Projet LO23 ChessP2P

08/01/2013

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

Le projet ChessP2P consiste à la réalisation d'un jeu d'échec décentralisé dans le cadre de l'unité de valeur LO23 à l'Université de Technologie de Compiègne, concernant la conduite de projets informatiques. Dans ce document, nous retraçons, par module, les différentes étapes de réalisation du projet.

Après avoir effectué la conception du projet et mis en place l'organisation et la planification du projet, chaque module a pu se mettre à la réalisation concrète du projet. Nous verrons ainsi que, tout d'abord, il a fallu s'approprier la conception et décider de choix techniques permettant l'implémentation des fonctions demandées.

Ce document montre ensuite les difficultés qui ont pu être rencontrées par chacun des modules au cours du développement, en s'intéressant aux causes et aux solutions proposées pour résoudre ces problèmes.

Enfin, pour certains modules, nous mettrons en avant des points particuliers soulevés durant la réalisation du projet. Ces points particuliers concernent des éléments qui nous auront constitués un questionnement particulier ou des phases critiques dans le développement.

2 Communication et traitement

2.1 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.

2.2 Choix technologiques

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

2.2.1 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.

2.2.2 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).

2.3 Difficultés rencontrées

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

2.3.1 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 réception de messages, un autre pour l'envoi de messages, et un dernier qui contient les deux précédents permettant la synchronisation des ressources.

2.3.2 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.

2.3.3 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.

2.3.4 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.

2.4 Points particuliers

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

2.4.1 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.

2.4.2 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.

3 Gestion des données

3.1 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.

3.2 Choix technologiques

3.2.1 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.

3.2.2 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.

3.3 Difficultés rencontrées

3.3.1 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.

4 IHM connexion

4.1 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.

4.2 Choix technologiques

4.2.1 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.

4.2.2 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.

4.3 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 »

4.3.1 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.

4.3.2 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.

4.3.3 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.

4.4 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

5 IHM grille

5.1 Introduction

Notre module s'est attaché à réaliser l'interface de jeu, c'est-à-dire l'échiquier, le chat, les différentes éléments de jeux définis lors de la phase de conception (chargement d'une partie, revisionnage, coups spéciaux …). Nous avons également réalisé la gestion des coups possibles et des règles du jeu. Lors de la conception, nous nous sommes attachés à définir avec précision les phases de développements et les interfaces à réaliser en accords avec les autres modules, durant la phase de réalisation nous nous sommes efforcer à respecter nos choix de conception. Cependant cette dernière n'étant pas parfaite, nous avons du prendre des décisions pour trouver des solutions à des problèmes qui n'avaient pas été envisagés au préalable.

5.2 Choix technologiques

5.2.1 Netbeans

Notre choix d'IDE s'est porté sur Netbeans, ce choix avait été fait dès le début du projet au sein du groupe de projet afin d'uniformiser le développement, cependant le choix de l'IDE restait libre pendant le déroulement du projet.

Nous avons choisi cet IDE car il était opérationnel sur les machines de l'UTC et permettait une bonne interconnexion avec SVN. Plus léger qu'Eclipse, il possède toutes les fonctionnalités nécessaires à ce type de projet et s'avère être assez simple d'utilisation. Enfin, le choix des responsables qualités s'étant portés sur Netbeans, toutes les informations liés à son utilisation (tutoriel, documentation …) était disponible pour faciliter le développement.

Toutefois, il s'est avéré que certains membres aient utilisé Eclipse par souci d'habitude et de confort sur leur propre machines. Ceci ne posait pas de problèmes car nous ne stockions que les fichiers sources sur SVN.

5.2.2 Swing

Concernant l'interface de notre application, nous avons utilisé la librairie Swing en accord avec le module IHM Login. Cette librairie étant l'une des plus utilisées en Java, elle possède bon nombre de fonctionnalités et à l'avantage d'être très bien documenté et de nombreuses ressources sont disponibles sur Internet.

Plusieurs fonctionnalités sont intéressantes dans Swing et nous ont apportés des solutions au développement comme le multithreading ou la gestion des layout et des panels (exemple : l'utilisation d'un GridBagLayout, une grille de layout, pour l'échiquier).

5.3 Difficultés rencontrées

Durant la phase de réalisation, nous avons rencontrés divers problèmes que nous avons pu régler en partie.

5.3.1 Interconnexions avec les autres modules

La première difficulté que nous avons rencontrée a été de gérer notre dépendance avec les autres modules. En effet, bon nombre d'outils et de codes nécessaires à notre réalisation et nos tests devaient être réalisé par les autres modules, c'est pourquoi nous avons eu une phase de latence au cours du développement qui engendré un certain retard. Nous avons donc du prolonger notre phase de développement pendant l'intégration afin d'achever notre code, des séances supplémentaires ont été requises pour notre module pour le développement et l'intégration.

5.3.2 Gestion des coordonnées de l'échiquier

Un des problèmes que nous n'avions pas perçu lors de la conception était la gestion des coordonnées de l'échiquier en fonction des différents joueurs. En effet, notre jeu dispose de trois systèmes de coordonnées différentes : celle de l'échiquier généré par le module DATA, celle de l'IHM Grille pour le joueur blanc et celle de l'IHM Grille pour le joueur noir (lors de la conception nous avons décidé d'orienter l'échiquier en fonction du joueur, de sorte que le joueur local ait ses pièces en bas de l'écran pour le confort du jeu).

Ces différences de coordonnées ont très vite engendré des problèmes, les déplacements n'étant plus cohérents les parties devenaient corrompues et le jeu devenait impossible. Pour palier à ce problème, nous avons réalisé un système de conversion des positions, pour une position donnée nous pouvons récupérer soit les coordonnées du repère de DATA, soit celle de l'IHM du joueur blanc, soit celle de l'IHM du joueur noir.

5.3.3 Gestions des règles du jeu

La gestion des règles (coups possibles) a été réalisée par l'IHM Grille après un choix de conception entre DATA et IHM Grille (le choix se portant sur qui de l'IHM Grille ou DATA allait réaliser cette partie du code). Ce choix a apporté beaucoup de travail à notre module et a donc engendré une dose de développement conséquente qui nous a pris un temps important.

a) Le roque

Le roque est un coup spécial qui permet dans des conditions particulières de déplacer le roi et la tour en même temps. Les conditions sont nombreuses. Il faut que le roi et la tour concernée ne se soit pas encore déplacés, que le roi n'est jamais été mis en échec, que les cases entre le roi et la tour soient libres et qu'elles ne soient pas en danger, c'est-à-dire qu'elles ne mettent pas le roi en échec. Néanmoins les conditions ne nous ont pas vraiment posé de problèmes. Le plus difficile à été de transmettre le coup à l'autre joueur. En effet, le déplacement de deux pièces en même temps n'était pas prévu dans les spécifications à l'origine. Nous avons donc décidé de transmettre seulement le déplacement du roi. Comme il se déplace de deux cases d'un coup, se qu'il ne peut pas faire normalement, il est obligatoirement en train de faire un roque. Il a donc fallu que nous détections le coup et que nous déplacions la tour en fonction. Cela aussi à pausé un problème puisque qu'il n'était pas prévu non plus qu'un joueur puisse faire deux coups à la suite.

b) La promotion de pion

La promotion de pion, c'est-à-dire le changement d'un pion en une nouvelle pièce lorsque celui-ci arrive au bout de l'échiquier, a posé divers problèmes lié à la synchronisation entre les deux joueurs. En effet, certains points de conception n'ont pas été vu entre IHM Grille et DATA et la gestion de promotion de pion n'était pas géré de base entre le joueur local et le joueur distant. Aussi la promotion était uniquement active chez le joueur distant.

Pour régler ce problème, nous avons vu avec le module DATA pour réaliser une notification particulière permettant d'envoyer les informations nécessaire à la promotion d'un pion. Cependant, nous n'avons pas réussi à régler tous les problèmes, à l'heure actuelle, même si la promotion est active chez les deux joueurs, l'affichage lui ne fonctionne pas chez le joueur distant qui verra un pion bouger comme une reine, un cavalier, un fou ou une tour.

c) La prise en passant

La prise en passant n'a pas été prévu dans les spécifications. Nous avons essayé de le développer à la fin mais au vue de l'avancé des autres fonctionnalités, nous avons décidés de ne pas intégrer le code dans le projet final. Il n'a en effet pas été testé.

d) Les coups classiques

La recherche des coups possibles doit être gérée différemment suivant le type de pièce. Le plus difficile à été de pouvoir tester sans avoir accès à l'échiquier. Il a donc fallu attendre la fin du développement pour pouvoir débuguer. Il a fallu aussi gérer le fait qu'une pièce ne peut pas se déplacer si cela met son propre roi en échec. Pour faire cela nous avons testé sur chaque coup possible si les pièces ennemies mettent le roi en danger.

6 Conclusion générale

Une fois la conception réalisée, il reste encore des choix à effectuer, d'ordre technique. En théorie, la réalisation ne devrait pas poser de problèmes si ces points ont été bien réalisés. Nous avons néanmoins rencontré des difficultés que nous n'avons pas pu anticiper ou qui auront été difficiles à solutionner.

D'une manière générale, le projet était complexe dans son organisation et la principale difficulté à ce stade consistait en l'interdépendance des différents modules. Un problème dans un des modules se répercutait souvent sur les autres et il fallait trouver des compromis, d'une part, pour satisfaire chaque groupe et, d'autre part, pour savoir à quel point il fallait rester fidèle à la conception.

En outre, nous avons pu réellement nous rendre compte de l'importance de la gestion de projet dans un tel contexte et de l'importance de toutes les étapes qui ont été réalisées précédemment pour la bonne conduite du projet. Si elles paraissent, au premier abord, fastidieuses et chronophages, elles prennent tout leur sens au cours de la réalisation.

Nous avons également pu réaliser la criticité de l'étape d'intégration : c'est une étape qui se prépare tout au long du projet.

Au final, nous avons pu reconnaître les qualités importantes à la gestion d'un projet : la rigueur, la communication et l'anticipation.