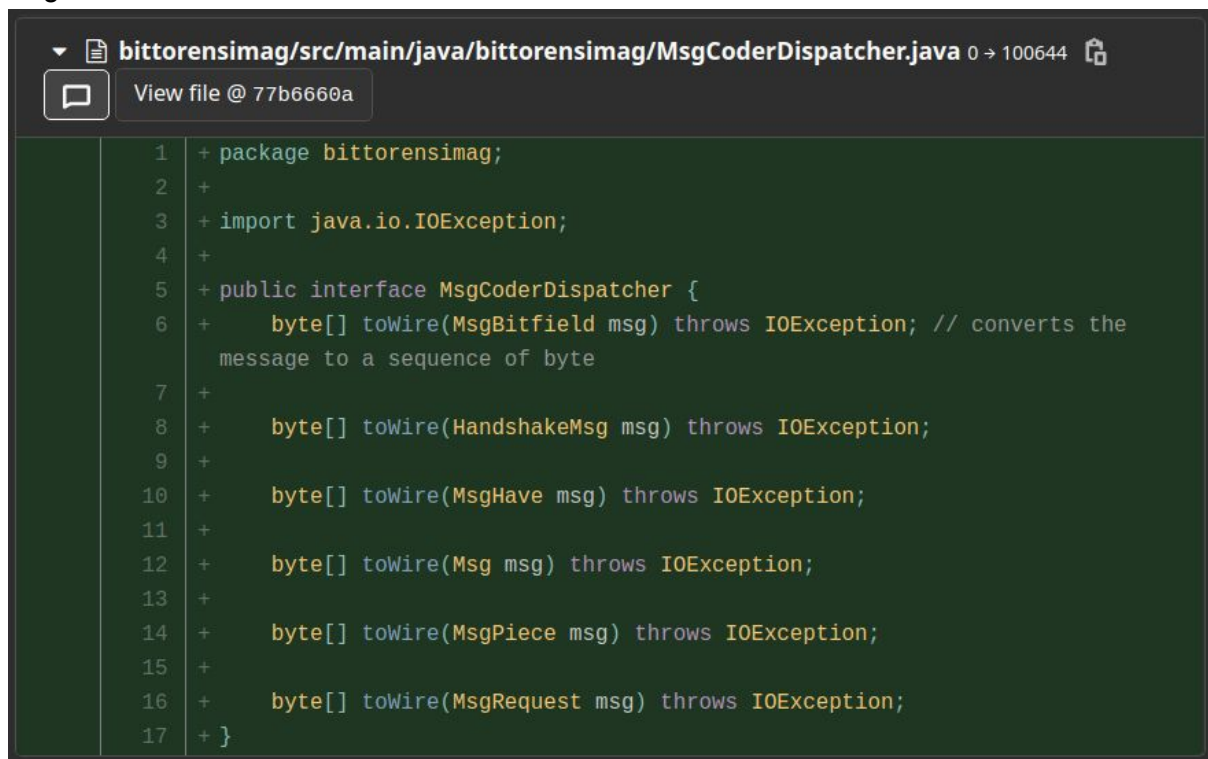
## 4) Bonnes pratiques

### 4.1) Etapes du travail d'architecture/factorisation

Le meilleur exemple du travail d'architecture est l'implémentation du design pattern Visiteur ainsi que de la machine à états.

Commit correspondant :  
<https://gitlab.ensimag.fr/projet-reseau/2020-2021/equipe4/-/commit/77b6660aaffa99b3158559f9542d5a0a5afca2d0>

Ce commit assez imposant a permis de factoriser toutes les classes de codage des messages dans une seule interface `MsgCoderDispatcherToWire` qui est implémentée par `MsgCoderToWire`.



```
1 + package bittorensimag;
2 +
3 + import java.io.IOException;
4 +
5 + public interface MsgCoderDispatcher {
6 +     byte[] toWire(MsgBitfield msg) throws IOException; // converts the
        message to a sequence of byte
7 +
8 +     byte[] toWire(HandshakeMsg msg) throws IOException;
9 +
10 +    byte[] toWire(MsgHave msg) throws IOException;
11 +
12 +    byte[] toWire(Msg msg) throws IOException;
13 +
14 +    byte[] toWire(MsgPiece msg) throws IOException;
15 +
16 +    byte[] toWire(MsgRequest msg) throws IOException;
17 + }
```

Lorsqu'un message doit être envoyé, nous pouvons faire appel à `dispatcher.toWire(Message)` comme montré ci-contre :



```
byte[] dataBitfield = { 0, 0 };

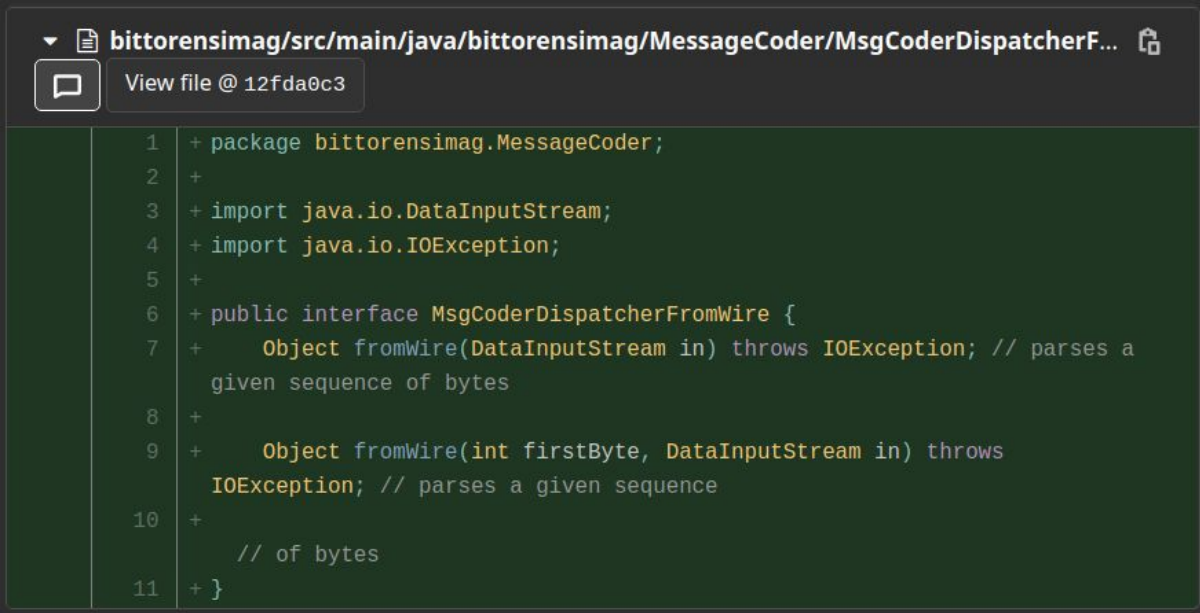
MsgBitfield msgBitfield = new MsgBitfield(3, 5, dataBitfield);
Msg msgInterested = new Msg(1, 2);

frameMsg(coder.toWire(msgBitfield), Out);
frameMsg(coder.toWire(msgInterested), Out);
```

Puis nous avons ajouté le même fonctionnement pour la réception des messages avec `MsgCoderDispatcherFromWire` :

commit

<https://gitlab.ensimag.fr/projet-reseau/2020-2021/equipe4/-/commit/12fda0c3f06a249adc6a96bcaf48006e3fab51a9>



The screenshot shows a GitLab commit view for the file `bittorensimag/src/main/java/bittorensimag/MessageCoder/MsgCoderDispatcherF...` at commit `12fda0c3`. The code is in Java and defines a public interface `MsgCoderDispatcherFromWire` with two methods: `Object fromWire(DataInputStream in) throws IOException;` and `Object fromWire(int firstByte, DataInputStream in) throws IOException;`. The interface is intended for parsing a given sequence of bytes.

```
1 + package bittorensimag.MessageCoder;
2 +
3 + import java.io.DataInputStream;
4 + import java.io.IOException;
5 +
6 + public interface MsgCoderDispatcherFromWire {
7 +     Object fromWire(DataInputStream in) throws IOException; // parses a
      given sequence of bytes
8 +
9 +     Object fromWire(int firstByte, DataInputStream in) throws
      IOException; // parses a given sequence
10 +
      // of bytes
11 + }
```

Le switch a été intégré dans ce commit :

<https://gitlab.ensimag.fr/projet-reseau/2020-2021/equipe4/-/commit/f3903b36e77150c3b841339d3672a1bbc31c7210>

## 4.2) Design Pattern, Clean Code

### 4.2.1) 2 Visiteurs (réception et envoi des messages)

2 Visiteurs ont été créés afin de pouvoir envoyer le message selon sa classe. La partie précédente (partie 4.1) détaille déjà bien leur implémentation.

### 4.2.2) Fabrique (réception des messages)

Lorsqu'un message est reçu, il est traité par un visiteur qui est aussi une fabrique, il renvoie le message qu'il caste dans la bonne classe. La fonction `fromWire()` de la classe `MsgCoderFromWire` renvoie donc un objet qui dépendra du message reconnu.

Elle a été implémentée dans ce commit :

<https://gitlab.ensimag.fr/projet-reseau/2020-2021/equipe4/-/commit/12fda0c3f06a249adc6a96bcaf48006e3fab51a9>

```

switch (type) {
    case Simple.CHOKE:
        return new Simple(Simple.CHOKE);
    case Simple.UNCHOKE:
        return new Simple(Simple.UNCHOKE);
    case Simple.INTERESTED:
        return new Simple(Simple.INTERESTED);
    case Simple.NOTINTERESTED:
        return new Simple(Simple.NOTINTERESTED);
    case Have.HAVE_TYPE:
        int index = this.readLengthInt(clntChan, 4);
        if (index < 0) {
            return null;
        } else if (index > Torrent.numberOfPieces) {
            LOG.error("Received have for invalid index : " + index);
            return null;
        } else {
            return new Have(index);
        }
}

```

#### 4.2.3) Adaptateur (ProgressBarArray)

Nous avons choisi pour afficher l'avancée du téléchargement de notre client, des autres clients ainsi que des performances de se servir d'une implémentation de ProgressBar en java trouvée sur Github :

<https://github.com/ctongfei/progressbar>

Une fonctionnalité permet déjà d'avoir plusieurs barres de progressions avec un concept introduit dans java-7 qui permet de créer des objets dans un try qui seront détruits lors de la fin de ce block. Elle s'appelle try-with-resources. Les objets instanciés de cette manière doivent implémenter une interface AutoCloseable qui détermine la manière de détruire (close) cet objet.

Initialement les tableaux de taille dynamique ne sont pas supportés par ce projet. Nous avons donc adapté cette librairie avec une nouvelle classe ProgressBarArray qui permet d'utiliser un tableau de progressbar. Ceci permet d'ajouter dynamiquement des nouvelles barres de progression lors de la connexion avec de nouveaux clients.

Le commit correspondant à cette implémentation :

<https://gitlab.ensimag.fr/projet-reseau/2020-2021/equipe4/-/commit/f2939525c6e18830f11590d68c3d7ea5fbf2893f>

On peut voir qu'on instancie donc l'adaptateur comme ressource du try et qu'on peut ajouter facilement des nouvelles progressbar :

```

private void startProgressBars() {
    try (ProgressBarArray torrentProgressBars = this.createTorrentProgress();
        ProgressBar pbCPU = new ProgressBarBuilder().setTaskName("CPU").setInitialMax(100)
            .setStyle(ProgressBarStyle.UNICODE_BLOCK).build();
        ProgressBar pbMemory = new ProgressBarBuilder().setTaskName("MEMORY")
            .setInitialMax((long) StatGetter.getTotalMemory()).setStyle(ProgressBarStyle.UNICODE_BLOCK)
            .setUnit("MB", 1).build()) {
    }
}

```



Et on peut voir ici lorsqu'on accepte un nouveau pair :

```
if (key.isAcceptable()) {
    // It's an incoming connection. Register this socket with
    // the Selector so we can listen for input on it
    SocketChannel clntChan = ss.accept().getChannel();

    LOG.info("Received a new connection from " + clntChan);

    clntChan.configureBlocking(false);

    // Register it with the selector, for reading and writing
    clntChan.register(selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE);
    otherClientsChannels.add(clntChan);
    Handshake.sendMessage(this.torrent.info_hash, clntChan);
    this.handshakeSent.add(clntChan);
    if (Logger.getRootLogger().getLevel() == Level.INFO) {
        torrentProgressBars.add(this.progressBarBuilder,
            clntChan.socket().getRemoteSocketAddress().toString());
    }
}
```

#### 4.2.4) Clean Code

Un gros travail tout au long du projet a été fait pour éviter les fichiers trop volumineux, répartir toutes les fonctions concernant un message dans sa propre classe.

De plus pour avoir du code qui puisse facilement évoluer nous avons utiliser de nombreux valeurs constantes :

```
private static final Logger LOG = Logger.getLogger(Client.class);

public final static String IP = "127.0.0.1";
public final static int PORT = 6881;

private final int KB = 1024;
private final int MB = 1024 * 1024;
private final int GB = 1024 * 1024 * 1024;

public final static int LENGTH = 1;

public final static int CHOKER = 0;
public final static int UNCHOKER = 1;
public final static int INTERESTED = 2;
public final static int NOTINTERESTED = 3;
```

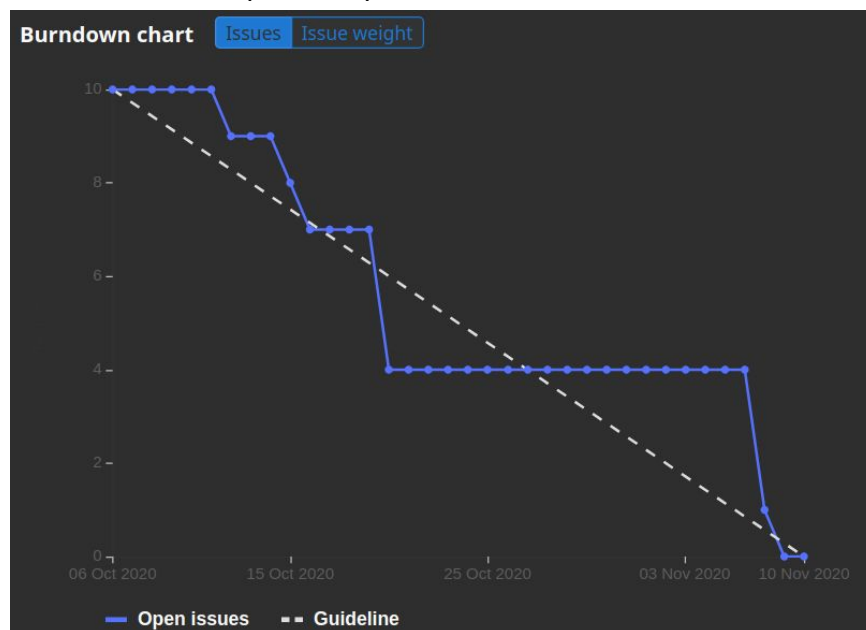
## 4.3) Scrum, Gitlab et Git

### 4.3.1) Méthode Scrum

La méthode scrum a été mise en place pour les 3 sprints avec les différentes étapes pour chaque sprint :

- poker planning afin de définir les étapes du sprint (issues gitlab) et évaluer la complexité (poids)
- les cartes étaient ensuite assignées
- nous avons un groupe de discussion Messenger afin de parler de l'avancée
- un point régulier était fait via discord avec partage d'écran afin que toute l'équipe puisse suivre l'évolution du projet
- lorsqu'une étape est terminée sa carte est déplacée dans closed ce qui actualise le burndown chart

Exemple de la burndown chart pour le Sprint 1 :



### 4.3.2) Utilisation de git

Nous avons des branches selon le type de fonctionnalité implémentée. Liste des branches utilisées tout au long du projet : cli, tracker, messages, seeder, verifyContent (a posé des problèmes), algo, sockets, resume. Les branches étaient ensuite fusionnées avec master après avoir testé les fonctionnalités.

Pour les commits nous n'avons pas utilisé la partie commentaire seulement le titre.

## 5) Organisation du travail

### 5.1) Répartition des tâches

Pour bien mener le projet, notre équipe s'est organisée depuis le premier sprint à faire une répartition initiale des tâches suite à une analyse primaire des objectifs fixés pour le sprint. Les membres choisissent les leurs tâches en faisant une estimation du temps nécessaire pour l'accomplir. Pour des tâches ayant un poids important, il arrive que les autres membres interviennent pour finir les tâches ou pour faire plus de tests. Des objectifs sont également ajoutés au fur et à mesure dans le cas de bugs à résoudre ou de fonctionnalités à rajouter.

### 5.2) Enchaînement des événements

Pour faire un suivi du projet, on a opté pour des réunions en début et à la fin du sprint pour faire le bilan du sprint précédent et pour fixer les objectifs et répartir les tâches pour celui qui arrive. La majorité du projet s'est passée en distanciel donc il était important de trouver un moyen de communication efficace entre les membres du groupe. Pour notre cas, on a choisi "discord" pour organiser nos réunions et discuter sur l'avancement de chaque membre sur sa tâche, ainsi que l'avancement global. Lors du premier sprint, la gestion du temps était moyenne, en effet la partie seeder était faite dans avec un peu de retard par rapport au prévu, mais on a réussi à atteindre le plus part des objectifs avant le délai. Cela nous a appris à mieux gérer notre temps lors du deuxième sprint et à mettre des dates sprint pour les buts principaux et fondamentaux pour éviter de travailler sous pression. On a constaté que cela était fructueux et que les tâches ont été mieux réparties.

### 5.3) Difficultés d'organisation rencontrées

À cause du contexte sanitaire, la moitié du sprint 1 et le reste des sprints se sont passés en distance, donc notre premier défi était de collaborer et de garder une bonne organisation à distance. L'un des problèmes qui est survenu est le manque de communication, en effet parfois on peut ne pas avoir les réponses immédiatement, ce qui peut être un inconvénient lorsqu'il faut agir rapidement, ou lorsqu'un travail dépend d'un autre pour être poursuivi.

En plus, la perte des liens sociaux et du contact direct entre les membres de l'équipe a un peu freiné l'avancement et a empêché de suivre de près le progrès le travail vu que les échanges par messagerie instantanée parfois ne suffisent pas et les informations se perdent en chemin. Cela a eu des conséquences sur la cohésion d'équipe et il fallait faire plus d'effort pour maintenir un bon rythme de travail et pour garder la motivation nécessaire pour travailler.

D'une autre part, la répartition du travail peut s'avérer parfois difficile puisqu'il y a des tâches qui ne sont pas divisibles et doivent être faites par une seule personne pour garantir une cohérence dans le code et dans le raisonnement et pour éviter les conflits dans les branches de gitlab. La fusion des travaux faits séparément était aussi pénible parfois puisqu'on était amenés à travailler dans des contextes ou des hypothèses différents (exemple implémenter la partie de la reprise du téléchargement en travaillant avec les sockets bloquantes et en

parallèle implémenter la migration de la communication mono-client vers multi-clients en utilisant les sockets non bloquantes)

Ainsi, nous constatons qu'il est important de mettre en place dès le début une stratégie efficace pour communiquer et pour suivre l'avancement pour éviter un potentiel relâchement dans l'équipe et afin de respecter les délais des livrables, en plus il faudrait bien utiliser les outils de gitlab pour collaborer et essayer d'avoir une répartition la plus égalitaire en global.

## 5.4) Retours personnels de chaque membre

## 5.4.1) William GOULOIS

### 5.4.1.1) Ce que ce projet m'a appris

Monsieur Roche, lors de l'entrée en 2ème année, nous a dit que le projet GL ferait de nous des ingénieurs accomplis. Dans mon cas, c'est ce projet, de créer un logiciel complet en Java en commençant de rien, qui a eu ce rôle. J'ai pu appliquer directement toutes les leçons que j'avais apprises du projet GL pour réaliser ce projet.

De la recherche documentaire sur le protocole bittorrent à la création d'un environnement de développement via Maven en passant par une interface complète en ligne de commande, j'ai pu apprendre une étape d'un projet que l'on fait peu à l'Ensimag : le déploiement d'un environnement de développement.

J'ai pu appliquer tous les concepts d'architecture objet et plusieurs design patterns ce qui m'a permis d'approfondir ma compréhension des relations entre les classes, l'encapsulation et le principe de clean code.

De plus, étant déjà Scrum Master pendant le projet GL, j'ai pu découvrir les issues, milestones et le burndown chart de Gitlab (nous nous étions servis de Trello) afin de pouvoir suivre l'avancée du projet.

J'ai énormément monté en compétence à la fois en termes de développement, résolution de problème et architecture orientée-objet.

### 5.4.1.2) Ce qui a été le plus difficile

Ayant passé beaucoup de temps à faire avancer ce projet en implémentant des nouvelles fonctionnalités et corrigeant celles sources de problèmes, j'ai pu rencontrer de nombreux bugs. Certains étaient triviaux et nécessitent une simple exécution avec un point d'arrêt et une vérification des variables locales et globales. Mais d'autres m'ont posé de longues heures d'arrachage de cheveux à chercher sur le net (#StackOverflow) les réponses. Je pense notamment à toutes les fonctions regroupées dans le dossier Util du projet qui ont mis en lumière ma faiblesse en tant que programmeur : la manipulation de bits, d'octets, les encodages, conversions de types, les variables signées ou non...

### 5.4.1.3) Des choses à améliorer ?

Je suis fier du produit fini, qu'il réponde au cahier des charges fixé et va même au-delà sur certains points. Ce projet représentait un vrai défi en termes d'investissement personnel surtout dans ces conditions sanitaires qui empêchent toutes interactions dans l'équipe et atteint notre santé mentale.

Pour ce qui est du travail réalisé, des raccourcis ont dû être pris pour rendre un produit qui répond aux exigences. Première victime : les tests. Aucun test unitaire et des tests fonctionnels lancés manuellement. Deuxième victime : la javadoc. Aucune documentation des méthodes/classes et très peu de commentaires. Troisième victime : clean code. Surtout à partir du sprint 2 lorsque je me suis aperçu que je n'aurais pas le temps de produire un code propre et performant suffisamment rapidement. J'aurais aimé pouvoir refactoriser, nettoyer et faire une architecture mieux construite.

Ce projet m'a passionné et si j'avais plus de temps j'aurais encore beaucoup amélioré ce client pour le rendre plus stable, lui permettre de fonctionner avec d'autres trackers (non localhost), ainsi que d'implémenter le PEX et le DHT.

## 5.4.2) Souha TIBI

### 5.4.1.1) Ce que ce projet m'a appris

Le projet SEOC a été pour moi très enrichissant et formateur sur le niveau technique et sur le niveau personnel. En effet, cela m'a permis d'approfondir mes connaissances sur la communication au niveau sockets JAVA bloquantes et non bloquantes, j'ai pu découvrir les différents mécanismes pour envoyer les informations entre un client et un serveur, et comment mettre en place un protocole (codage, envoi et réception des messages) et comment faire des mesures de performance, stocker un fichier en disque et mettre en pratique les patterns et les bonnes pratiques de la programmation en Java.

Sur le plan personnel, le projet est une opportunité de plus pour collaborer et travailler en groupe puisqu'on apprend beaucoup des échanges que l'on a avec les autres membres de l'équipe mais aussi une occasion pour utiliser et maîtriser un peu plus les outils Scrum qui vont être fondamentaux pour travailler en entreprise. Cela m'a beaucoup apporté en "soft skills" également car j'ai dû améliorer mon expression orale afin de m'exprimer de façon plus claire ce qui est primordial pour garantir que la collaboration soit qualitative.

### 5.4.1.2) Ce qui a été le plus difficile

L'un des difficultés rencontrées était la compréhension de l'implémentation du protocole avec un seul client au début du sprint, c'était l'un des points où j'ai passé plus de temps à faire des tests et à relire la documentation jusqu'à la bonne assimilation des concepts de la communication avec des sockets. De plus, pour la partie multi-clients, les défis étaient d'utiliser les bonnes méthodes pour la lecture et pour l'écriture en travaillant en mode asynchrone, les tests étaient aussi difficiles à mettre en place pour tester plusieurs scénarios et vérifier les performances. La collaboration n'était pas également évidente tout le temps vu l'absence de séances où on est présent physiquement et cela demande plus d'effort et d'implication pour travailler.

### 5.4.1.3) Des choses à améliorer ?

Parmi les choses à améliorer dans le projet est l'implémentation de l'algorithme avec un coût optimisé, et aussi appliquer au mieux les principes du "clean code" pour factoriser le code et utiliser au mieux les fonctions implémentées. Les phases de tests et de validation peuvent être aussi améliorées en ajoutant plus de tests unitaires et des tests de système et en variant les scénarios de tests. Il vaut mieux aussi travailler quasi quotidiennement/ périodiquement au lieu de concentrer le travail sur des courtes périodes et afin de laisser plus de marge pour les tests et la résolution des bugs et l'amélioration de la qualité du code et l'ajout continu de la documentation et le logging.



### 5.4.3) Aiman Wahdi MOHD IMRAN

#### 5.4.3.1) Ce que ce projet m'a appris

Ce projet m'a permis de développer mes compétences dans le domaine réseaux. J'ai pu découvrir le protocole Bittorrent et comment mettre en place un client Bittorrent depuis scratch (sans code de départ). J'ai pu également mettre en pratique mes connaissances en programmation orientée objet (Java) et les approfondir surtout en terme d'architecture. Ce projet m'a donné l'opportunité d'aborder les concepts de programmations avancées comme design pattern et de découvrir le fonctionnement de socket TCP/IP en Java que je ne connaissais pas avant. J'ai pu également approfondir mes compétences dans l'utilisation avancée de git tels que des propres commits, l'utilisation de Issues pour répartir les tâches entre les membres de l'équipe, l'utilisation de branch etc. Ce projet m'a appris également les compétences en gestion de projet avec la méthode Scrum qui seront très importantes dans le monde du travail surtout qu'on travaille dans une équipe.

#### 5.4.3.2) Ce qui a été le plus difficile

C'était un peu difficile au début de comprendre le fonctionnement de sockets TCP/IP en Java mais avec l'aide de professeur et le livre sur les sockets TCP/IP avec des exemples pertinents, j'ai pu comprendre le fonctionnement de sockets et ensuite les mettre en pratique. Sachant que ce projet n'a pas de code de départ, j'ai trouvé que c'était assez difficile pour commencer à écrire des codes sans avoir totalement compris les concepts sur les protocoles bittorrent. Le confinement qui empêche la réunion physique entre chaque membre de l'équipe, me rend un peu difficile à collaborer avec les autres membres même si on communique avec Messenger et Discord, ce n'était pas très efficace à mon avis.

#### 5.4.3.3) Des choses à améliorer ?

Il reste quelques fonctionnalités qui manquent dans notre projet tels que plusieurs messages bittorrent au sein d'un segment TCP et la gestion concurrence Thread mais globalement ce projet est bien fait par rapport au cahier des charges. Les choses importantes à améliorer selon moi sont la communication entre les membres et la répartition des tâches qui est un peu difficile à mettre en place équitablement car cela dépend des compétences de chaque membre. Le fait de travailler chez soi ne nous permet pas de surveiller l'avancement des tâches de chaque membre et je pense que cela ira mieux si nous pouvons nous rencontrer pour une réunion physique au moins une fois par semaine en respectant les règles sanitaires.