Rapport UGE Greed

Université de Gustave Eiffel - M1 informatique

2022-2023

Matière

Programmation Réseau

- Ramaroson Rakotomihamina Johan
- Tellier Quentin

Table des matières

I. Introduction

Ce document est un document technique contenant l'architecture principale du projet ainsi que nos divers choix d'implémentation pour la réalisation de celui-ci. Celui-ci contiendra également les améliorations apportées entre la première soutenance bêta et le rendu final.

Ce projet a été effectué dans le contexte de l'université de Gustave Eiffel pour notre année de Master 1 informatique. Il a pour but d'appliquer les bases de la programmation réseau dans le langage de programmation Java, et plus particulièrement le protocole TCP. La réalisation de ce projet se base autour de l'implémentation d'un protocole réseau nommé "UGE_GREED" dont la RFC se trouve dans le fichier rfc_9384.txt.

UGE_GREED est un protocole visant à implémenter un réseau d'applications connectées entre elles permettant à chacun de demander d'effectuer une conjecture sur une plage de valeur, le réseau d'application permet le partage équitable pour une optimisation visant à accepter une plage de calcul conséquente.

Vous pouvez suivre le fichier README.md pour y trouver les détails de compilation et exécution du programme.

II. Fonctionnalités du programme

Nous attestons de la fonctionnalité de chaque feature demandé :

- Mise en place d'un réseau d'application qui communiquent entre eux avec une table de routage fonctionnelle et mise à jour en temps réel
- Algorithme de distribution équitable lors du lancement d'un calcul et prise en compte de leur potentielle de calcul lors de la demande.
- Lecture des commandes suivantes dans le terminal : INFO, START, SHUTDOWN
- Permettre à une application de demander une conjecture, de la traiter et de renvoyer la réponse
- Lecture et envoi de chaque frame défini dans la RFC
- Récupération et implémentation du Jar en http non bloquant
- Gestion d'une déconnexion propre pour chaque application du réseau pour chaque situation

III. Choix d'implémentation

III. 1 - Les frames et leur component

Notre organisation se base sur les Frame à envoyer ainsi que leur Component.
Une Frame en Java est une sealed interface représentant le contenu d'une trame à envoyer dans un ByteBuffer, une frame doit être un record JAVA pour pouvoir être lue correctement.
La Frame permet uniquement à 3 sous-interfaces d'être implémentés conformément à notre RFC:

- LocalFrame
- TransferFrame
- BroadcastFrame

Une Frame va donc implémenter l'une de ses trois interfaces pour définir son type de Frame.
Un GreedComponent est une interface Java permettant d'implémenter le contenue d'une Frame et contient une méthode permettant d'insérer ses données dans un byteBuffer ainsi qu'une autre permettant de récupérer sa taille en bytes nécessaire pour être insérer dans un byteBuffer.
Chaque Frame est et doit toujours être composé d'un GreedComponent. Si un développeur souhaite créer un autre component, il doit impérativement implémenter cette interface et remplir correctement les méthodes putInBuffer et size.
Le programme renvoie une exception si une Frame a été créée en contenant une classe qui n'implémente pas cette interface.

Nous avons ici fait le choix de sceller l'interface Frame pour être en accord avec notre RFC, qui spécifie qu'il n'existe que 3 types de paquets à savoir le TransferPacket, BroadcastPacket et LocalPacket. Avec cette implémentation, nous utiliserons le pattern-matching pour pouvoir factoriser le comportement de chacun de ces trois types de frames.

III. 2 - Le pattern Visitor

Pour les actions à effectuer lors de la réception d'une Frame, nous avons décidé d'utiliser le design Pattern Visitor pour deux raisons principaux :

- Permettre de séparer les actions que le serveur doit effectuer lors de la réception d'une certaine Frame de l'objet même représentant le serveur.
- Séparer deux cas de réception d'une frame: cas où on reçoit une frame depuis un enfant, et l'autre depuis un parent, avec un code factorisé au mieux.

A noter que nous sommes conscient que depuis java 17 il est possible d'effectuer le Pattern Matching sur les sealed interfaces et donc d'éviter d'utiliser le pattern, mais l'avons fait en connaissance de cause.

III. 3 - Les readers

Un Reader est une interface JAVA permettant de lire des données dans un ByteBuffer. Nous avons implémenté un reader pour lire chaque type de données qui peut être contenue dans une Frame. Avec ces readers, nous avons conçu notre propre usine à reader qui utilise la réflection pour parser les composants d'un record et créer le reader qui lui est associé de manière automatisée. Ainsi, pour chaque Frame, lui est immédiatement associé son reader. Grâce à cette implémentation, la création d'une Frame s'effectue en créant un record contenant le contenue de son body, implémentant le bon type de Frame, il faudra également mettre à jour le switch dans la classe du 'Visitor' en ajoutant un case pour pouvoir appeler la méthode 'visit' sur cette Frame, ainsi qu'à spécifier l'opCode dans l'énumération ainsi que l'objet à qui cet opCode est associé.

Pour résumer, voici un exemple d'implémentation d'une nouvelle frame contenant

- Une IDComponent qui encapsule une InetSocketAddress
- Une StringComponent qui encapsule une simple String
- un IntegerComponent qui encapsule un simple int

Et étant une frame local :

```
public record ExampleFrame(IDComponent src, StringComponent message,
IntegerComponent number) implements LocalPacket{
}
```

Il faut également ajouter sa méthode visit dans l'interface FrameVisitor.

```
default void visit(Frame packet) {
   switch(packet) {
        case ConnectFrame p -> visit(p);
        case ConnectOKFrame p -> visit(p);
        case ConnectKOFrame p -> visit(p);
        case AddNodeFrame p -> visit(p);
        case WorkRequestFrame p -> visit(p);
        case WorkAssignmentFrame p -> visit(p);
        case WorkResponseFrame p -> visit(p);
        case WorkRequestResponseFrame p -> visit(p);
        case LogoutRequestFrame p -> visit(p);
        case LogoutDeniedFrame p -> visit(p);
        case LogoutGrantedFrame p -> visit(p);
        case PleaseReconnectFrame p -> visit(p);
        case ReconnectFrame p -> visit(p);
        case DisconnectedFrame p -> visit(p);
        case ExampleFrame p -> visit(p); //A ajouter ici
   }
}
```

Et pour finir l'ajouter dans l'énumération Opcode en spécifiant le byte qui lui est associé ainsi que sa frame :

```
public enum OpCode {
   EXAMPLE((byte)0x09, ExampleFrame.class),
```

III. 4 - La déconnexion

On peut deconnecter une application qui n'est pas ROOT en tapant SHUTDOWN dans le terminal, l'application se déconnecte si elle n'effectue aucun calcul et que la mère accepte sa déconnexion, toutes les filles de l'application qui cherche à se déconnecter vont donc pouvoir se connecter à leur nouvelle mère et tout le réseau va mettre à jour sa table de routage.

Si l'application ne peut pas se déconnecter toute de suite par exemple parce qu'elle est en train de faire des calculs alors elle n'acceptera plus aucun calcul, aucune nouvelle connexion et aucune nouvelle déconnexion, dès qu'elle aura fini ses calculs elle demandera à sa mère une nouvelle tentative de déconnexion.

On peut aussi déconnecter une application ROOT si elle ne possède aucun voisin.

IV. Difficultés rencontrées

Le refactoring du code a été la partie la plus longue et difficile de ce projet. Nous voulions avoir un équilibre entre la lisibilité du code et la propreté dans le sens "scolaire" de celui-ci. Que ce soit en termes de factorisation de lignes, mais également de factorisation de traitement, comme séparer le contexte en deux.

En terme d'implémentation, l'ajout de la possibilité de récupérer un jar depuis internet en non-bloquant nous a été la plus compliquée, il a fallu lui concevoir un système de reader et de context totalement dissocié du contexte du projet en lui-même.

Avec tout ceci, le plus dur n'a pa été l'implémentation des features, mais plutôt de permettre à un développeur extérieur de comprendre le code en lui-même mais également de le réutiliser et/ou implémenter lui-même ses Frame et Components depuis notre code pour lui accorder la possibilité d'améliorer le programme tout en restant flexible et cohérent avec la base. Cet état d'esprit a permis de nous obliger à coder de manière compréhensible et efficace.

Le gros point faible de notre programme est son manque cruel de test unitaire, surtout côté réseaux. Nous en avons tout de même fait quelques-uns pour s'assurer du bon fonctionnement de l'usine à reader, ou de l'implémentation du jar en http.

V. Prise en compte des critiques

Depuis la première soutenance, notre projet à beaucoup évolué, nous avons eu le temps d'approfondir tant les aspects de factorisation de notre code que l'implémentation de nouvelles features conformément à ce qui nous a été demandé.

V. 1 - Rendre la récupération thread-safe et optimiser le temps de calcul

Pour commencer, nous avons non seulement résolu le problème de la réception des paquets, ainsi que rendu le traitement thread-safe, mais également amélioré la rapidité du traitement des calculs en se basant sur le "pre-working" vu en tp.

Pour le traitement des paquets thread-safe on utilise une LinkedBlockingQueue de paquets qu'on va remplir avec les informations nécessaires pour envoyer les paquets, puis le thread principal va les récupérer pour les envoyer donc on a seulement le thread principal qui s'occupe de l'envoi des paquets.

Pour le calcul on utilise des thread pré-démarrés, quand on reçoit une demande de calcul on remplit la LinkedBlockingQueue avec les informations pour que nos threads puissent faire les calculs, puis ils vont mettre la réponse dans une autre LinkedBlockingQueue, et on a un thread qui s'occupe de récupérer les réponses et de placer les paquets à envoyer dans une autre LinkedBlockingQueue.

Selon les résultats du calcul on aura des codes différents, 0 si tout se passe bien, 1 si le Checker lève une exception et 3 si on n'arrive pas à récupérer le checker.

Si on obtient un code autre que 0, dans le fichier des réponses on met le problème qu'on à rencontrer selon le code.

Les threads qui font les calculs:

```
Thread.ofPlatform().daemon().start(() -> {
    for(;;) {
        try {
            var computation = task.take();

            if(computation.checker() == null) {
                responses.put(new ResponseTaskComputation(computation.packet(), computation.id(),
                        computation.value(),  response: "Value: " + computation.value() + " -> " +
                        "Cannot get the checker\n", (byte) 0x03));
            }
            else {
                try {
                    var response = computation.checker().check(computation.value());

                    responses.put(new ResponseTaskComputation(computation.packet(),
                            computation.id(), computation.value(),  response: "Value: " +
                            computation.value() + " -> " + response + "\n", (byte) 0x00));
                } catch(Exception e) {
                    responses.put(new ResponseTaskComputation(computation.packet(),
                            computation.id(), computation.value(),  response: "Value: " +
                            computation.value() + " -> " + "Exception raised\n", (byte) 0x01));
                }
            }
        } catch (InterruptedException e) {
            // Ignore exception
        }
    }
});
```

Le thread qui récupère les réponses pour construire les paquets:

```
Thread.ofPlatform().daemon().start(() -> {
    for(;;) {
        try {
            var response = computation.takeResponse();

            if(response.packet().src() != null) {
                transfer(response.packet().src().getSocket(), new WorkResponseFrame(
                        response.packet().dst(),
                        response.packet().src(),
                        response.packet().requestId(),
                        new ResponseComponent(response.value(), response.response(), response.code())
                ));

                incrementComputation(response.id());
            }
            else {
                var id = new ComputationIdentifier(response.packet().requestId().get(), getAddress());

                treatComputationResult(id, response.response());
            }

            if(isShutdown() && !isComputing()) {
                sendLogout();
            }

            selector.wakeup();
        } catch (InterruptedException e) {
            return;
        } catch (IOException e) {
            // Ignore exception
```

V. 2 - Récupération d'un Jar via HTTP en non-bloquant

Nous avons également réussi à implémenter la récupération du jar en Non bloquant, en se basant sur un système classique de [client non bloquant - context - reader] tout en adaptant les méthodes de lecture pour qu'il puisse correspondre à la RFC Http. Cette feature rend notre projet actuel totalement non bloquant. Cette partie de code est disponible dans le package util.http, à noter qu'elle ne figure pas dans les packages core du projet car le développement à été réalisé de telle sorte à ce que cette partie soit une API distinct du projet de base.

En terme d'implémentation le principe reste simple, nous avons une méthode principale qui lis le contenue d'un buffer et renvoi le header qui lui est associé en tant que String.

HTTPHeaderReader.java

```
String read(ByteBuffer buffer) {
   byte b = 0;
   buffer.flip();
   var done = false;
   while(buffer.hasRemaining()){
       b = buffer.get();
       builder.append((char) b);
       if (builder.length() > 3 && builder.substring(builder.length() -
4).equals("\r\n\r\n")) {
           done = true;
           break;
       }
   }
   if(!done){ //This case occurs when the buffer has no more content but the header
is not complete
       return null;
   }
   buffer.compact();
   return builder.substring(0, builder.length() - 4);
}
```

A l'extérieur on gère le cas où la fonction renvoi null, il faut lancer un Refill.
Après récupération du header, on effectue le pattern récurrent de la lecture d'un ByteBuffer en non-bloquant avec les readers. Lors de la lecture du Body on prépare un byteBuffer de la longueur du content-length et on insère le contenu dans un tableau de bytes, c'est ce que notre Reader renvoi. Voici un aperçu de l'action à effectuer lors de la lecture du body :

HTTPReader.java on method process(ByteBuffer)

```
if(state == State.WAITING_BODY){
   var header = headerReader.get();
   if(header.getCode() != 200){
       state = State.ERROR;
       return ProcessStatus.ERROR;
   }
   try {
       var contentLength = header.getContentLength();
```

```
        if(contentLength == 0){
            state = State.DONE;
            return ProcessStatus.DONE;
        }
        Buffers.fillBuffer(buffer, bodyBuffer);
        if(bodyBuffer.hasRemaining()){
            return ProcessStatus.REFILL;
        }
        body = new byte[contentLength];
        bodyBuffer.flip();
        bodyBuffer.get(body);
        bodyBuffer.compact();
        state = State.DONE;
        buffer.compact();
    }catch (HTTPException e){
        state = State.ERROR;
        return ProcessStatus.ERROR;
    }
}
```

Après réception du tableau de byte contenant le jar, le context s'occupera de générer le fichier ainsi que de couper la connexion entre le client et le serveur distant.
A noter pour la réutilisation que notre client possède un champ 'onDone' qui est un consumer à appliquer sur le jar récupéré. Elle s'exécutera lorsque le client se sera assuré que le jar a bien été récupéré.

V. 3 - Automatisation de l'encodage et du décodage d'une frame

Notre système de reader à entièrement été revu et factorisé de sorte à ce que la création d'une nouvelle Frame s'effectue en exactement 3 lignes de code avec son reader automatisé. Il faut néanmoins se référer à la partie suivante ainsi qu'à la RFC pour comprendre le fonctionnement de l'ajout d'une nouvelle Frame.
Avec cette nouvelle implémentation, c'est une usine à reader qui va permettre la création du reader spécifique à une Frame, cette classe sera notre décodeur de frame dans un byteBuffer.

Pour initialiser ce décodeur, nous allons parser chaque composant du record d'une frame à lire pour savoir quels composants il faut récupérer. Nous stockerons ensuite, pour chaque OpCode correspondant à une Frame, un array de Class représentant le constructeur de la frame à décoder.
Lorsque le décodeur aura besoin de décoder une frame, il recherchera dans la map le constructeur de celui-ci sous forme de tableaux, il va le parser et pour chaque composant il appellera son reader, exécutera la méthode process pour récupérer le component en question et l'ajouter dans une liste d'objet que l'on convertira en tableau pour enfin invoquer le constructeur de la frame à décoder.

Voici l'algorithme sous forme de code, elle gère également le cas où il faut re-remplir le buffer car il n'y a pas assez d'informations lue d'un coup :

```
var packet = opCodeToConstructors.get(opcode);
for (var i = currentPosition; i < packet.components.length; i++) {
   var component = packet.components[i];
   var reader = packetToReader.get(component);
   var status = reader.process(buffer);
   if (status == Reader.ProcessStatus.DONE) {
       currentReadingComponents.add(reader.get());
       reader.reset();
       currentPosition = 0;
   } else if (status == Reader.ProcessStatus.REFILL) {
       currentPosition = i;
       return Reader.ProcessStatus.REFILL;
   } else if (status == Reader.ProcessStatus.ERROR) {
       state = State.ERROR;
       return Reader.ProcessStatus.ERROR;
   }
}
state = State.DONE;
value = packet.packet.getDeclaredConstructor(packet.components)
       .newInstance(currentReadingComponents.toArray());
return Reader.ProcessStatus.DONE;
```

V. 4 - Automatisation de l'encodage des frames

Une autre amélioration a été la mise en place d'un système permettant à un singleton de pouvoir insérer les composants d'une frame dans un byteBuffer ainsi que de connaître sa taille de manière automatisée. Cette idée se base sur l'usine à reader, à la différence que pour ces deux méthodes put et size, la parsing des records Components est à effectuer pour chaque appel des méthode put et size. Nous avons donc implémenter un cache via la classe ClassValue pour permettre de ne lire qu'une seule fois les records components comme montré ci-dessous :

class Frames.java

```
@FunctionalInterface
private interface GreedComponentConsumer {
   void accept(Frame frame, Consumer<? super GreedComponent> greedConsumer);
}

private final static ClassValue<GreedComponentConsumer> CACHE = new ClassValue<>()
{
   @Override
   protected GreedComponentConsumer computeValue(Class<?> type) {
       return (frame, greedConsumer) -> {
           for(var component: type.getRecordComponents()){
```

```
                GreedComponent packet = (GreedComponent)
invoke(component.getAccessor(), frame);
                greedConsumer.accept(packet);
            }
        };
    }
};
```

De cette façon, il ne nous reste qu'à appeler le cache et appliquer le code pour les méthodes put et size :

```
public static <T extends Frame> void put(T frame, ByteBuffer buffer) {
    var opCode = OpCode.of(frame);
    buffer.put(frame.kind().BYTES).put(opCode.BYTES);
    CACHE.get(frame.getClass()).accept(frame, packet -> packet.putInBuffer(buffer));
}
```

A noter que l'utilisation de la ClassValue du côté des readers ne nous a pas semblé nécessaire car l'accès aux records Component de ce côté ne s'effectue qu' une seule fois durant le programme.

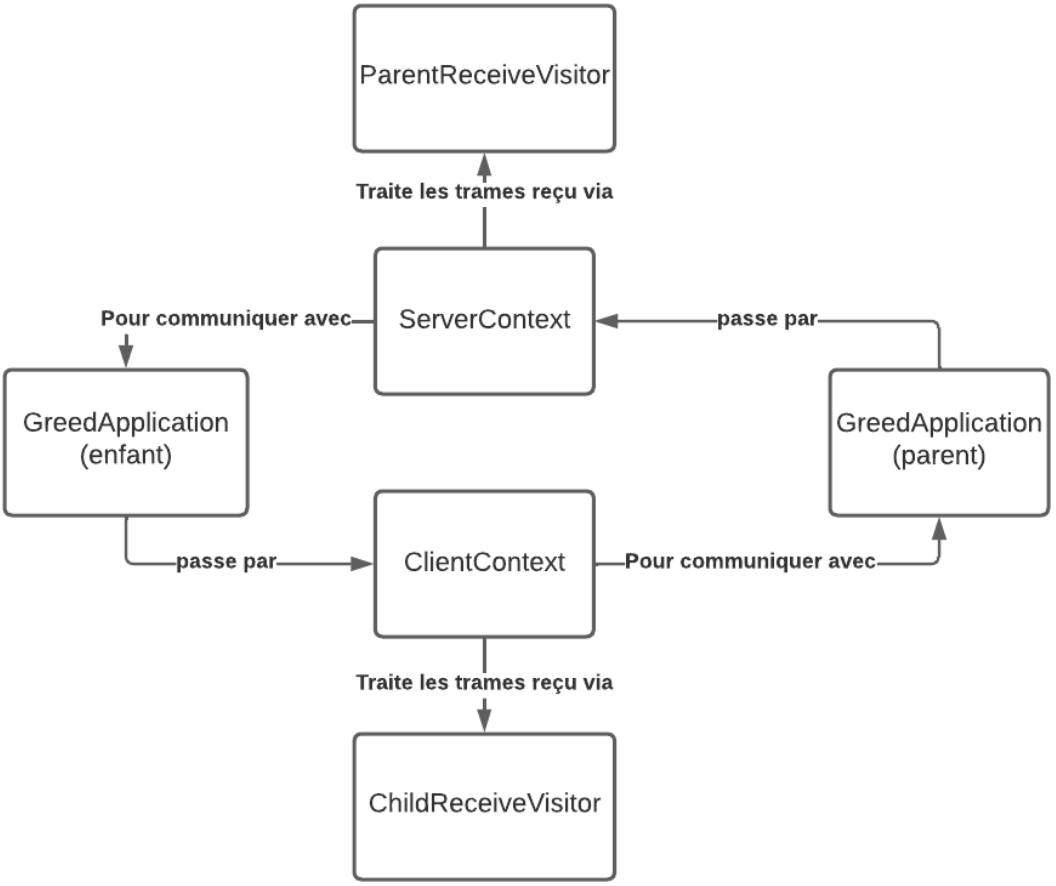
Nous avons décidé de séparer l'interface Frame en 3 sous-interfaces Local, Broadcast et Transfert Frame, dans le but de pouvoir unifier le comportement de ces 3 types de frame en un seul endroit dans le code.
Voici l'exemple de l'utilisation du visitor associé à l'implémentation des Frames décrits plus haut :

```
private void processPacket(Frame packet) {
    switch(packet) {
        case BroadcastFrame b -> {
            b.accept(visitor);
            var oldSrc = b.src().getSocket();
            server.broadcast(b.withNewSource(new IDPacket(server.getAddress())),
oldSrc);
        }
        case TransferFrame t -> {
            if(t.dst().getSocket().equals(server.getAddress())){
                t.accept(visitor);
            } else {
                server.transfer(t.dst().getSocket(), t);
            }
        }
        case LocalFrame l -> l.accept(visitor);
    }
}
```

Ainsi, l'action de base à effectuer par une Frame (local, broadcast ou transfert) est factorisé dans la méthode, et non plus à chaque implémentation de la réception de cette Frame, de plus l'action à effectuer lors de la réception de la frame est englober dans la classe du visitor via sa méthode accept.

V. 5 - Séparation des contextes

Nous avons également, comme conseillé lors de la soutenance, séparé le contexte en deux selon si l'application agit en tant que serveur, ou en tant que client. Il nous a donc fallu un long refactoring de notre système actuel pour pouvoir gérer les méthodes en commun, ou uniquement selon le contexte. Nous avons donc créé une classe abstraite Context qui contiendrait les méthodes communes de chaque cas, ainsi qu' implémenter deux sous-classes de celle-ci permettant de séparer les méthodes utilisées dans chaque context, avec respectivement pour rôle : un context utilisé pour la communication entre un client et son père, et un autre pour celle entre un serveur et son enfant. A la création de chaque contexte on lui associe un visitor, le contexte saura donc quelle action effectuer lors de la réception d'une frame. Voici ci-dessous un schéma résumant la communication entre deux applications.



V. 6 - Mémorisation des calculs à faire

La mémorisation des calculs va se faire en deux étapes:

- Dans un premier temps lorsqu'une application reçoit une demande de calcul elle va répondre avec l'intervalle qu'elle peut effectuer et elle va sauvegarder le fait qu'elle à prévu de faire tant de calculs pour éviter que si elle reçoit plusieurs demandes de calculs avant de les effectuer elle renvoie un intervalle maximum à toutes les applications et qu'ensuite elle se retrouver surcharger avec ces calculs.

- Puis lorsqu'on reçoit la demande qui nous dit d'exécuter les calculs on sauvegarde cette fois dans une autre liste qui représente les calculs en cours. Ce qui nous permet à chaque fois qu'on calcule une valeur de l'incrémenter et de savoir quand le calcul est fini et qu'on peut créer notre fichier.

On identifie un calcul avec un long et son InetSocketAddress, et ses informations avec l'url du jar à récupérer, le nom de la classe et l'intervalle.

```
public record ComputationIdentifier(long id, InetSocketAddress src) {
    1 usage  Sacane
    public String outputTitle() { return "result_" + id + ".txt"; }
}
```

```
public record ComputeInfo(String url, String className, long start, long end) {}
```

```
public final class ComputationEntity {
    2 usages
    private final ComputationIdentifier id;
    2 usages
    private final ComputeInfo info;
    5 usages
    private long intendedUc;
    5 usages
    private long currentUcDone;
    2 usages
    private long nbUcOk;
```

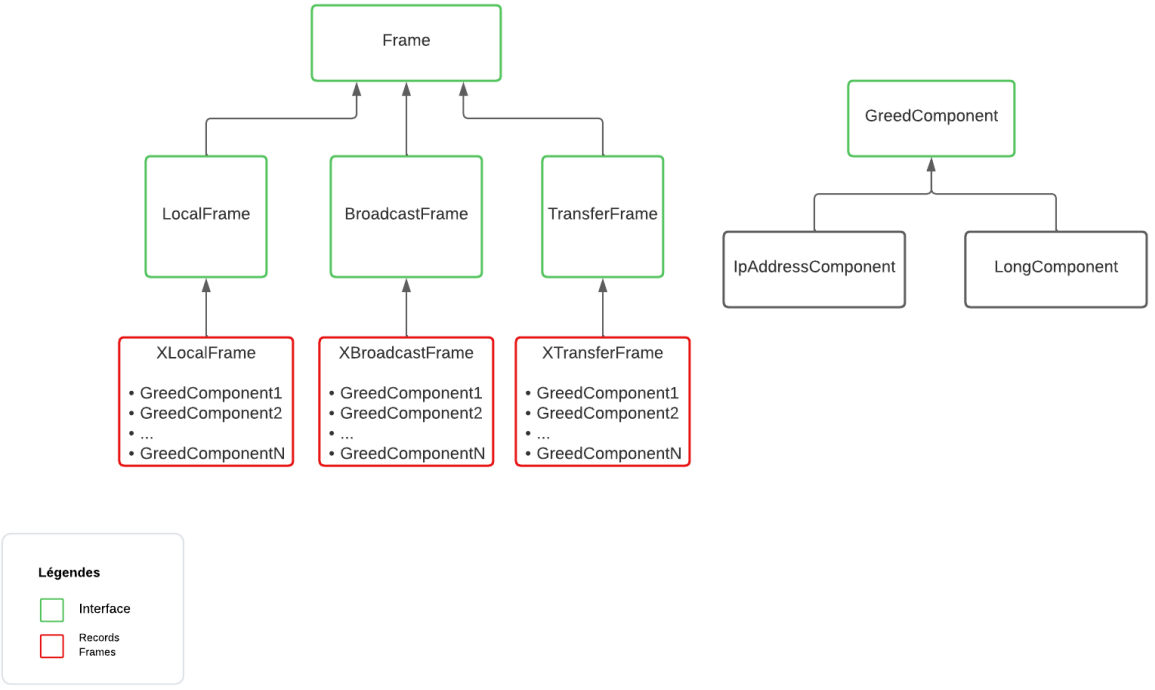
```
public final class ComputationRoomHandler {
    11 usages
    private final Object lock = new Object();
    3 usages
    private final HashMap<ComputationIdentifier, CounterIntend> prepareWaitingRoom = new HashMap<>();
    9 usages
    private final ArrayList<ComputationEntity> computations = new ArrayList<>();
```

V. 7 - Ré organisation sous forme de Component et de Frame

Initialement notre programme ne faisait pas la dissociation entre les frames et les composants, une Frame était un sous-type d'une interface qui possédait les méthodes putInBuffer et chaque composant de frame héritait de ce sous-type.
Ce design posait plusieurs problèmes :

- Une frame était un sous-type d'une interface associant également les composants.
- Aucune dissociation entre une frame et un composant donc il y avait trop d'interfaces qui héritent d'autres interfaces avec des super-interfaces qui n'étaient pas utilisées en tant que telles.
- Ce design nous empêchait d'automatiser le décodage et l'encodage des frames.

Nous avons donc opté pour un design construit autour des Frames et séparant les deux notions. Ainsi comme décrit dans la rubrique précédente, une Frame est une interface possédant 3 sous-types conformément à la rfc. Les records représentant des Frame sont eux composés uniquement d'une Greed Component, pour être certain de pouvoir être lue comme le décrit le schéma ci-dessous :



Non seulement ce design nous a permi de pouvoir rendre notre code beaucoup plus lisible et compréhensible, mais il permet également de pouvoir unifier le comportement des différentes Frames en une seule méthode :

Context.java

```
private void processFrame(Frame packet) {
   switch(packet) {
       case BroadcastFrame b -> {
           b.accept(visitor);
           var oldSrc = b.src().getSocket();
           server.broadcast(b.withNewSource(new IDComponent(server.getAddress())),
oldSrc);
       }
       case TransferFrame t -> {
           if(t.hasReachedDestination(server.getAddress())){
               t.accept(visitor);
           } else {
               server.transfer(t.dst().getSocket(), t);
           }
       }
       case LocalFrame l -> l.accept(visitor);
   }
}
```

Ainsi le comportement des différents types de frames sont traités conformément à la RFC, le serveur :

- Traite une frame broadcast et la transfère à tous ses voisins avec la nouvelle source
- Traite la frame transfert uniquement si le serveur est la destination, sinon il propage le transfère
- Traite une local Frame sans comportement particulier

VI. Conclusion

Le projet a été très intéressant à réaliser, que ce soit dans le contexte de la programmation réseau ou bien dans la pratique du langage Java. Ce projet nous a permis d'utiliser les connaissances étudiées, de rédiger une RFC, de suivre un protocole détaillé selon cette RFC, et de pouvoir de manière générale exploiter les connaissances et les bonnes pratiques sur le langage.

Plusieurs améliorations auraient pu être envisagées, notamment en termes de contrôle et de sécurité pour la communication au sein du réseau de calcul, par exemple en permettant à la root de définir un mot de passe à l'initialisation et de contrôler ce mot de passe pour chaque communication entre les demandes de calculs. Il aurait également été utile de gérer les cas des grosses pannes d'une application.