TÉLÉCOM SudParis

Projet informatique 1ère année
PRO 3600

# MESSAGERIE INSTANTANÉE DÉCENTRALISÉE

BENOIT MARZELLEAU
CLÉMENT SAFON
GATIEN ROUJANSKI
SIFFREIN SIGY

**Encadrante responsable :** Julien ROMERO

Soutenance : 01/06/2022

*Chapitre 1*

# Introduction

Ce document décrit de façon succincte le fonctionnement d'un commencement de développement en Java d'un logiciel de messagerie instantanée décentralisée et en particulier de la table de hachage distribuée sous-jacente. Une table de hachage distribuée (DHT, *Distributed Hash Table*) permettant la mise en place d'une table de hachage dans un système répartis. Dans le cadre d'un système de messagerie instantanée décentralisée, la DHT permet de créer un réseau de recouvrement afin d'interconnecter à l'aide de liens logiques les instances de la DHT (ci-après appelés nœuds) composant la messagerie. Les tables de hachage distribuées sont des techniques efficaces (notamment le protocole Kademlia sur lequel nous nous sommes basés) pour rechercher n'importe quel nœud participant au réseau et résilientes dans la mesure ou la défaillance d'un ou plusieurs nœuds n'impacte pas la performance globale du système. Notre table de hachage distribuée, première pierre à l'édifice d'une messagerie instantanée décentralisée, permet de maintenir une liste de nœuds connus et en fonctionnement ainsi que de recherche n'importe quel nœud présent dans le réseau. Une interface graphique via une application web a aussi été créée afin d'illustrer le fonctionnement de la table de hachage distribuée.

*Chapitre 2*

# Cahier des charges

Le projet consiste à développer la table de hachage distribuée d'une application de messagerie instantanée décentralisée en Java.

Le logiciel final devra permettre :

- De maintenir une liste de nœuds à jour
- De rechercher des nœuds dans le réseau
- D'ajouter des nœuds à une instance locale de la table de hachage distribuée
- Le protocole de transport utilisé sera UDP
- La table de hachage distribuée devra être efficace

Les différentes étapes de son implémentation sont :

- Mise en place d'une chaîne de traitement pour l'intégration continue pour l'assurance qualité

- La gestion du réseau par la DHT (construction des paquets, envoi / réception de paquets UDP asynchrones, appels de callbacks)

- Gestion des pings par la table de hachage distribuée

- Gestion des nœuds connus, mise à jour de cette liste

- Gestion de la recherche de nœuds non connus

*Chapitre 3*

# Préliminaires

## 3.1 - Analyse du problème et spécification fonctionnelle

Les messageries instantanées les plus connues utilisent des serveurs pour acheminer les données des utilisateurs. Le but d'une messagerie instantanée décentralisée est de contourner l'utilisation de serveurs en utilisant un réseau décentralisé pair-à-pair. Les messages seront envoyés en se connectant directement au destinataire (donc sans passer par des serveurs), l'aspect pair-à-pair servant à déterminer l'adresse IP et le port des nœuds ainsi qu'à établir une route vers les destinataires en question. La table de hachage distribuée permet de réaliser cela.

## 3.2 - Conception préliminaire

Le logiciel sera constitué de la DHT et d'une interface web et permettra de tenir une liste de nœuds connus.

Les dépendances du projet sont gérées par Apache Maven (Maven), qui est un outil de gestion de projet de développement Java (gestion de la compilation, des tests, etc.).

La cryptographie (par courbes elliptiques) sera gérée par la librairie libsodium (une interface utilisant Java Native Access sera utilisée, libsodium-jna), qui permettra d'utiliser facilement les outils cryptographiques nécessaires.

La DHT est contenue dans un package Java. Ce dernier contient plusieurs packages, certains destinés à la gestion du réseau, d'autres à celle de la liste des nœuds connus ou encore à la gestion de l'asynchrone.

### 3.2.1 - Table de hachage distribuée

La DHT est un essaim auto-organisé de tous les nœuds du réseau. Ce module se charge de maintenir une liste de nœuds connus et actifs et de trouver n'importe quel nœud non-connu disponible. Chaque nœud de la DHT possède une clé publique. Cette clé publique fait office d'adresse. Cette adresse est temporaire et est renouvelée chaque fois que l'instance de tox est fermée ou redémarrée. Une fois que la clé publique DHT d'un nœud est connue, la DHT est utilisée pour le trouver et se connecter directement à lui via

UDP. La distance entre deux nœuds de la DHT est définie comme le XOR (OU EXCLUSIF) entre les deux clés publiques des nœuds, toutes deux traitées comme des nombres non signés de 32 octets au format grand-boutiste (octets de poids forts d'abord). Un nœud de la DHT avec la clé publique 1 sera plus proche d'un nœud avec la clé publique 0 que d'un nœud avec la clé publique 5 par exemple car : 1 XOR 0 = 1 et 1 XOR 5 = 4. Plus cette distance est petite, plus on dit que les nœuds sont proches. Puisque 1 est plus petit, cela signifie que 1 est plus proche de 0 que de 5. L'auto-organisation dans la DHT se produit par le biais de chaque nœud de la DHT qui se connecte à un nombre arbitraire de nœud les plus proches de leur propre clé publique et d'autres qui sont plus éloignés. Si chaque nœud dans le réseau connaît les nœuds avec la clé publique la plus proche de sa clé publique, alors pour trouver un nœud spécifique avec la clé publique X, un nœud a juste besoin de demander récursivement aux nœuds dans la DHT les nœuds connus qui ont les clés publiques les plus proches de X. Le nœud finira par trouver les nœuds dans la DHT qui sont les plus proches de lui et, si ce nœud est en ligne, il les trouvera.

### 3.2.2 - L'interface graphique

L'interface graphique permettra de visualiser le fonctionnement de ce protocole. Une représentation graphique de l'ensemble des utilisateurs et de leurs interactions sera également disponible.
Enfin, si l'aspect technique s'y prête, nous pourrons envisager une interface graphique qui correspond à une application de messagerie (comme Messenger).

*Chapitre 4*

# Conception détaillée

## 4.1 La table de hachage distribuée

Toutes les classes dans lesquels figure l'élément `log` sont enveloppées par le décorateur `@Slf4j` du package `org.projectlombok.lombok`, qui permet d'à partir d'une classe :

```
@Slf4j
public class Exemple {
}
```

de générer à la compilation :

```
public class Exemple {
    private static final org.slf4j.Logger log =
                        org.slf4j.LoggerFactory.getLogger(Exemple.class);
}
```

### 4.1.1 La cryptographie

La libraire libsodium est utilisée pour la cryptographie à clés publiques à base de courbes elliptiques (Ed25519). Ce type de cryptographie, en plus de permettre le chiffrement / déchiffrement, l'authentification d'un message chiffré. Ainsi, il est possible de s'assurer que le message reçu provient bien du nœud attendu. La librairie libsodium étant écrite en C, nous utilisons une bibliothèque Java, libsodium-jna, qui se lie aux APIs cryptographiques de libsodium avec *Java Native Access* (JNA).

Lors du démarrage de la table de hachage distribuée, la librairie C est d'abord chargée :

```
String libraryPath;
if (Platform.isWindows()) {
   libraryPath = "C:/libsodium/libsodium.dll";
} else if (Platform.isMac()) {
   libraryPath = "/usr/local/lib/libsodium.dylib";
} else {
   libraryPath = "/usr/lib64/libsodium.so.23";
}
SodiumLibrary.setLibraryPath(libraryPath);
```

Ensuite, un couple clé publique / clé privée est créé :

```
SodiumKeyPair keyPair = SodiumLibrary.cryptoBoxKeyPair();
```

```
this.publicKey = keyPair.getPublicKey();
this.privateKey = keyPair.getPrivateKey();
```

À présent, imaginons que Bob veuille envoyer un message chiffré à Alice. Pour ce faire, il utilisera cette méthode, où `receiverPublicKey` désigne la clé publique d'Alice, `nonce` est un nombre aléatoire de 24 octets et `this.privateKey` la clé privée de Bob, tous deux utilisés pour l'échange et servant à l'authentification du message :

```
public byte[] encrypt(byte[] receiverPublicKey,
                      byte[] nonce,
                      byte[] data) throws SodiumLibraryException {
      return SodiumLibrary.cryptoBoxEasy(data,
                                         nonce,
                                         receiverPublicKey,
                                         this.privateKey);
}
```

De la même manière, Alice, pour déchiffrer le message, utilisera la méthode suivante où `senderPublicKey` désigne ici la clé publique de Bob, `nonce` le nombre aléatoire de 24 octets utilisé par Bob pour signer son message, tous deux servant à vérifier la signature du message et `this.privateKey` la clé privée d'Alice :

```
public byte[] decrypt(byte[] senderPublicKey,
                      byte[] nonce,
                      byte[] data) throws SodiumLibraryException {
      return SodiumLibrary.cryptoBoxOpenEasy(data,
                                             nonce,
                                             senderPublicKey,
                                             this.privateKey);
      }
```

## 4.1.2 L'asynchronisme

L'ensemble des classes et interfaces permettant de gérer l'asynchronisme sont situées dans le package `pear.core.dht.async`.

La table de hachage distribuée doit être en mesure de gérer la réception de paquets sans que leur traitement n'engendre de blocage et qu'elle ne puisse plus les recevoir. Pour ce faire, ces classes permettent de découpler les entrées / sorties synchrones des asynchrones dans un système, de manière à simplifier la programmation concurrente sans pour autant dégrader les performances.

Les tâches devant être exécutées de manière asynchrone doivent implémenter l'interface suivant (qui étend l'interface `java.util.concurrent.Callable`) :

```
public interface AsyncTask<O> extends Callable<O> {
      void onPreCall();
```

```
    void onPostCall(O result);
    void onError(Throwable throwable);
    @Override
    O call() throws Exception;
}
```

La méthode `void onPreCall()` est appelée dans le contexte du fil d'exécution de l'appelant avant l'appel de `O call()`. De petites tâches peuvent être exécutées ici afin que la pénalité de performance du changement de contexte ne soit pas encourue en cas de demandes invalides.

La méthode `void onPostCall(O result)` est une méthode de rappel appelée après que le résultat a été déterminé avec succès par `O call()`. Cette méthode est appelée dans le contexte du fil d'exécution d'arrière-plan.

La méthode `void onError(Throwable throwable)` est une méthode de rappel appelée si le traitement de la tâche a entraîné une exception, que ce soit dû à `void onPreCall()` ou a `O call()`.

La méthode `O call()` est appelée dans le contexte du fil d'exécution d'arrière-plan, le traitement de la tâche résidant ici.

L'asynchronisme en tant que tel est géré par la classe `public class AsynchronousService`. Cette classe possède un attribut `private final ExecutorService service` qui représente la couche de mise en file d'attente ainsi que la couche synchrone. Le groupement de fils d'exécution contient des fils d'exécution qui exécutent les tâches de manière bloquante / synchrone. Les tâches longues sont exécutées en arrière-plan, ce qui n'affecte pas les performances du fil d'exécution principal. Le constructeur de la classe initialise cet attribut de façon à créer un service asynchrone utilisant `workQueue` comme un canal de communication entre les couches synchrones et asynchrones :

```
public AsynchronousService(BlockingQueue<Runnable> workQueue) {
    this.service = new ThreadPoolExecutor(10, // corePoolSize
                                          10, // maxPoolSize
                                          10, // keepAliveTime
                                          TimeUnit.SECONDS,
                                          workQueue);
}
```

Cette classe possède une méthode non bloquante qui exécute la tâche fournie en arrière-plan et termine immédiatement. Lorsque la tâche est terminée avec succès, le résultat est renvoyé à l'aide de la méthode de rappel `void onPostCall(O result)` de l'interface précédemment décrite. Si l'exécution de la tâche ne peut pas se terminer normalement en raison d'une exception, la raison de l'erreur est renvoyée à l'aide de la méthode de rappel `void onError(Throwable throwable)` vue précedemment :

```
public <T> void execute(final AsyncTask<T> task) {
    try {
        task.onPreCall();
    } catch (Exception e) {
        task.onError(e);
```

```
            return;
        }
        service.submit(new FutureTask<>(task) {
            @Override
            protected void done() {
                super.done();
                try {
                    // Appelée dans le context du fil
                    // d'exécution d'arrière-plan
                    task.onPostCall(this.get());
                } catch (InterruptedException ignored) {
                    // Ne devrait pas survenir
                } catch (ExecutionException e) {
                    task.onError(e.getCause());
                }
            }
        });
    }
```

Enfin, une méthode permet d'arrêter le groupement de fils d'exécution (c'est un appel bloquant pour attendre que toutes les tâches soient terminées qui n'est appelé qu'à l'arrêt de la table de hachage distribuée) :

```
public void close() {
    this.service.shutdown();
    try {
        service.awaitTermination(10, TimeUnit.SECONDS);
    } catch(InterruptedException e) {
        log.error("Error waiting for executor service shutdown!");
    }
}
```

### 4.1.3 Le réseau

Toutes les classes liées au réseau sont dans le package `pear.core.dht.network`.

Bilan des objets :

- `Network` :
    - Contient la méthode `public void sendPacket(Packet packet, Node receiver, Callable<?> callback)` qui permet d'envoyer un paquet et enregistre la méthode à rappeler (`callback`) quand la réponse sera reçue s'il s'agit d'une requête
    - Contient la méthode `public void handle()` qui lance une tâche en arrière-plan (via `PacketManagementTask` et `AsynchronousService`) chargée de gérer le paquet reçu

    - Contient la méthode public void close() qui arrête le service d'exécution asynchrone et ferme le connecteur UDP
- `Node` : modélise un nœud
    - Contient la méthode `public boolean isAlive()` qui indique si le nœud modélisé est actif ou non en lui envoyant un ping
- `Packet` : modélise un paquet standard
    - Possède un constructeur qui décode un paquet brut
    - Contient une méthode `public byte[] toByteArray()` qui renvoie le paquet sous forme brute, prêt à être envoyé
- `PacketManagementTask` : implémente l'interface `AsyncTask<Callable<?>>` et gère l'action à réaliser selon le paquet reçu
    - La méthode de rappel est appelée, dans le cas d'une réponse, par la méthode `public void onPostCall(Callable<?> result)`
    - La méthode `public Callable<?> call` détermine l'action à réaliser selon le paquet reçu
- `PacketType` : énumération des types disponibles (`REQUEST`, `RESPONSE`)
- `Ping` : modélise un ping
    - Contient la méthode `public boolean isPending()` qui indique si la réponse au ping à été reçue
    - Contient la méthode `public void send(PacketType type)` qui envoie un paquet encapsulant dans `Packet` les données du ping et en utilisant la classe `Network` pour l'envoie et l'enregistrement de la méthode de rappel (lui-même)
    - Contient la méthode de rappel `public Integer call()` qui se charge de renseigner dans la liste des nœuds connus la date de réception du (dernier) ping
- `RPCService` : énumération des procédures distantes disponibles (`PING`, `FIND_NODE`)

Tous les données envoyés par la table de hachage distribuée sont encapsulées dans un paquet standard de la forme :

| | |
|---|---|
| Section non chiffrée | Nature du paquet (1 octet) |
| | Clé publique de l'expéditeur (32 octets) |
| | Nonce (24 octets) |
| Section chiffrée | Nature du paquet (1 octet) |
| | Identifiant de l'échange (8 octets) |
| | Charge utile (taille variable) |

Les principales natures de paquet sont les requêtes / réponses ping et de recherche de nœud. La requête et la réponse étant chiffrées par la même clé, la nature du paquet est dupliquée dans la section chiffrée pour empêcher qu'il soit possible de créer une réponse à un ping sans avoir besoin de déchiffrer la section chiffrée. Un identifiant d'échange est utilisé afin de s'assurer que la réponse reçue correspond bien à la requête précédemment effectuée. La charge utile contient les informations utiles au fonctionnement de la table de

hachage distribuée, en particulier les nœuds demandés lors d'une recherche de nœud (elle est vide pour un ping). La classe `Packet` permet de modéliser un tel paquet. Cette dernière est utilisée dès qu'un paquet est reçu et dès qu'il faut en envoyer un pour encapsuler les données.

La classe `Node` modélise un nœud, et permet notamment de tester si un nœud est actif ou non, en lui envoyant un ping. Pour ce faire, la méthode `public boolean isAlive()`. Elle instancie la classe `Ping` (qui modélise un ping) et appel la méthode `void send (PacketType type)` en spécifiant le type du paquet (requête / réponse, ici requête), ce qui déclenche l'appel de la méthode `public void sendPacket(Packet packet, Node receiver, Callable<?> callback)` de la classe `Network` qui réalise l'envoie effectif du paquet et enregistre la méthode de retour associée à l'identifiant de l'échange. Un délai d'attente maximal est défini par une constante définie dans la classe `Network`, et l'attente est réalisée par la méthod `public boolean isAlive()` de la classe `Node` de la manière suivante :

```
while (ping.isPending() && ((new Date()).getTime() -
ping.getSentDate().getTime()) < Network.PING_TIMEOUT) {
    try {
        synchronized (this) {
            this.wait(100); // Attente de 100 ms
        }
    } catch (InterruptedException e) {
        // Affiche d'un message d'erreur
}
```

Lorsque le délai d'attente est dépassé, la méthode de rappel précédemment enregistrée est supprimée et le nœud est considéré comme inactif.

La méthode `public void handle()` de la classe `Network` qui reçoit les paquets est implémentée de la manière suivante :

```
public void handle() throws SocketException {
    this.pingSocket = new DatagramSocket(DHT_PORT); // DHT_PORT = 33445
    var service = new AsynchronousService(new LinkedBlockingDeque<>());
    this.running = true; // Mis à false à l'extinction
    while (running) {
        DatagramPacket receivedPacket = new DatagramPacket(
                new byte[MAX_UDP_PACKET_SIZE],
                MAX_UDP_PACKET_SIZE // 65536
        );
        try {
            // En attente de paquets
            this.pingSocket.receive(receivedPacket);
            service.execute(new PacketManagementTask(
                    this.dht,
                    Arrays.copyOfRange(receivedPacket.getData(),
                            0,
                            receivedPacket.getLength()
```

```
                    ),
                    receivedPacket.getAddress()
            ));
        } catch (IOException e) {
            log.warn(e.getMessage());
        }
    }
    service.close();
}
```

Les méthodes de rappels sont enregistrés à l'aide d'une table de hachage placée en attribut de la classe `Network` :

```
private final ConcurrentHashMap <ByteBuffer, Callable<?>>
        trackingSentPacket = new ConcurrentHashMap<>();
```

On notera deux choses :

- La classe `ByteBuffer` est utilisée au lieu du type primitif byte[] car ce dernier n'implémente ni la méthode `boolean equals(Object o)` ni `int hashCode()` (car c'est un type primitif et non une classe !) et ne peut donc être utilisée dans une table de hachage sous peine que la méthode `Callable<?> get(ByteBuffer key)` renvoie systématiquement `null`.
- Utiliser la classe `ConcurrentHashMap` au lieu de `HashMap` permet de se prémunir de tous les effets indésirables liés à l'exécution de plusieurs fils d'exécution accédant et modifiant la même structure de donnée de manière concurrente.

# 4.2 L'interface graphique

Après avoir étudié la documentation du protocole de la table de hachage distribuée (Kademlia), nous nous sommes tournés vers son implémentation pour créer une application fonctionnelle. Cependant, comme nous ne pouvions pas tous travailler en même temps sur cette partie du projet, nous avons jugé utile de faire une interface graphique afin de pouvoir tout d'abord visualiser les différents utilisateurs dans le réseau..

Dans cette partie, nous faisons donc une interface graphique pour pouvoir visualiser la manière dont l'application fonctionne. Pear permet de connecter directement deux utilisateurs en parcourant le réseau formé par l'ensemble de tous les utilisateurs. Les utilisateurs possèdent une table de hachage qui permet de référencer certains nœuds connus. Un utilisateur peut alors se connecter (envoyer des paquets) à n'importe quel nœud du réseau en passant par un ou plusieurs utilisateurs intermédiaires, afin d'obtenir son adresse IP et son port pour communiquer.

## 4.2.1 Détail et utilisation de l'application web

Dans cette représentation graphique, une première page sert à lister tous les nœuds du réseau dans un tableau. On peut aussi connaître leurs nombres totaux de connexions et leurs différents utilisateurs présents dans leur propre table de hachage.

De plus une représentation graphique permet de visualiser ces connexions avec des liens physiques pondérés par leur poids (qui correspond au nombre total de connexions qu'ils possèdent). Cela crée un graphique avec des forces de type gravité » qui permet d'observer des points centraux (plus riches en informations) du réseau.

Pour naviguer sur la page web et passer d'une visualisation à une autre, on peut utiliser le menu de gauche. Un bouton permet aussi d'alterner entre les visualisations 3D et 2D pour les graphiques. Les graphiques se manipulent intuitivement en déplaçant les nœuds.

## 4.2.2 Données d'entrée

L'application graphique prend en entrée un exemple de réseau et d'utilisateurs. Nous avions initialement prévu de faire une API pour que la page web puisse communiquer avec le programme en java et permettre une mise à jour automatique. Malheureusement, par manque de temps, nous n'avons pas pu.

Les données sont au format JSON. Il rassemble les nœuds avec leur id, nom et une valeur "val" qui représente leur poid. De cette manière on liste tous les nœuds et toutes les connexions. La valeur 'val' qui représente leur poid va permettre d'attribuer un poids physique au point du graphique.

Exemple :

```
var myData = {
    "nodes": [
        {
          "id": "192.168.1.1",
          "name": "192.168.1.1:33445",
          "val": 0
        },
        {
          "id": "192.168.1.2",
          "name": "192.168.1.2:33445",
          "val": 0
        },
```

```
"links": [
    {
        "source": "192.168.1.1",
        "target": "192.168.1.2"
    },
    {
      "source": "192.168.1.1",
      "target": "192.168.1.3"
    },
```

Par la suite on va calculer les valeurs stockés dans 'val' en fonction du nombre d'apparition de chaque 'id' dans 'links'. De ce fait, les nœuds auront une taille proportionnelle à leur nombre de connexions.

## 4.2.3 Programmation

Cette interface graphique est donc une interface web, elle utilise les langages html, Javascript et css. De plus, nous nous servons de deux librairies JS pour générer les graphiques.
Dans l'arborescence on retrouve :

- la page programmé en html
- un script css principal (codé en scss puis compilé en css)
- un script css avec des designs pré-faits
- un script Javascript qui permet de faire fonctionner les boutons sur la page
- un script Javascript qui charge les graphiques et le tableau
- deux scripts Javascript qui sont les modules importés et qui permettent de générer les graphiques (2D et 3D).

Le scripte qui charge les élément graphiques permet aussi de compter le nombre d'apparition d'un certain 'id' et de mettre cette valeur dans le champ 'val' afin que son poids soit ajusté.

Le tableau est créé en itérant sur la liste de nœud et en ajoutant des enfants ("Child") aux éléments de la page html.

```
//On récupère l'ip
var row_ip = document.createElement("td");
row_ip.appendChild(document.createTextNode(node["id"]));
new_line.appendChild(row_ip);
```

De cette façon on créer des colonnes et des lignes.

## 4.2.3 Graphiques

La tableau :

| Nodes | | | |
|---|---|---|---|
| IP | Port | Connexion(s) | Target(s) |
| 192.168.1.9 | 33445 | 6 | 192.168.1.2 |
| 192.168.1.8 | 33445 | 6 | 192.168.1.5 |
| 192.168.1.7 | 33445 | 8 | 192.168.1.8 |
| 192.168.1.6 | 33445 | 10 | 192.168.1.7<br>192.168.1.2<br>192.168.1.1 |
| 192.168.1.5 | 33445 | 8 | 192.168.1.8<br>192.168.1.7 |
| 192.168.1.4 | 33445 | 8 | 192.168.1.2<br>192.168.1.2<br>192.168.1.2<br>192.168.1.9 |
| 192.168.1.3 | 2230 | 6 | 192.168.1.6<br>192.168.1.5 |
| 192.168.1.2 | 33445 | 16 | 192.168.1.9<br>192.168.1.7 |
| 192.168.1.1 | 33445 | 8 | 192.168.1.6<br>192.168.1.3<br>192.168.1.2 |

Le graphique 2D :

*Chaque point représente un nœud, c'est-à-dire, un utilisateur.*

*Chaque lien représente le fait que l'utilisateur ait la cible dans sa table de hachage.*

*Chapitre 5*

# Résultats des tests de validations

Composante essentielle des projets de développement informatique, l'intégration continue permet de détecter les nouveaux bugs lors de la modification de code et d'assurer un certain niveau de fonctionnement en ce sens grâce à l'exécution d'une série de tests.

## 5.1 - L'intégration continue avec Maven

Avant la mise en place d'intégration continue, il est nécessaire de comprendre la place qu'elle occupera dans le projet. Tel qu'expliqué en partie 3, celui-ci utilise Maven comme utilitaire de gestion de dépendances et d'automatisation. Grâce au système de plugins de Maven, il suffit donc d'ajouter les outils requis dans la configuration POM (*Project Object Model*, fichier contenant des informations sur le projet utilisé par Maven notamment pour lister les modules sur lesquels le logiciel dépend) du projet.

## 5.2 - Constitution de la Pipeline

Une "pipeline d'intégration" décrit un ensemble de processus exécutés lors de la modification du code. Nous avons fait le choix d'inclure trois étapes à notre intégration continue :

### Tests unitaires

Ils constituent le cœur de l'intérêt de l'intégration continue. Inclus dans le code du projet, ils permettent de tester une fonctionnalité ou un comportement précis du programme. Par exemple, ils permettent de tester une méthode avec différents paramètres pour observer le bon fonctionnement de celle-ci.
Les tests sont encadrés par *JUnit 5*, un framework de test pour *Java* qui fournit de nombreuses fonctions utiles au développement des tests. Ceux-ci sont exécutés automatiquement à la publication de code et retournent une erreur en cas d'échec.
L'avantage majeur est la capacité de cibler rapidement la méthode responsable et de comprendre la raison de l'échec.

### Couverture de code

Complémentaire aux tests unitaires, les rapports de couverture de code permettent de suivre les parties du projet qui sont couvertes ou non par les tests. A l'issue du rapport, un taux est mesuré en pourcentage de code couvert par les tests. L'objectif est alors d'approcher au mieux les 100% avec ce taux. En pratique, une bonne valeur est supérieure à 75%.
Nous avons utilisé ici *Jacoco* pour sa facilité d'intégration avec *Maven* et sa configuration.

### Contrôle de code

Il est clair que le niveau et les habitudes de chacun diffèrent, peu importe le projet ou le langage. Il est donc intéressant d'imposer des règles et un formatage particulier pour le code afin d'uniformiser l'ensemble du projet.
Grâce au plugin *Checkstyle*, un second rapport est généré à chaque exécution de la pipeline détaillant les écarts par rapport aux règles choisies. Certains environnements de développement tels que *Visual Studio Code* proposent même un formatage automatique avant publication des modifications pour corriger un maximum d'erreurs.
En cas d'écarts trop importants, il est même possible de retourner une erreur et donc d'invalider les modifications de code.

## 5.3 - Intégration avec GitHub Actions

Bien qu'utilisable uniquement avec Maven sur l'ordinateur de chaque développeur, il est bien plus intéressant que la pipeline d'intégration soit exécutée automatiquement à chaque publication de code. Nous utilisons déjà GitHub pour héberger notre répertoire Git, nous pouvons donc exploiter GitHub Actions pour atteindre cet objectif.
Ainsi, à chaque commit, deux processus s'exécutent en parallèle : les tests unitaires et le contrôle de code. A l'issue de ceux-ci, un rapport Jacoco et un rapport Checkstyle sont publiés et associés au commit testé. De plus, une version compilée du programme est téléchargeable et peut être testé manuellement si souhaité.
Enfin, en cas d'échec, GitHub Actions avertit par mail les développeurs et le commit est identifié en temps que tel.

## 5.4 - Résultats

A la lumière du rapport Jacoco du dernier commit à l'écriture de ces lignes, la couverture de code par les tests unitaires n'est pas suffisamment élevée pour atteindre un niveau satisfaisant : 15% de 2 513 lignes de codes sont couvertes par les tests.
Cela s'explique tout d'abord par le temps écoulé entre le début du projet et les premiers tests unitaires écrits dû à un apprentissage du fonctionnement de Maven et JUnit : le retard s'est alors accumulé et n'a pas pu être rattrapé. De plus, un accent particulier a été mis sur la table de hachage distribuée et de sa classe principale qui présente quant à elle une couverture deux fois plus importante.

On notera également, grâce aux statistiques mises à disposition par GitHub, que 30 pipelines ont échoué sur les 153 exécutées au total dont une seule concernant le contrôle de qualité du code.

Enfin, il est intéressant de remarquer que les tests unitaires nous ont permis de s'assurer que le changement de version de Java en milieu de projet ne l'impacte pas. En effet, la version JDK 18 a été publiée au cours du développement et une modification de la configuration de Maven nous a permis de basculer sur cette nouvelle version tout en vérifiant qu'aucun bug n'apparaîtra en cours de route. Cependant, la faible couverture de code n'a pu garantir un résultat totalement fiable et il a été nécessaire d'appuyer celui-ci manuellement.

*Chapitre 6*

# Bilan

## 6.1 - Histoire de notre projet

Le projet pear est un projet destiné à implémenter le protocole Tox en Java, qui est un protocole de communication en peer-to-peer de messagerie. Nous avions donc comme projet d'implémenter ce protocole en Java et de potentiellement réaliser une application de messagerie.
Nous avons commencé par étudier toutes les documentations disponibles sur ce protocole et particulièrement son implémentation en C à laquelle nous avions accès. Puis nous avons aussi dû se répartir le travail pour pouvoir se répartir les tâches et avancer en même temps.

Le point central du projet est de programmer une table de hachage distribuée qui permet d'établir le réseau d'accès entre tous les utilisateurs. L'implémentation de cette table de hachage n'as pas été simple car sa conception implique une reformulation de ce que l'on avait compris et extrait des différentes sources. De plus, il a fallu faire une petite interface graphique afin de pouvoir illustrer et présenter le projet.
Initialement nous n'avions pas prévu d'interface graphique car nous savions que l'application de messagerie était un très gros projet qui allait nous prendre toute l'année. Cependant étant quatre dans le groupe, nous ne pouvions pas travailler sur exactement la même partie, et de plus, une interface simplement de visualisation du réseau et pour visualiser les utilisateurs sur le réseau s'avérait intéressant. Nous avons alors décidé de faire une petite interface graphique qui pourrait illustrer en temps réel le réseau à l'aide de graphiques.

# 6.2 - Difficultés rencontrés et Solutions

La première des difficultés concernant notre projet est la table de hachage. En effet, l'étape ultime était de faire une application de messagerie qui utilise le protocole Tox. Cependant dès le début du projet nous savions que l'application était très ambitieuse car il fallait déjà réussir à implémenter le protocole en Java, en partant des seules documentations que nous avons pu recueillir. La première grosse étape à alors été de réaliser cette table de Hachage distribuée. Cependant, nous nous sommes fait surprendre par la difficulté de créer une vraie table de hachage distribuée et par le temps que cela allait prendre. De ce fait, nous n'avons pas pu faire le lien (utilisation d'une API) entre le programme en Java et l'interface graphique en langage Web.

## 6.2.1 Interface Graphique

### 6.2.1.1 Problèmes rencontrés

Lors de la recherche des librairies pour former les graphiques en JavaScript nous avons mis du temps à réussir à implémenter la librairie qui générait les graphiques en 2D. Il a fallu se documenter sur les import de module en Javascript. Il a fallu faire des importations de librairie en différé (car ce sont des librairies volumineuses qui se chargeaient après l'exécution de notre propre script). De plus, il a fallu chercher une version du script de la librairie Javascript qui fonctionnait (la dernière version disponible étant visiblement buggé).

### 6.2.1.2 Améliorations possibles

Si nous avions eu plus de temps pour développer l'interface graphique, nous aurions pu implémenter une API qui permet de visualiser le réseau en temps réel. Nous aurions également aimé faire une véritable application de messagerie.

Dans cette partie, des améliorations étaient possibles en accord avec l'avancement de la partie « technique » du projet. Cela a été contraignant et n'a pas pu permettre la finalisation de cette partie. Le développement de l'API aurait été un plus pour expliquer comment fonctionne le réseau mais il aurait fallu un travail non négligeable pour faire une

application de messagerie fonctionnelle, au niveau interface graphique, mais surtout au niveau technique.

# 6.3 - Manuel utilisateur

## 6.3.2 L'interface Graphique

Cette page web possède deux pages :

- La première affiche un tableau de noeuds
- La seconde affiche les graphiques

Pour passer d'une page à une autre on peut se servir de la barre latérale de navigation.

Le graphique peut passer d'une représentation 2D à 3D en cochant le bouton en haut à gauche.

Enfin les graphiques sont interactif : on peut passer sa souris sur les points, déplacer des points, zoomer…

# 6.4 - Conclusion

Nous pouvons dire pour conclure que ce projet à été ambitieux et que nous avons tout de même pu réaliser le plus gros du travail en réalisant les fondations qui sont l'implémentation de protocol Tox en Java. Si nous devions continuer ce projet nous pourrions développer la messagerie entièrement. L'avantage d'avoir une application en Java et que nous aurions pu assez facilement en faire une application Android.