|  |  |
| --- | --- |
|  | **Projet LO23 ChessP2P** |
|  | Auteur(s) :   * Communication et traitements * Gestion de données * IHM Connexion * IHM Grille |

|  |
| --- |
| **Dossier de Réalisation**  **Réalisation d’un jeu d’échec en réseau décentralisé** |
|  |

Table des matières

[1 Introduction 4](#_Toc345347414)

[2 Communication et traitement 5](#_Toc345347415)

[2.1 Introduction 5](#_Toc345347416)

[2.2 Choix technologiques 5](#_Toc345347417)

[2.2.1 La découverte des joueurs 5](#_Toc345347418)

[2.2.2 Les dialogues entre adversaires 6](#_Toc345347419)

[2.3 Difficultés rencontrées 6](#_Toc345347420)

[2.3.1 Synchronisation des threads 6](#_Toc345347421)

[2.3.2 Thread MAJ interface graphique 6](#_Toc345347422)

[2.3.3 Apprentissage de JAVA 7](#_Toc345347423)

[2.3.4 Envoi des profil en UDP 7](#_Toc345347424)

[2.4 Points particuliers 7](#_Toc345347425)

[2.4.1 Polymorphisme 7](#_Toc345347426)

[2.4.2 Récupération adresse IP machine 8](#_Toc345347427)

[3 Gestion des données 9](#_Toc345347428)

[3.1 Introduction 9](#_Toc345347429)

[3.2 Choix technologiques 9](#_Toc345347430)

[3.3 Difficultés rencontrées 9](#_Toc345347431)

[3.4 Points particuliers 9](#_Toc345347432)

[4 IHM connexion 10](#_Toc345347433)

[4.1 Introduction 10](#_Toc345347434)

[4.2 Choix technologiques 10](#_Toc345347435)

[4.2.1 Netbeans 10](#_Toc345347436)

[4.2.2 Swing 10](#_Toc345347437)

[4.3 Difficultés rencontrées 11](#_Toc345347438)

[4.3.1 Partie 1 : Connexion et gestion de profil 11](#_Toc345347439)

[4.3.2 Partie 2 : Liste des joueurs connectés et gestion des invitations 11](#_Toc345347440)

[4.3.3 Partie3 : Liste des parties terminées et parties en cours 12](#_Toc345347441)

[4.4 Points particuliers 12](#_Toc345347442)

[5 IHM grille 13](#_Toc345347443)

[5.1 Introduction 13](#_Toc345347444)

[5.2 Choix technologiques 13](#_Toc345347445)

[5.3 Difficultés rencontrées 13](#_Toc345347446)

[5.4 Points particuliers 13](#_Toc345347447)

[6 Conclusion générale 14](#_Toc345347448)

# Introduction

# Communication et traitement

## Introduction

Une fois l’établissement du dossier de conception terminée, l’équipe du module « communication et traitement » a entamé la phase de réalisation de son module et des différentes fonctionnalités qui lui sont associées. Nous avons veillé à répartir le travail en fonction des compétences et des disponibilités des membres du groupe. Au cours de cette phase de développement, nous avons dû faire face à des problématiques diverses et donc des choix techniques à faire.

Dans un premier temps, nous allons mettre en lumière les divers choix technologiques que nous avons faits en les expliquant. Ensuite nous nous attacherons aux principales difficultés rencontrées. Et pour finir, nous évoquerons divers points spécifiques.

## Choix technologiques

Nous présenterons dans cette partie les divers choix techniques que nous avons dû prendre tout en argumentant nos choix.

### La découverte des joueurs

Lorsque l’utilisateur lance une partie, il doit être capable de choisir son adversaire parmi la liste des utilisateurs connectés sur le même réseau que lui. De plus, il nous fallait un moyen de lister les utilisateurs sans serveur étant donné que l’application doit être décentralisée.

Après avoir fait quelques recherches et quelques tests, nous avons décidé d'utiliser le mode multicast pour l'envoie d'un message à tous les joueurs disponibles lors de la recherche d’un joueur. Il nous a semblé que le mode multicast était plus simple à mettre en œuvre que le mode broadcast car on n’a pas besoin de connaître précisément la topologie du réseau pour envoyer un message. De plus, on souhaite communiquer seulement avec ceux qui lance l’application, ce qui justifie aussi l’utilisation du Multicast.

Lors de la recherche d’un joueur, voici les points importants :

1) Envoi d’une demande d’informations de la part de tous les autres joueurs connectés au réseau. Lorsqu’un utilisateur reçoit cette demande, il envoie à l’émetteur un message contenant l’ensemble des informations constituant son profil.

2) Pour mettre à jour la liste un bouton permettant de renvoyer le message de demande d’information sera implémenté.

3) Les joueurs doivent être capables de gérer plusieurs demandes de parties simultanément.

Il faudra donc utiliser java.net.MulticastSocket pour lire les paquets reçus en mode multicast et java.net.DatagramSocket pour envoyer un paquet en mode non-connecté (UDP). Enfin, il faudra également utiliser une adresse IP multicast comprise entre 224.0.0.0 et 239.255.255.255 en vérifiant bien que cette adresse n'est pas déjà réservée.

### Les dialogues entre adversaires

Lorsque 2 joueurs se sont mis d’accord pour lancer une partie, les joueurs doivent être capable de dialoguer pour s’envoyer des messages via le chat ou les déplacements des pions.

Pour se faire, losrque deux joueurs lancent une partie, une connexion TCP entre les deux est initialisée. Nous utilisons java.net.Socket et java.net.ServerSocket qui permettent de créer une connexion TCP assez simplement. Par exemple, on peut envoyer et recevoir directement des objets Java sérialisables via java.io.ObjectOutputStream et java.io.ObjectInputStream et une exception se produit s'il se produit un problème avec la connexion.

La réception et l’envoi des messages se feront par le biais de handlers, lors de la reception d’un message, le message sera transmis vers le module correspondant (pour la plus part des cas, ce sera le module traitement de données).

## Difficultés rencontrées

Nous évoquerons dans cette partie les différentes difficultés que nous avons rencontrées et qui nous ont amenées à adapter notre code.

### Synchronisation des threads

Pour être capable de recevoir les messages venant de l’adversaire tout en gardant le contrôle sur l’application, l’utilisation des threads semble être le choix le plus judicieux.

Quand on veut récupérer des messages du réseau, les appels des méthodes concernées sont bloquants, c'est-à-dire que le thread est en attente jusqu’à la réception effective du message. Il faudra donc mettre en place des threads dédiés à la réception des messages mais aussi faire très attention aux problèmes de synchronisation entre les threads. Par exemple, si on envoie un message et qu’au même moment on a reçu un message du réseau alors il est possible d’avoir des problèmes d’accès concurrent à certaines ressources.

Pour répondre à ce problème, nous avons développé plusieurs handlers, un handler de reception de messages, un autre pour l’envoi de messages, et un dernier qui contient les deux précédents permettant la synchronisation des ressources.

### Thread MAJ interface graphique

Un autre problème vient de Swing qui ne peut mettre à jour son interface graphique que dans un thread particulier, le Event Dispacher Thread, appelé aussi EDT alors que nous avons différents threads dédiés au réseau. Nous avons décidé de traiter cette problématique dans notre module pour ne pas la répandre dans les autres modules notamment les modules IHM. Nous devrons donc utiliser des méthodes et des objets spécifiques comme SwingUtilities ou SwingPropertyChangeSupport pour exécuter les interfaces des autres modules dans le thread EDT.

### Apprentissage de JAVA

Il s’est avéré que certain membres du groupe n’avait jamais pratiqué le langage JAVA, une période d’apprentissage (auto-formation) a été prise en compte pour comprendre les principes et les outlis spécifiques à JAVA.

### Envoi des profil en UDP

Quand un joueur cherche un adversaire, l’ensemble des informations qui concernent les joueurs sont envoyées sur le réseau en multicast et donc en UDP. Lors de la phase de test des solutions techniques, nous avions testé d’envoyer des profils sans avatar. Cependant lorsque l’on rajoute un avatar au profil, les datagrammes à envoyer deviennent trop gros et ne peuvent plus être envoyé en une seule fois.

Nous avions donc plusieurs choix possible pour résoudre ce problème :

1) Etablir une connexion TCP pour tout échange de profil. L’inconvénient est que cette méthode risque d’être très couteuse à partir d’un certain nombre de joueurs connectés.

2) Mettre en place un système permettant de gérer l’envoi de datagrammes segmentés qui prend en compte le multicast. L’inconvénient de cette solution est qu’elle risquait de prendre beaucoup de temp de développement.

3) Limiter la taille des avatars de manière à contrôler la taille maximum des datagrammes. C’est la solution que nous avons choisi car c’est celle qui a le meilleur rapport Temps de développement / Sécurité.

## Points particuliers

Cette partie traitera de divers points ayant été soulevés avant, pendant et après la phase de réalisation du module communication et traitement.

### Polymorphisme

En effet, au lieu de tester la classe de chaque message reçu afin d’appeler le traitement correspondant, nous aurions pu utiliser le concept de polymorphisme. Nous aurions défini une classe abstraite « Message »possédant une méthode « traiter » et nous aurions redéfini cette méthode dans chacune des classes filles afin de spécifier le comportement à adopter vis-à-vis de chaque type de message. Ceci nous aurait évité de tester explicitement la classe du message reçu dans les Handlers liés à la réception de messages.

### Récupération adresse IP machine

L’application cliente étant identifiée sur le réseau via son profil (contenant une adresse IP) et le fait qu’un ordinateur muni de plusieurs cartes réseaux puisse disposer de plusieurs adresses IP a soulevé une question : quelle adresse IP devons-nous récupérer lorsque plusieurs sont disponibles (cas de la majorité des pc portables munis d’une interface réseau WIFI et Ethernet). Afin de répondre à cette question, il a été décidé que nous écrirons une fonction permettant de sélectionner la première adresse IP locale étant différente de celle de loopback.

# Gestion des données

## Introduction

Lors de la phase de conception, le module Gestion de données a procédé à des choix sur les technologies à employer sur certains aspects critiques du module. Nous discuterons ici de ces choix.

Par ailleurs, le travail a été effectué en binômes débutant/vétéran afin permettre la montée en compétence des membres du groupe qui n’avaient pas encore formé d’expérience sur les technologies du projet. Malgré les bonnes conditions de travail, nous avons rencontré des points de difficulté que nous exposerons.

Enfin, certains points particuliers nous ont semblé mériter quelques commentaires.

## Choix technologiques

### Sérialisation

La persistance ou sérialisation est un mécanisme technique qui permet de stocker les objets dans des fichiers (on aurait aussi pu imaginer une base de données mais cette solution n’est pas adaptée dans notre cas). Nous avons eu une réflexion sur le format de la persistance (XML ? JSON ? binaire ?) qui a donné lieu à une étude technique.

Au début, nous nous étions mis d’accord sur une persistance en JSON (privilégié par rapport à XML pour sa plus grande concision) puisque ce format aurait permis une évolution de l’application plus intéressante (réutilisation par des terminaux mobiles par exemple) par rapport à une persistance binaire.

Bien qu'il existe des librairies très puissantes pour la sérialisation en JSON  
(par exemple google-gson, developpée par Google : http://code.google.com/p/google-gson/), nous avons finalement privilégié la sérialisation Java native (binaire). En effet, celle-ci est la seule à supporter les références circulaires dans les objets sérialisés. Étant donné que supprimer les  
références circulaires dans notre graphe d'objet aurait représenté un travail important, nous avons décidé que le jeu n'en valait pas la chandelle.

La seule contrainte de ce mode de persistance est la nécessité de l’implémentation de l’interface Java Serializable. Cette dernière ne nécessitant aucune implémentation de méthode, la contrainte est extrêmement faible.

De plus, nous avons du réfléchir à la mise en place de la persistance (méthode de sauvegarde/chargement dans les objets vs objet externe chargé de la persistance).

Comme nous avons opté pour une architecture données/manager, il nous a semblé naturel de respecter le même schéma pour la persistance, pour des raisons de cohérence. Une classe (le Serializer, non présenté sur le diagramme de classe dans un souci de clarté, ce dernier n’apportant rien à la compréhension du problème) aura donc la responsabilité de faire sauvegarder et charger les objets dans des fichiers.

### Observation des événements

Au début du développement, nous avons mis en balance l’utilisation de l’interface Java Observable afin d’implémenter l’annonce d’événements avec l’utilisation de la classe PropertyChangeSupport. Finalement, c’est PropertyChangeSupport qui a été retenue car elle correspond à une solution clé-en-main pour notre problème particulier. En effet, la méthode publish() permet d’envoyer une information sur une chaîne dédiée. Tous les objets qui ont souscrit avec la méthode subscribe() sont notifiés du changement. Par ailleurs, on peut passer un string en complément d’information ce qui est précieux vis-à-vis de notre conception.

## Difficultés rencontrées

### Java, NetBeans, et Subversion

Dans le groupe, les compétences sur le langage Java et le JDK, l’IDE NetBeans, ou encore la manipulation de Subversion étaient hétérogènes au départ. Nous avons opté pour un travail par binôme avec un vétéran et un débutant, afin de faire monter en compétences les étudiants qui n’avaient jamais eu d’expérience sur ces technologies.

Le challenge était de taille car même si le langage Java est relativement accessible aux étudiants ayant fait de l’Objet auparavant (LO21), le concept de SVN ou même l’environnement NetBeans sont fortement étrangers au premier abord pour un débutant. Après la période de développement, nous avons constaté que la stratégie avait payé car les binômes se montraient très capables.

# IHM connexion

## Introduction

Lors de la phase de réalisation, nous avons donc implémenté les différentes interfaces (connexion, gestion de profil, liste des joueurs connectés, liste des parties terminées et en cours) définies dans notre étude préalable de conception. Défini le plus clairement et explicitement possible, le dossier de conception n’était cependant pas parfait et des choix ou modifications ont dû être effectuées. Ce fut essentiellement à ce niveau que notre module a rencontré des difficultés.

Concernant la répartition des tâches, nous avons opté pour une séparation en binôme de notre groupe. Chaque binôme a ainsi été affecté à la réalisation d’interfaces différentes. Cette méthode d’organisation nous a permis d’assurer un équilibre dans les compétences techniques de chaque binôme.

Nous expliciterons donc les différents choix technologiques définis avant la phase de développement pour ensuite décrire les difficultés rencontrées ainsi que les points particuliers de cette étape de réalisation.

## Choix technologiques

### Netbeans

L’IDE retenu pour l’implémentation de l’application P2P-Chess a été Netbeans. Notre choix était mitigé entre celui-ci et Eclipse qui est un autre IDE java très populaire. Cependant, l’interface de Netbeans est moins lourd que Eclipse et permet les mêmes fonctionnalités nécessaires à notre projet (renommage des variables dans tout le code, auto complétion, SVN, ajout de getters/ setters automatiquement...). De plus, tous les membres du groupe était familiarisé avec Netbeans. Il nous a donc paru plus judicieux d’utiliser cet IDE plutôt que Eclipse.

### Swing

Pour l’interface de notre application, la librairie Swing a été retenue. Celle-ci est intégrée à J2SE et permet un développement d’interface optimisé et multiplateformes car elle est écrite complètement en Java. Comparé à Abstract  Window Toolkit dont les composants sont écrits directement pour une plateforme spécifique, les performances de Swing sont un peu moindres mais cela est finalement imperceptible pour les utilisateurs finaux.

Swing permet une utilisation de l’architecture MVC où les données et leurs représentations sont découplées afin d’assurer une plus grande modularité et une plus grande séparation des différentes tâches. Elle utilise la programmation par événements pour signaler les changements du modèle. Par exemple si la liste des joueurs change, un signal sera envoyé à Swing qui actualisera l’affichage de la fenêtre dont un des  composants dépend de cette liste (modèle).

De plus, Swing permet la création de plusieurs threads. Cette fonctionnalité nous est indispensable pour l’application car un thread est chargé d’écouter sur le réseau et un thread est chargé de l’interface graphique.

Enfin, l’intégration de Swing a Netbeans rend très rapide la création d’interface puisqu’une interface WYSIWYG est disponible pour la génération automatique du code des fenêtres souhaitées.

## Difficultés rencontrées

Dans cette partie, nous détaillerons les difficultés rencontrées lors de la phase de réalisation pour chacune des parties internes du module « IHM Login »

### Partie 1 : Connexion et gestion de profil

Pour la connexion, nous avons utilisé un système de clé unique (**UUID**) pour chaque couple login/ mot de passe. Or la liste des joueurs pouvant se connecter se référait uniquement à ces UUID pour afficher les joueurs différents. La connexion s’effectuant en fonction d’un couple login/ mot de passe, il nous a fallu régler le problème du cas “deux pseudos identiques” qui aurait causé des dysfonctionnements. Pour ce faire, avant l’import ou la création de profil, il y a une vérification effectuée et dans le cas où les pseudos seraient identiques alors l’utilisateur est invité à redéfinir son pseudo. Cela évite ainsi des problèmes du fonctionnement de l’application par la suite.

Pour la gestion de profil, il a fallu mettre en place un système de redimensionnement des images du joueur pour qu’elles soient facilement serialisable. Data utilisant des packets UDP pour le broadcast des profils vers tout les joueurs, il fallait absolument garder une taille d’image assez faible pour éviter la corruption de l’image. Nous avons donc effectué une homothétie de l’image pour la redimensionner à un format acceptable.

### Partie 2 : Liste des joueurs connectés et gestion des invitations

L’une des difficultés principales que nous avons rencontrée dans cette partie est liée à la mise à jour de la liste des joueurs à chaque fois qu’un nouveau joueur se connecte ou se déconnecte. En effet, il fallait que la liste se rafraîchisse automatiquement lorsqu’un joueur arrivait dans la partie ou la quittait. Il a donc fallu mettre en place le pattern Observeur-Observable qui peut être implémenté de deux manières différentes en Java : avec les interfaces **Observable** et avec le système **PropertyChange** disponible sur les JavaBeans. Nous avons donc choisi le système PropertyChangeListener ainsi qu’un abonnement au Data pour qu’il nous envoie les bons signaux en cas de connexion ou de déconnexion d’un joueur. Il a donc fallu se mettre en contact avec le groupe Data.

D’autre part, il a aussi fallu afficher des images et des boutons dans le composant JTable représentant la liste des joueurs. Nous avons du comprendre comment marchait en profondeur le système des cellules du tableau et le système des Renderer (qui décident de l’affichage de la valeur d’une cellule) pour afficher l’icône verte ou rouge montrant l’état du joueur (disponible ou occupé).

Les difficultés rencontrées sont donc liées principalement à une connaissance encore limitée du langage Java mais grâce à ce projet, nous avons pu approfondir notre connaissance du langage et de la librairie d’affichage.

### Partie3 : Liste des parties terminées et parties en cours

La principale difficulté rencontrée lors de la réalisation des parties terminées et des parties en cours fût l’affichage des parties en cours uniquement pour celles dont l’adversaire est actuellement connecté. De plus, à cause d’un problème de conception, lors d’une reprise de partie, l’adversaire n’était pas invité. Ceci a été corrigé par la suite.

## Points particuliers

Nous allons mettre en avant les points importants de la phase de réalisation du module « IHM Login »

* **« IHMLoginModel »**
  + Mise à jour de la liste des parties en cours en fonction des joueurs connectés
  + Encapsulation des appels aux interfaces des autres modules
* **« IHMConnexion »** 
  + Gestion des notifications en cas de pseudo déjà utilisé lors de l’export
  + Les profils locaux sous forme de liste déroulante
* **« IHMProfile »**
  + Création de l’interface à l’aide de l’assistant graphique (pour générer le layout manager de manière plus rapide) et adaptation de celui-ci pour avoir un code propre
  + Implémentation de “switch“ pour afficher les différentes fenêtres (modification du profil, création du profil et visualisation du profil)
  + Gestion des images de profile (taille et sérialisation)
  + Intégration avec DATA
* **« IHMListPlayers »**
  + Mise à jour dynamique de la vue à partir de messages de la liste des joueurs lorsqu’un joueur se connecte ou se déconnecte
  + Abonnement à la gestion de données qui se charge d’envoyer des messages lorsqu’un joueur se connecte ou se déconnecte
  + Adaptation graphique de boutons et images dans les cellules
  + Réception des messages d’invitation et création des fenêtres d’alerte
* **« IHMListGames »**
  + Mise à jour dynamique de la vue en fonction des joueurs connectés
  + Adaptation graphique de boutons et images dans les cellules

# IHM grille

## Introduction

## Choix technologiques

## Difficultés rencontrées

## Points particuliers

# Conclusion générale