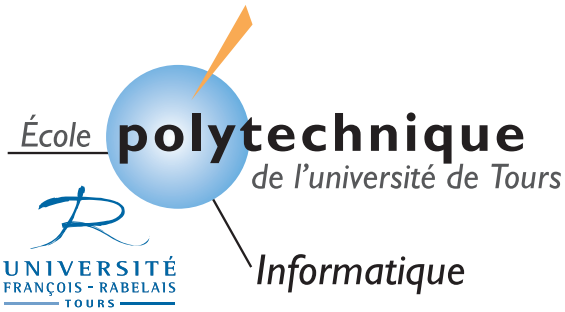

École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
Fax +33 (0)2 47 36 14 22
www.polytech.univ-tours.fr

Département Informatique
4ème année
2007-2008

Rapport du projet d'Unix

Interface graphique d'un jeu d'échecs

Étudiants :
Jonathan COURTOIS
jonathan.courtois@etu.univ-tours.fr
Florent RENAULT
florent.renault@etu.univ-tours.fr

Encadrants :
Cédric PESSAN
cedric.pessan@univ-tours.fr
Patrick MARTINEAU
patrick.martineau@univ-tours.fr
Université François-Rabelais, Tours

Version du 24 janvier 2008

Table des matières

Table des figures

Introduction

Il existe à l'heure actuelle un très grand nombre de jeux développés sous les systèmes Unix et pour les systèmes Unix. Les distributions Linux s'accompagnent parfois de jeux pré-installés dont le jeu d'échecs. Le but de ce projet n'est pas de concurrencer les jeux déjà existants, mais d'apprendre à travailler sous un environnement de type Unix et d'en maitriser les bases de la programmation système et réseau. Néanmoins, il est difficile de ne pas essayer d'atteindre un idéal tant dans le résultat que dans la conception de notre application. C'est pourquoi nous avons également accordé un temps certain au développement d'une interface agréable d'utilisation ou à l'implémentation des règles les plus marginales du jeu. Ce jeu a été baptisé "TeQChess", "TeQ" fesant référence d'une part à "Polytech'" pour la sonorité et à "Qt" (inversé), librairie utilisée pour le développement sur laquelle nous reviendrons plus tard dans ce rapport.

Ce rapport de projet Unix est décomposé en cinq chapitres. Le premier chapitre présente le cahier des charges (remis et validé par notre encadrant Cédric Pessant au début du développement). Dans un second temps nous détaillerons nos choix pour le développement de "TeQChess ; le troisième chapitre concerne les outils utilisés lors de la réalisation du projet. Pour finir, les chapitres quatre et cinq expliquent respectivement la conception de l'interface et la conception du jeu point de vue Unix.

CHAPITRE 1

Cahier des charges

1.1 Présentation du document

Ce cahier des charges rentre dans le cadre du projet d'unix de 4ème année à l'école d'ingénieur Polytech'Tours. Le demandeur est Cédric Pessan, enseignant au Département Informatique de Polytech'Tours.

1.2 Présentation du produit

Le produit demandé est un jeu d'échecs avec interface graphique sous Unix. Le logiciel doit répondre au besoin suivant :

- 2 joueurs peuvent jouer au tour par tour sur un même ordinateur.
- 2 joueurs peuvent jouer en réseau sur des ordinateurs différents.
- 1 joueur peut jouer seul sur un ordinateur contre une intelligence artificielle (GNU Chess).

1.3 Description de l'environnement

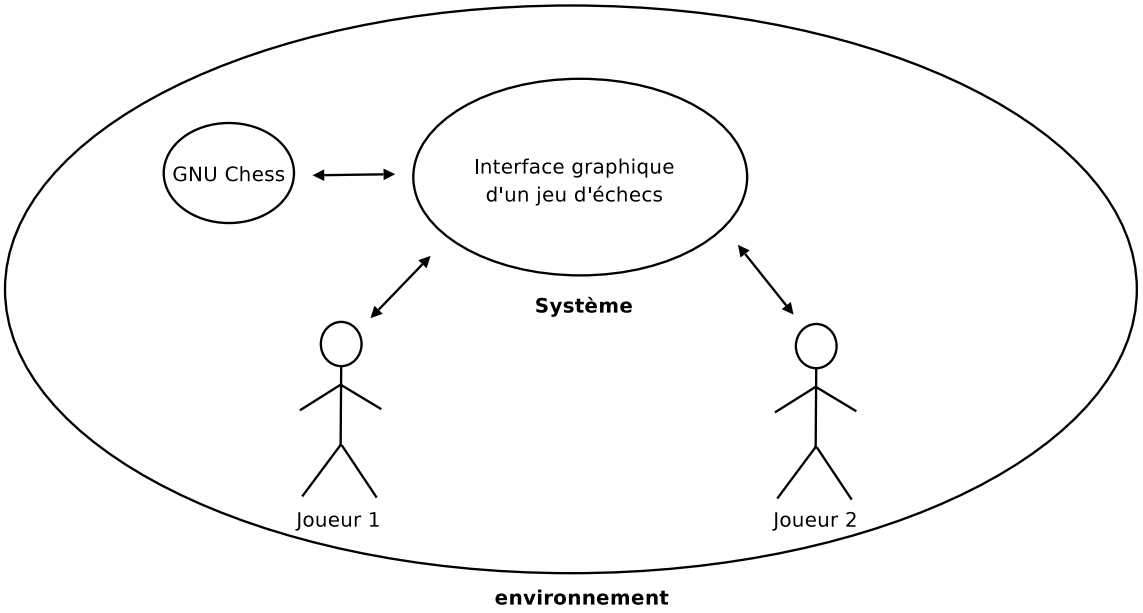

FIG. 1.1 – Schèma de l'environnement

Le système que l'on va concevoir est l'interface graphique d'un jeu d'échecs sous unix. Il pourra donc intéragir avec 1 ou 2 joueurs pour des parties sur un même ordinateur, ce qui constitue l'environnement lié au système. Le programme devra également intéragir avec GNU chess qui est un moteur de jeux d'échecs dans le cas d'une partie contre une IA (Intelligence Artificielle). Une gestion d'un mode de jeux en réseau sera également intégré au logiciel et donc au systéme.

1.4 Description des fonctions à satisfaire

Les fonctions principales que le logiciel devra satisfaire sont les suivantes :

- Permettre à deux joueurs de jouer aux échecs via une seule instance du logiciel sur la même machine.
  - Gestion du jeu au tour par tour.
- Permettre à deux joueurs de jouer aux échecs en réseau en fesant communiquer deux instances du logiciel sur des machines différentes.
  - Gestion de la communication en réseau local ou réseau distant des deux processus.
- Permettre à un joueur de jouer seul contre une intelligence artificielle utilisant GNU Chess.
  - Gestion de la communication entre l'interface de jeu et GNU Chess.

De plus le logiciel intégrera les fonctions complémentaires suivantes :

- Possibilité de régler un compteur de temps de jeu par tour.
- Zone de chat permettant aux deux joueurs de communiquer lors du jeu en réseau.
- Gestion des règles du jeu d'échecs :
  - Contrôle de la validité des déplacements des pièces.
  - Gestion des règles "spéciales" type Roque.
  - Alertes lors des mises en échec, mises en échec-et-mat.
- Zone de consultation des pièces perdues par l'un des joueurs ou par l'IA.
- Activer ou désactiver la numérotation de l'échiquier.
- Possibilité de choisir différents thèmes graphiques pour les pièces.

Pour terminer, le logiciel devra également se voir ajouter les fonctionnalités suivante de façon simple :

- Relation avec un client léger (serveur web) pour la recherche de partie sur Internet.
- Gestion des statistiques entre le programme et le serveur web.

1.5 Contraintes

Le logiciel devra être réalisé pendant la période consacrée au projet d'Unix de fin octobre 2007 à début janvier 2008. Il devra être réalisée de préférence à l'aide d'un langage orienté objet (C++, java,...) sur une plateforme Unix à destination des utilisateurs Unix. Le cahier des charges devra être rédigé et validé par l'encadrant avant de commencer la phase de réalisation du système. La prise en main du logiciel devra être facile et intuitive, et le jeu sera réalisé de la façon la plus ergonomique possible.

1.6 Documentation

Une documentation utilisateur sera fournie avec le logiciel, pour une prise en main plus facile. Une documentation du code source (classes, fonctions...) sera également mise à disposition après la phase de conception.

1.7 Schématisation de l'interface

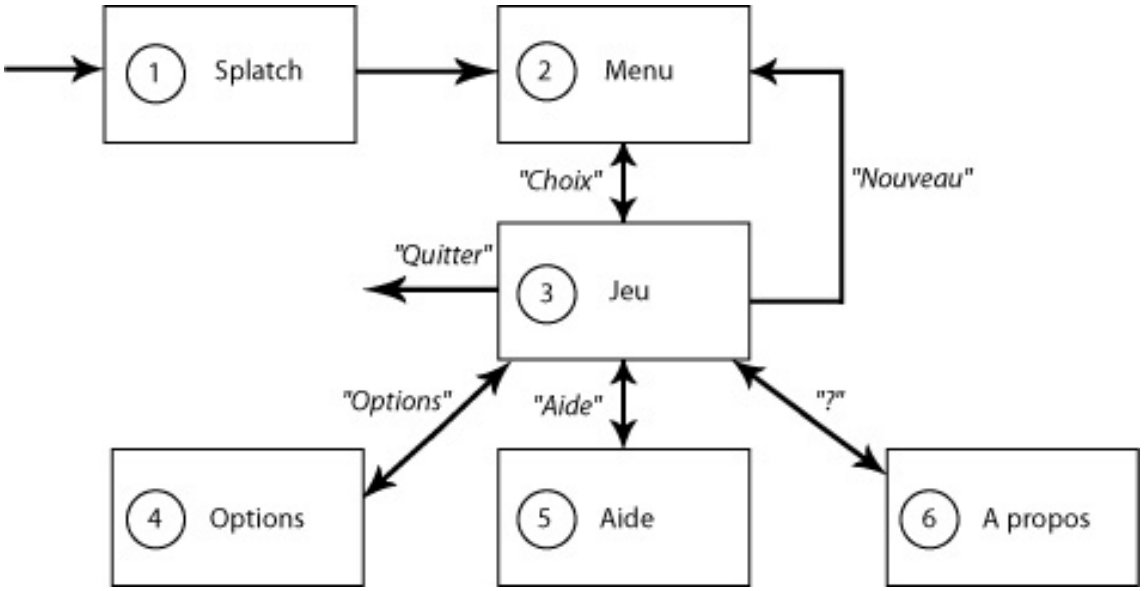

FIG. 1.2 – Schèma de l'interface

1. Splach Screen : chargement et contrôle des librairies/dépendances.
2. Menu : choix du type de partie, de la couleur et du nom de chaque joueur, du temps par tour (timer).
3. Jeu : écran principal permettant l'interaction avec l'échiquier, zone de chat.
4. Options : choix du thème, numérotation du plateau ou pas.
5. Aide : documentation diverse et aide succinte sur les règles du jeu d'échecs.
6. A propos : informations sur le programme.

CHAPITRE 2

Choix de developpement

2.1 Le type de client

Lors de la création d'un jeu, le programmeur a la possibilité de choisir entre réaliser son jeu sur un client dit "léger" ou bien sur un client dit "lourd". Le client léger désigne éssentiellement le navigateur internet, il sagit donc de creer une application "web". Le client lourd quand à lui désigne une application autonome programmée dans un langage compilé et non interprété. L'avantage du client lourd est qu'il ne dépend pas d'un serveur ou d'un quelconque autre processus et permet ainsi une plus grande autonomie. C'est donc ce type de client que nous avons choisi.

2.2 Le langage de programmation

Une multitude de langages de programmation permettent de concevoir des jeux vidéo, il n'a donc pas été facile d'en choisir un plus que les autres. Notre choix s'est porté sur un langage permettant la programmation orienté objet que l'on a étudié en cours de MOO et qui est souvent plus simple lors de la réalisation d'un jeux vidéo. Nous avons par la suite choisi d'utiliser le langage C++ pour ne pas avoir à apprendre un nouveau langage et pour anticiper le cours du 2ème semestre. La conception d'un modèle orienté objet pour le developpement d'un jeu est simple, elle permet de plus de faire évoluer simplement le jeu ou de réutiliser des classes dans un autre contexte.

2.3 La bibliothèque graphique

Une fois le langage choisi, il convient de décider quelle bibliothèque graphique utiliser pour la réalisation de l'interface graphique de notre jeu. Trois possibilités s'offraient alors à nous : SDL, GTK et Qt. SDL (Simple DirectMedia Layer) est une bibliothèque très utilisée dans le monde de la création d'applications multimédias en deux dimensions comme les jeux vidéo ou les simulations. Elle permet de gérer l'ensemble du matériel connecté à l'ordinateur, comme l'affichage vidéo, l'audio numérique, etc. Il s'agit d'une bilbiothèque très puissante et surement trop pour l'utilisation que l'on en aurais, nous avons donc décidé de mettre cette bilbiothèque de coté. Ensuite viens les 2 bibliothèques graphiques les plus connues, GTK et Qt (GTK correspondant à l'environnement gnome, et Qt correspondant à KDE). L'une comme l'autre était adaptée à ce type de projet, et le choix a été personnel du fait de l'utilisation de GTK l'an passé lors de la réalisation du projet d'algo/C et de l'envie de découvrir une nouvelle librairie. Nous avons donc opter pour la bibliothèque Qt dans sa version 4.3. L'avantage de Qt par rapport à GTK est sa conception orientée objet qui permet d'implémenter facilement un moteur de jeu également développé en objet.

2.4 Plateforme de développement

Notre développement s'est déroulé sur la distribution linux : Ubuntu 7,10. Pour programmer les différents éléments de notre interface de jeu nous avons conjointement utilisé QDevelop dans sa version 0.25 et QDesigner dans sa version 4.3.2. QDevelop est un éditeur/compilateur de code C/C++ et QDesigner est un outil de design graphique pour la librairies Qt.

CHAPITRE 3

Présentation des outils

3.1 GNU Chess

GNU Chess est un programme informatique pour jouer aux échecs. Il est un des plus vieux programmes d'échecs pour Unix et a été porté sur de nombreux environnements. Le projet GNU Chess est l'un des plus vieux paquets GNU, il a été créé en 1984. La première version a été écrite par Stuart Cracraft. Toutes les versions suivantes et précédant la version 5 ont été écrites par John Stanback. GNU Chess est un logiciel libre, sous les termes de la license GPL, et est maintenu par une collaboration de développeurs. Il est utilisé avec un environnement graphique comme XBoard. Pour anecdote, en 1998-1999, GNU Chess a subi une transition vers la version 5 qui consistait essentiellement à réécrire GNU Chess sur de nouvelles bases pour éliminer le code spaghetti. Le code spaghetti qualifie un programme dont le code n'est pas clair et qui fait un usage excessif de sauts inconditionnels (Goto).

Par défaut, GNU Chess utilise un protocole de communication abrégé qui est la notation internationalement utilisée en échecs. Elle permet d'échanger très facilement des parties stockées dans un simple fichier texte. Chaque coup est désigné par une lettre représentant une pièce suivie de sa case d'arrivée :

1. e4 e5 signifie que le pion blanc joue de e2 en e4 et que le pion noir joue de e7 en e5
2. Cf3 Cc6 signifie que le cavalier blanc joue de g1 en f3 et que le cavalier noir joue de b8 en c6
3. Fb5 a6 signifie fou blanc en b5, pion en a6
4. Fxc6 dxc6 signifie fou prend cavalier et pion d7 prend fou

Le petit roque est noté O-O, le grand roque est noté O-O-O. (lettre O).
La promotion est notée e8=Q, parfois le signe égal est omis e8Q.
Les lettres utilisées pour désigner les pièces en anglais sont K pour le roi (King), Q pour la reine (Queen), B pour le fou (Bishop), N pour le cavalier (kNight), R pour la tour (Rook) et rien pour le pion (pawn).

Ce protocole étant un peu difficile à implémenter avec notre type de déplacement, nous avons décidé d'utiliser le protocole 2 de GNU Chess qui communique avec la notation algébrique standard (SAN). Pour ce faire nous envoyons la commande "protover 2" à GNU Chess avant le début de la partie, cela permet d'éviter d'avoir en sortie l'ensemble de la réflexion de GNU Chess mais seulement le déplacement effectué sous la forme SAN :

Chaque coup est désigné par sa case de départ et d'arrivée. Aucune ambiguïté n'est possible.

1. e2e4 d7d5
2. e4d5 d8d5

La promotion est notée avec le signe égal suivi de la lettre de la pièce : e7e8=Q

Cela nous permet lors de la récupération de ce mouvement, d'extraire la case de départ (et donc la pièce se trouvant dessus) ainsi que la case d'arrivé. On peut ensuite facilement simuler un déplacement dans la partie du joueur humain, ce qui permet de ne rien changer au système de déplacement des pièces. Pour des raisons d'homogénéité et de simplicité, nous avons utilisé la même notation et la même méthode dans la partie en réseau.

3.2 Qt

Qt est une bibliothèque logicielle orientée objet et développée en C++ par la société Trolltech. Elle permet la portabilité des applications qui n'utilisent que ces composants par simple recompilation du code source. Les environnements supportés sont les Unix (dont Linux) qui utilisent le système de fenêtrage X11, Windows et Mac OSX. Qt est notamment connu pour être la bibliothèque sur laquelle repose l'environnement graphique KDE. Qt est sous licence GPL (General Public License) lorsque l'application développée était également sous GPL, pour le reste, c'est la licence commerciale qui entre en application. Cette politique de double licence a été appliqué uniquement pour Unix dans un premier temps, mais depuis la version 4.0 de Qt, elle est appliquée pour tous les systèmes.

3.2.1 Modèle objet de Qt

La classe QObject est la classe de base de tous les objets de Qt utilisant des signaux et/ou des slots. Tous les objets Qt héritant de QObject possèdent un paramètre dans le constructeur appelé parent. Les QObject s'organisent dans des arbres d'objets. Quand vous créez un QObject avec un autre QObject comme parent, l'objet parent ajoute l'autre à sa liste d'enfants. Lorsque le parent sera détruit, il supprimera automatiquement ses enfants dans son destructeur. La gestion de la mémoire est, grâce à ce mécanisme grandement facilitée et les risques de fuites mémoire sont moindres. En résumé, les seuls objets à détruire avec delete sont ceux ayant été créés avec new et ne possédant pas de paramètre parent renseigné.

3.2.2 Signaux et Slots

Le dispositif central du modèle objet de Qt est un mécanisme très puissant pour la communication d'objets faiblement couplés composé des signaux et des slots. Faiblement couplé signifie que l'émetteur d'un signal ne sait pas quel objet va le prendre en compte (il sera peut être ignoré). De la même façon, un objet interceptant un signal ne sait pas quel autre objet a émis le signal. Cette technique permet de relier entre eux des objets de types différents et offre une grande souplesse de développement. Lorsqu'un objet veut signaler que quelque chose s'est passé le concernant, un bouton qui vient d'être cliqué par exemple, il émet un signal . Un signal est également émis par un objet lorsque son état a changé. L'objet émetteur ne sait pas comment ce signal sera interprété (il ne sera peut être pas pris en compte). L'action à mener, le code à exécuter est contenu dans un slot. Les slots peuvent être employés pour recevoir des signaux, mais ce sont également des fonctions normales membres de classes. Il suffit alors de connecter (associer) le signal au slot pour que le programme exécute le code de cette méthode définie comme slot lorsque le signal sera émis. Par exemple, pour un bouton, le signal le plus utilisé est sans doute clicked() qui est émis lorsque l'utilisateur clique sur un bouton. Connecter un signal à un slot consiste à indiquer à Qt quel slot déclencher lorsqu'un signal est émis.

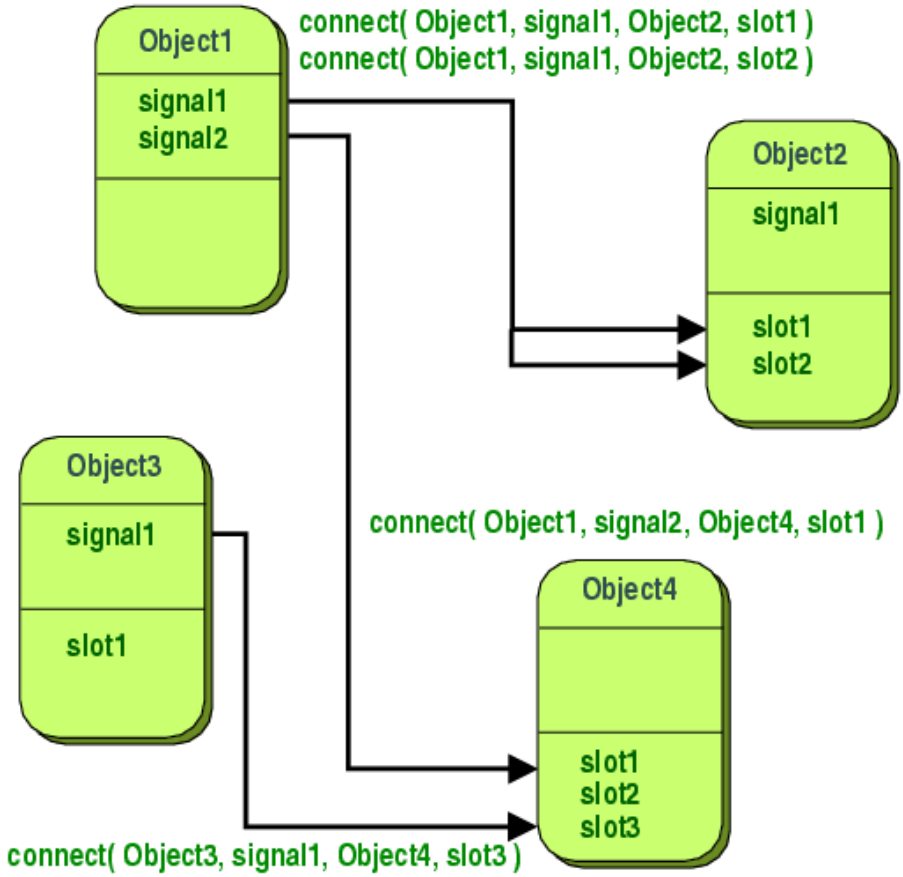

FIG. 3.1 – Schéma de connection de différents objets à l'aide du mécanisme signal/slot

3.3 QDevelop

QDevelop est un environnement de développement entièrement dédié à Qt4, il requière Qt4 et gcc. QDevelop est libre, sous license GPL, et fonctionne sur Unix, Windows et Mac OS X. Il a été développé par le français Jean-Luc Biord qui n'est autre que l'administrateur du site http://www.qtfr.org.

3.4 QDesigner

QDesigner est une interface intuitive permettant de dessiner rapidement une fenêtre ou une boite de dialogue en utilisant la bibliothèque Qt. Cette application à également été réalisé par la société Trolltech.

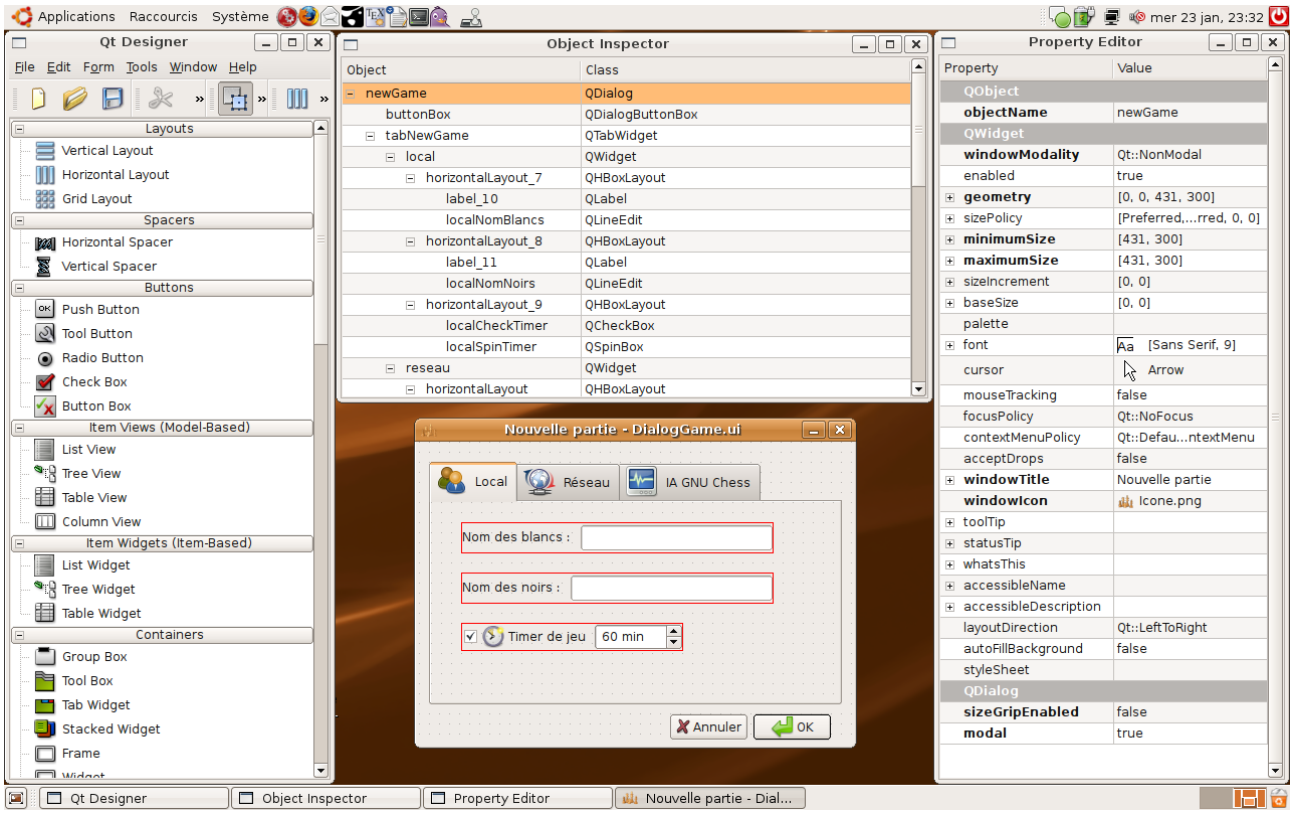

FIG. 3.2 – Interface de QDesigner

CHAPITRE 4

Conception de l'interface

Pour la réalisation de l'interface graphique de notre jeu d'échecs, nous avons réfléchi à un modèle orienté objet. Nous allons rapidement détailler la mise en place de ce modèle.

4.1 Diagramme de classe

Nous avons tout d'abord une classe **game** qui est divisée en 3 types de parties, Il s'agit donc d'une relation d'héritage. Les classes **pcgame**, **iagame** et **networkgame** héritent toutes les 3 de la classe abstraite game. Les classes **joinedgame** et **hostedgame** héritent quand à elles de la classe **networkgame**. Il s'agit ici de factoriser les éléments communs à l'ensemble des parties pour alléger l'implémentation des différentes classes. Pour faire simple, nous allons expliquer les relations entre nos différentes classes à l'aide d'un diagramme de classe représentant une partie de type 1 vs 1 sur le même ordinateur appelé ici **Chess Game** :

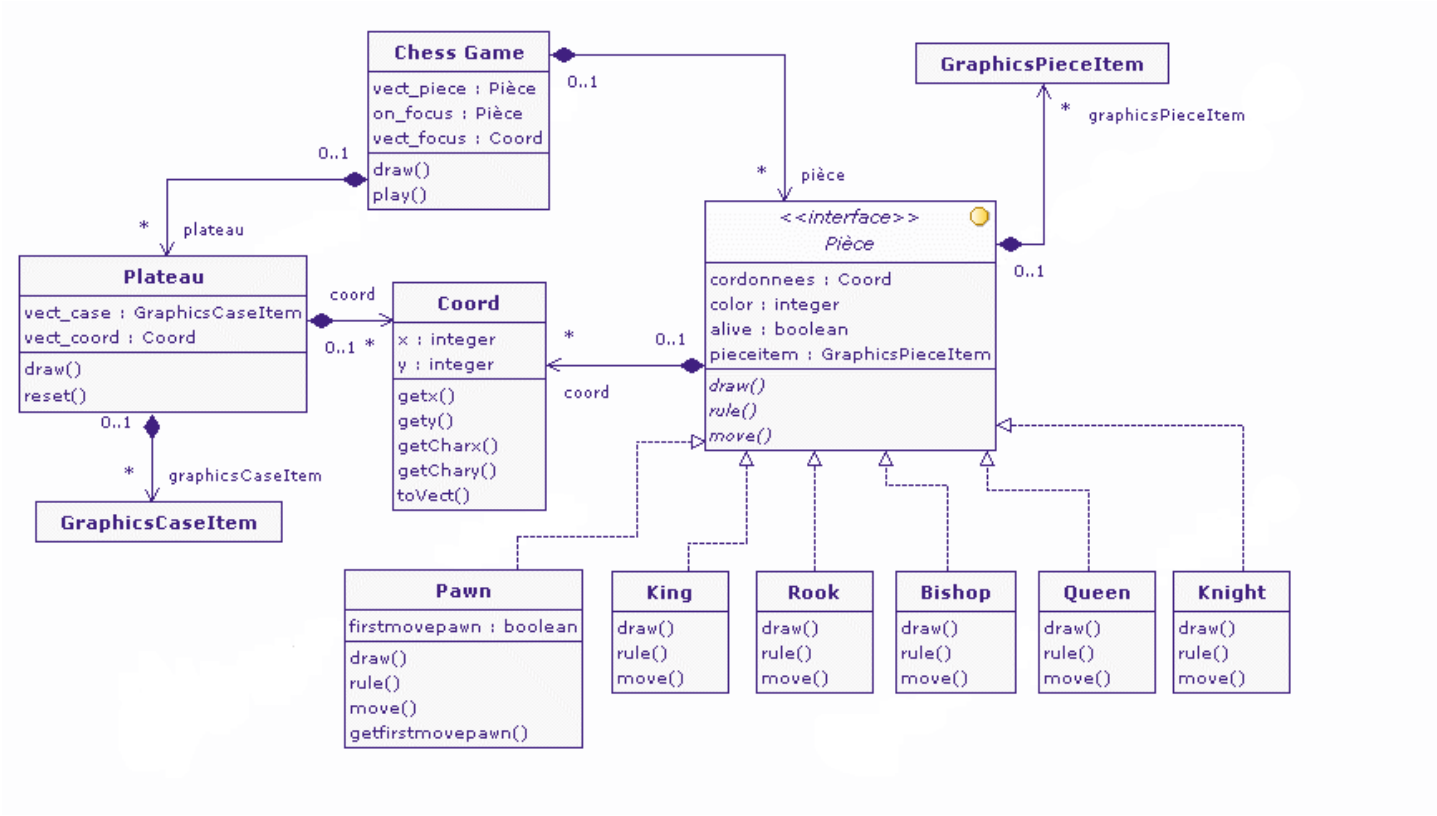

FIG. 4.1 – Diagramme de classe

La classe **Chess Game** est composée d'un plateau qui est lui même composé de 64 cases ("GraphicsCaseItem") et de 32 pièces. Il contient comme attribut un vecteur vect_piece réprésentant l'échiquier avec 64 indices comportant soit un pointeur sur la pièce se trouvant à la position correspondante soit

la valeur NULL. La variable on_focus contient la pièce séléctionnée par le joueur à un instant donné, et le vecteur vect_focus rassemble l'ensemble des coordonnées atteignables par la pièce ayant le focus. La classe coordonnée nous permet d'avoir une même coordonnée pour une case et pour une pièce et ainsi faire la corrélation entre les deux. Cette classe contient également des fonctions de conversion vers des chaines de caractères ou à partir de chaines de caractères. La classe pièce est une classe abstraite, cela implique qu'on ne peut pas créer directement une pièce mais qu'il faut obligatoire créer un objet héritié de cette classe. La classe pièce comporte différents attributs communs à l'ensemble des pièces ainsi qu'un "GraphicsPieceItem" qui correspond à la représentation de la pièce sur l'échiquier (une pixmap). La classe pièce implémente également des fonctions virtuelles comme **draw** ou **rule** qui seront toutes différentes en fonction de la pièce mais qui permettent de manipuler tous les objets pièce de la même façon sans faire de distinction entre un pion et une reine par exemple. Les classes pions (**pawn**), rois (**king**), reines (**queen**), fous (**bishop**), cavaliers (**knight**) et tours (**rook**) héritent donc de l'objet pièce. Ils implémentent chacun leur règles propres et certains possèdent des attributs supplémentaires comme la classe **pawn** qui possède une variable booléenne permetant de savoir si le pion a déjà fait un déplacement ou non.

4.2 Principe de déplacement

Le principe de jeu est le suivant, lorsque l'on clique sur une pièce, un écouteur d'évènement appelle la fonction focus, qui place la pièce dans la variable on_focus si il s'agit bien d'une pièce appartenant au joueur. Ensuite, on appelle la fonction rule qui détermine à l'aide des règles de la pièce séléctionnée et de l'état de l'échiquier contenu dans vect_piece, les différentes coordonnées atteignables par la pièce et stocke le tout dans vect_focus. Une fois une case vide ou la pièce d'une autre joueur séléctionnée, on appelle la fonction play qui teste si la case selectionnée fait bien partie des cases atteignables par la pièce qui possède le focus. Si c'est le cas, on effectue le déplacement ou les règles spéciales nécessaires, sinon on abandonne le coup.

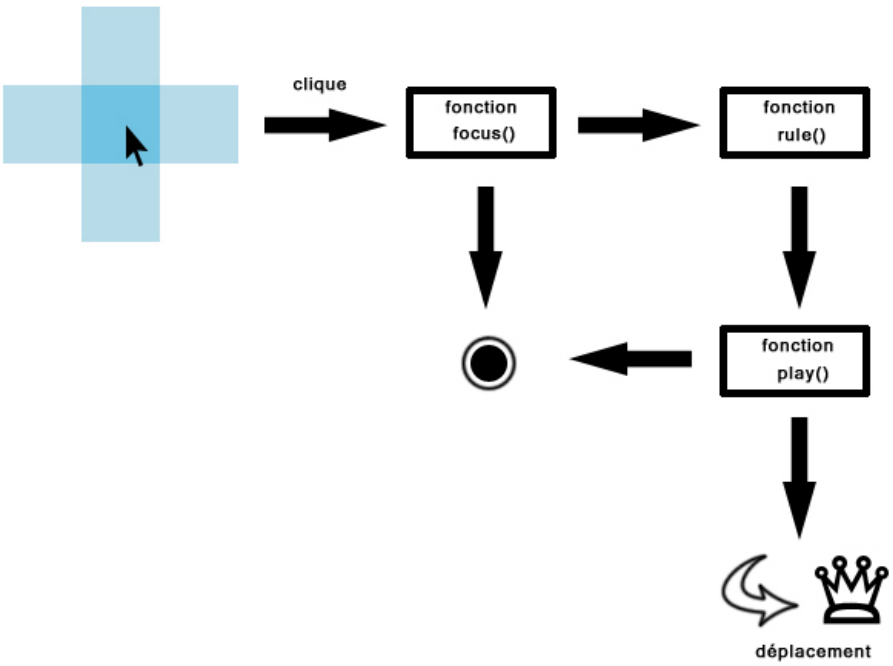

FIG. 4.2 – Explication d'un déplacement

4.3 Exemple de l'implémentation d'une fonction

L'exemple de la fonction isMenace est intéressant car c'est une fonction clé qui peut très vite alourdir, ralentir, l'exécution de notre application si elle est mal conçue.

Cette fonction a pour but de tester si une case est menacée (sous l'attaque) par une pièce adverse. Elle est utilisée dans plusieurs cas :

- Tester si le roi est en échec (on appelle alors la fonction isMenace avec les coordonnées du roi)
- Tester si le roi est échec&mat
- Tester si il est possible d'effectuer un roque (les cases entre le roi et la tour ne doivent pas être menacées)
- Gérer les règles de pièces clouées

isMenace est donc appelée très souvent et se doit d'être optimisée. La première solution est de prendre toutes les pièces du vecteur vect_piece, d'effectuer un focus sur chacune d'elles et de regarder si la case à tester fait partie d'un des vect_focus retournés. Cette solution est fonctionnelle mais ne réponds pas au besoin de rapidité précédemment cité. C'est pourquoi nous avons décidé de prendre le problème "à l'envers" et de partir de la case menacée puis d'appliquer les règles de déplacement possibles des pièces (déplacement horizontal, vertical, diagonal, "en L"). Quand une pièce est détectée lors de l'application de ces règles, on regarde si cette pièce utilise ou non la règle de déplacement utilisée.

Prenons un exemple :

Notre moteur de jeu souhaite tester si le déplacement du joueur a mis le roi adverse en échec. Le roi est en position e3. Nous allons donc dans un premier temps tester e4,e5,e6,e7 (déplacement en ligne droite verticale). Admettons qu'une tour adverse soit en e7. L'algorithme s'arrête, teste le type de la pièce : c'est une tour, elle peut se déplacer en ligne droite verticale : elle est donc menaçante pour le roi, le roi est en échec.

L'algorithme continue à tester les déplacements possibles tant qu'aucune pièce menaçante n'a été détectée. Nous avons donc classé les déplacements par ordre de probabilité afin de sortir le plus vite possible de la fonction si la case est bien menacée (on a plus de chance d'être menacé par un déplacement en ligne droite que par un déplacement "en L").

4.4 Présentation des éléments de l'interface

Nous allons maintenant vous présenter le rendu final de notre interface graphique, à l'aide de quelques captures d'écran de notre application TeQChess.

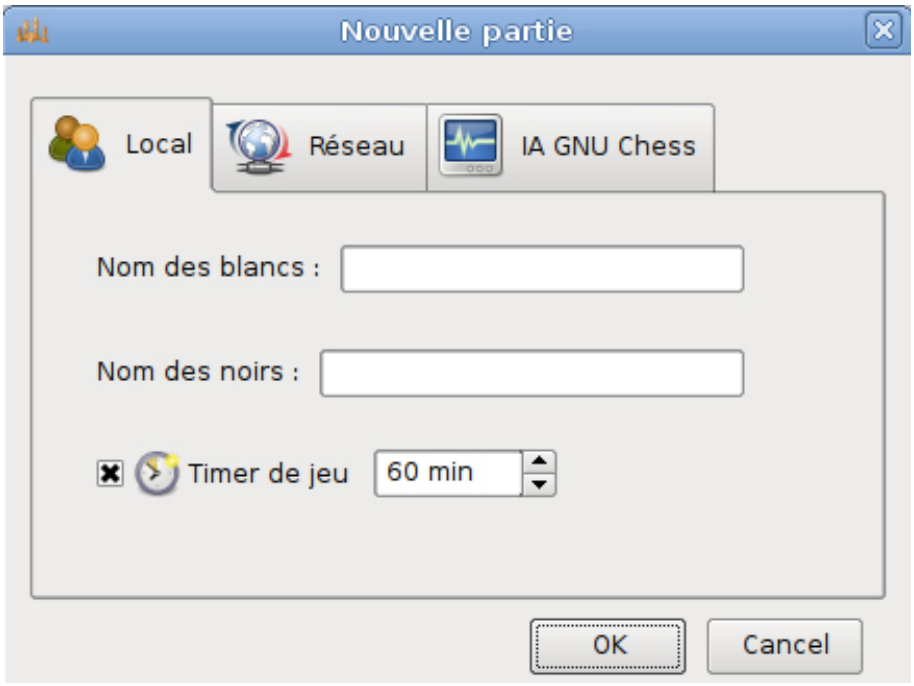

FIG. 4.3 – Fenêtre de création d'une nouvelle partie

FIG. 4.4 – Fenêtre à propos

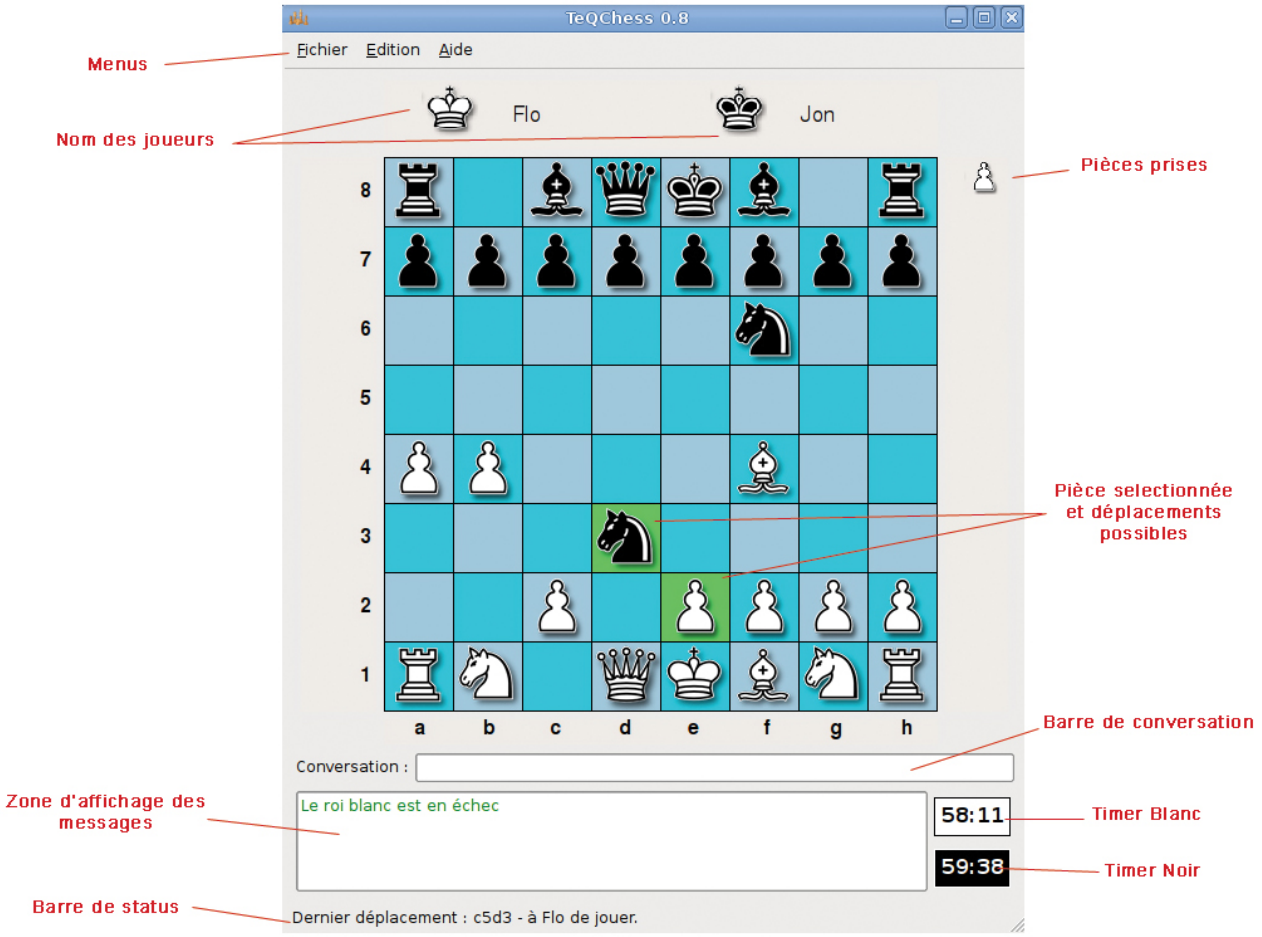

FIG. 4.5 – Fenêtre principale de TeQChess

Notre interface graphique comporte actuellement un menu options que nous avons implémenté. Il permet actuellement de changer la couleur du plateau, mais cela permet surtout à des futurs développeurs (ou à nous même) d'implémenter facilement de nouvelles fonctionnalités à notre jeu en ayant juste à ajouter un onglet à ce menu et quelques fonctions de modification dans le code source. On pourrait par exemple proposer plusieurs styles de pièces, proposer différentes organisations de l'interface, etc.

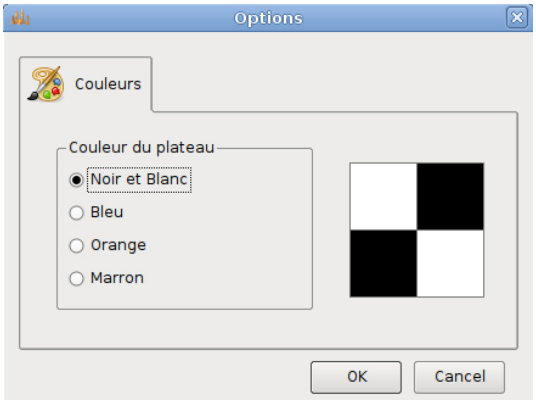

FIG. 4.6 – Fenêtre d'option

CHAPITRE 5

Conception du point de vue système Unix

Notre jeu d'échecs comporte au final 3 types de parties différentes demandées dans le cahier des charges. Une partie en 1 contre 1 sur le même ordinateur, la possibilité de jouer en réseau sur 2 ordinateurs distants avec 2 instances de notre logiciel et une partie contre une IA (Intelligence Artificielle) ici GNU Chess.

5.1 Partie réseau

La création de la classe **networkgame** et de ses deux dérivées **hostedgame** et **joinedgame** nécessite la mise en place de deux fonctions membres qui sont **host()** et **join()**. **host()** est une fonction de hostedgame qui permet de configurer le jeu en attente de connection d'un client. **join()** est une fonction de joinedgame qui permet de rejoindre une partie créer sur une machine distante. Pour réaliser ces deux fonctions nous avons été amenés à utiliser les appels système liés aux sockets vues en cours.
Voici la structure globale de la fonction **host()** :

```
/* int socket(int domaine, int type, int protocole)
AF_INET: accessible via interconnexion de réseaux
SOCK_STREAM: com en mode connecté
IPPROTO_TCP: protocole TCP */
if ((sock=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) <0 )
{
  /* échec de création de la socket */
  perror("socket");
  return 1;
}

bzero(&sin,sizeof(sin)); /* mise à zéro de la zone mémoire pour l'adresse */
sin.sin_family = AF_INET;
sin.sin_port = htons( getportS() );
sin.sin_addr.s_addr = INADDR_ANY; /* n'importe quelle adresse IP de la machine locale */

/* attachement de l'adresse à la socket */
if (bind(sock, (struct sockaddr*) &sin, sizeof(sin)) <0 )
{
  perror("bind");
  return 2;
}

/* le serveur se prépare à recevoir une connection du client */
if (listen(sock,1) <0) //1 :une seule connection possible
{
  perror("listen");
  return 3;
}

/* le serveur se met en attente de la connection du client: primitive bloquante */
namelen = sizeof(sin);
setservsock(accept(sock, (struct sockaddr*) &sin,&namelen));
if ( getservsock() <0 ) /* socket de service utilisée pour tous les échanges */
```

```
{
  perror("accept");
  return 4;
}
```

La méthode utilisée est très classique. Dans un premier temps, on créé la socket de connexion à l'aide de l'appel système **socket()**. Ensuite nous initialisons la structure d'adresse "sin". **getportS()** retourne la valeur du port utilisé pour la connection (choisi arbitrairement à 6260). Puis on attache l'adresse à la socket avec **bind()**. **listen()** permet de définir le nombre de connexions possibles, dans notre cas nous choisissons une valeur de 1. Enfin, le serveur se met en attente de son client avec l'appel système **accept()**. **accept()** retourne la socket "de service" qui sera utilisée par la suite pour communiquer avec le client (setservsock permet de définir cette socket comme donnée privée de notre classe networkgame). En cas d'erreur, **perror()** permet de renvoyer un descriptif sur la sortie d'erreur standard (par exemple le terminal si notre application a été lancée d'un terminal).

Voici la structure de la fonction **join()** :

```
if ((sock= socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) <0 )
{
  perror("socket");
  return 1;
}

bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_port = htons( getportC() );
sin.sin_addr.s_addr = INADDR_ANY;

if (bind(sock, (struct sockaddr*) &sin, sizeof(sin)) !=0 )
{
  perror("bind");
  return 2;
}

bzero(&sin,sizeof(sin));
sin.sin_family = AF_INET;
int ia = inet_aton(adresseIP,&sin.sin_addr);
if ( ia < 0)
{
  perror("inet_aton");
  cerr << adresseIP << endl;
  return 3;
}
sin.sin_port = htons( getportS() );

if (::connect(sock, (struct sockaddr*) &sin, sizeof(sin)) <0)
{
  perror("connect");
  return 4;
}
setservsock(sock);
```

Le début de cette fonction est le même que pour **host()**, on commence par créer une socket avec l'appel système **socket()**, on initialise "sin" et on assigne l'adresse à la socket avec **bind()**. **getportC()** retourne la valeur 6259. L'appel système **inet_aton()** permet de convertir l'adresse IP d'un format classique du type "xxx.xxx.xxx.xxx" en donnée binaire. Enfin **connect()** permet de se connecter au serveur dont l'adresse IP

a été passée en paramètre à la fonction **join()**. Ici on peut noter la présence de " : :" avant connect. Cet ajout est indispensable pour éviter toute ambiguité à la compilation puisqu'il existe également une fonction connect dans la librairie Qt.

5.2 Partie contre GNU Chess

Pour avoir la possibilité de jouer contre une IA déjà existante telle que GNU Chess, il a fallu réfléchir à comment l'intégrer dans notre programme. Dans un premier temps, nous avons chercher l'existence d'une librairie sur GNU Chess permettant de l'inclure facilement dans notre projet réalisé en C++ et pouvoir l'exécuter au sein de notre jeu d'échecs. Cela s'est vite révélé impossible. Nous nous sommes donc orienté sur une deuxième solution qui était de lancer le processus GNU Chess sur la machine indépendemment de notre logiciel. C'est finalement cette solution que nous avons adoptée après avoir réfléchi à la manière de procéder.

Nous avons été obligés de créer un thread pour gérer la communication entre notre programme (TeQ-Chess) et l'IA (GNU Chess). Nous expliquerons plus en détail par la suite, les méthodes mises en oeuvre dans nos threads de communications.

Au lancement du thread, nous créeons 2 tubes "send" et "receive" qui permettront la communication entre les 2 processus et nous effectuons la commande **fork()** pour pouvoir exécuter 2 processus en parallèle. Le processus fils sera GNU Chess et le processus père, notre thread de communication. Dans le processus fils (GNU Chess), nous effectuons en premier la commande **signal(SIGUSR1,SIG_DFL)** qui permet de préparer le processus à la réception d'un **kill()** pour le détruire en fin de partie. Ensuite nous réalisons la redirection des tubes, toujours à l'intérieur du fils, grâce à l'enchainement des commandes suivantes (première redirection) :

1. **close(0)**
2. **close(send[1])**
3. **dup(send[0])**
4. **close(send[0])**

Et pour terminer nous faisons un **execlp()** de la commande **"gnuchess"** qui lance le processus GNU Chess dans notre processus fils en écrasant les informations s'y trouvant (mémoire copiée du processus père).

execlp("gnuchess","gnuchess", NULL ,NULL)

Du point de vue du processus père "TeQChess", une simple fermeture des parties des tubes inutilisés est nécessaire étant donné que l'on écrira les informations à envoyé à GNU Chess à l'aide des commandes **read()** et **write()**. Ci-dessous, nous pouvons voir comment se passent les différentes redirections du point de vue de la table des fichiers par processus, de la table des fichiers ainsi que de la table des i-noeuds.

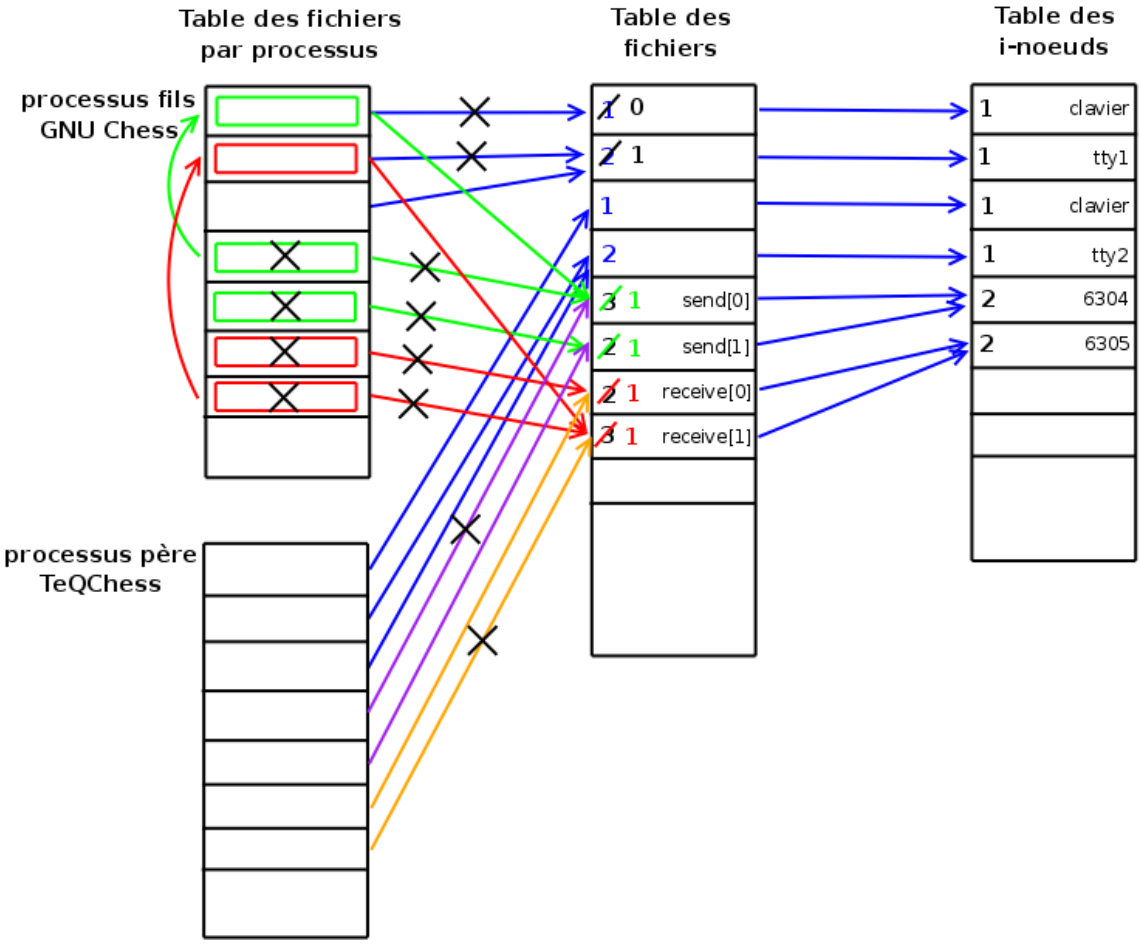

FIG. 5.1 – Schéma expliquant la redirection des tubes

Pour initialiser GNU Chess, notre application commence par lui envoyer deux commandes de configuration pour pouvoir utiliser la notation algébrique standard comme nous l'avons expliqué auparavant. Pour cela il écrit les commandes shell sous la forme d'une chaine de caractère terminée par un "\n" :

```
char xboard[8] = "xboard\n";
write(send[1],&xboard,7);
```

A ce moment là GNU Chess qui est en attente d'une saisie en entrée, lit sur **send[0]** qui est devenue après duplication, son entrée standard. Il effectue la commande et renvoie une réponse si elle a lieu d'être sur sa nouvelle sortie standard qui est **receive[1]**.

La récupération d'un coup de GNU Chess, ce fait de manière simple, le thread de communication attend indéfiniment une réponse sur le tube **receive** à l'aide des commandes suivantes :

```
while (1)
{
  if (read(receive[0],buf,max) <0){
    perror("read");
  }else{
    QString sTemp(buf);
    return extract(sTemp);
```

```
  }
}
```

On lit sur le tube jusqu'à temps que GNU Chess écrive son déplacement en faisant un read sur **receive[0]**. En cas de présence d'une chaine de caractère, il s'agit d'un mouvement effectué, par GNU Chess, on le transforme en QString puis on extrait la chaine pour pouvoir récupérer uniquement la case de départ et celle d'arrivée. Il suffit alors de mettre cette chaine dans un parser qui renvoie le déplacement sous forme d'un objet coordonnée (**coord**).

5.3 Thread de communication

Pour la création de threads, nous utilisons la classe de gestion de thread fournie par Qt. Cette classe se nomme "QThread" et se révèle très facile à utiliser au sein d'une interface développée à l'aide de widgets Qt. Cette classe est implémentée avec les API natives des systèmes d'exploitations (Win32 pour une application windows et pthread pour une application de type Unix). Dans notre cas c'est donc sur pthread que notre application s'appuie pour gérer les processus légers.
Dans un premier temps, notre choix était d'utiliser directement la librairie pthread mais nous avons rencontré un certain nombre de problèmes lors du passage en paramètre d'une fonction membre d'une classe. Pour créer un thread à l'aide de Qt il suffit de dériver la classe mère "QThread" et de réimplémenter la fonction run(). La fonction run() est l'équivalent de la fonction main() pour notre processus principal. Une fois le code de run() exécuté (retour de la fonction), le thread se termine. C'est pourquoi notre thread de communication boucle pseudo-indéfiniment en attente de données à lire.
L'intérêt de ce thread est d'éviter un "freeze" de l'interface lorsque notre application est en attente d'un message. En effet la lecture s'effectue via la primitive read() qui est une primitive bloquante. Cela permet par exemple au joueur qui est en attente de jouer d'envoyer des messages de chat à son adversaire.

La communication entre nos deux instances distantes de TeQChess se fait à l'aide d'un protocole de communication qui est le suivant :

- Les messages de jeu sont préfixés par 'G' (Game).
- Les messages de chat sont préfixés par 'C'.
- Les messages systèmes sont préfixés par 'S'.

Les messages de jeu concernent uniquement les déplacements de l'adversaire, par exemple la réception de "Ge5e3\n" signifie que la pièce en e5 se déplace en e3. La réception d'un message de jeu déclenche donc l'appel d'un parser pour analyser la chaine de caractère (ici "e5e3"). Le parser transforme cette chaine en données exploitables par notre moteur de jeu ; c'est à dire 2 objets de la classe "coord". La mise à jour de l'échiquier est effectuée et on donne ensuite la main à l'utilisateur pour lui permettre de déplacer une de ses pièces. La validité du déplacement est assurée, en amont de l'envoi du message, par le moteur de jeu (cf section "Principe de déplacement").

Les messages de chat sont traités de façon assez simple puisqu'il suffit d'afficher le contenu dans la fenêtre de chat de l'interface. L'affichage est complété par l'ajout du nom du joueur et l'heure de réception du message (cf section "Notation de l'heure"). Exemple de message de chat : "CHello World !\n".

Les messages systèmes sont les plus divers. Ils permettent en fait de gérer tout ce qui n'est pas un message de jeu ou un message de chat. Ces messages permettent par exemple d'envoyer le nom du joueur adverse ("SNxxxx\n"), la valeur d'initialisation du timer de jeu ("STxxxx\n") etc.

5.4 Détection de librairies

Notre application TeQChess utilise la librairie GNU Chess pour une partie jouer contre l'ordinateur. Pour cela, GNU Chess doit être installé au préalable. Pour éviter que l'utilisateur puisse lancer une partie **iagame** alors que GNU Chess n'est pas installé sur son ordinateur, nous avons voulu mettre au point un petit script de détection du processus en question. L'idée est de détecter la présence de la commande gnuchess qui exécute le processus. Pour cela, on utilise la commande **which gnuchess** qui renvoi 0 si le processus a été trouvé et posséde un chemin exécutable et 1 sinon. Pour utiliser cette commande shell, on se sert de l'apel à la fonction **system()** de la bibliothèque C. Cette fonction renvoie -1 en cas d'erreur ou le code de retour de la commande en cas de succès. Nous effectuons alors un test sur la valeur de retour de la fonction **system("which gnuchess")**, si elle est égale à 1, on bloque l'onglet relatif à IA game et on affiche une infobulle indiquant que GNU Chess n'est pas installé sur notre ordinateur. Actuellement cette solution ne donne pas le résulat attendu, et ne permet pas de bloquer l'onglet correspondant en cas d'absence de l'application GNU Chess.

5.5 Notation de l'heure

Pour que les joueurs puissent communiquer dans une partie en réseau ou même sur le même ordinateur en tour par tour, nous avons mis en place un petit Chat de conversation. Lors de l'envoi du message, on affiche l'heure dans la fenêtre de conversation. Pour ne pas avoir de problème de cohérence de l'heure, nous avons décidé de récupérer l'heure à afficher indépendemment sur chaque ordinateur, elle ne sera donc pas envoyée sur le socket. La récupération de l'heure se fait à l'aide de la structure *tm* pour pouvoir récupérer seulement les éléments qui nous intéresse, c'est à dire, les heures, les minutes et les secondes. On définit un variable de type *time_t* pour récupérer l'heure courante avec la fonction *time()*. Ensuite on renvoie l'heure obtenue sous la forme d'une structure *tm* à l'aide de la fonction *localtime()*. On peut pour finir récupérer facilement les différents éléments de l'heure à l'aide des variables de la structure *tm*. Voir ci-dessous un exemple simple de récupération de l'heure courante en C.

```
/* Declaration de la structure tm et d'une variable de type time_t */
struct tm today;
time_t now;

/*struct tm
{
    int    tm_sec;         // secondes
    int    tm_min;         // minutes
    int    tm_hour;       // heures
  ...
};*/

/* Renvoie l'heure actuelle sous forme du nombre de secondes écoulées depuis
 le 1er janvier 1970 à 00h 00m 00s GMT */
time(&now);

/* La fonction localtime renvoie l'heure courante sous la forme d'une structure tm*/
today = *localtime(&now);

/* Récupération des éléments de la date courante */
sec = today.tm_sec;
min = today.tm_min;
hour = today.tm_hour;
```

5.6 Temps d'attente pour host, join

Actuellement, lors de la création d'un serveur ou de la tentative de connexion à un serveur notre application devient inactive en raison des appels systèmes utilisés. L'idée est de mettre en place une temporisation à l'aide de la primitive alarm() afin de limiter le temps d'attente et d'éviter un blocage total de notre programme si aucun client ne se connecte (l'utilisateur est alors obligé de quitter l'application de façon brutale).
Pour réaliser cette petite temporisation, on commence par utiliser signal() afin de définir le comportement à adopter à la réception de signal SIGALRM.
En plaçant "alarm(60)" devant notre appel système bloquant il est alors possible de faire échouer l'appel système après 60 secondes (lequel doit alors retourner le code d'erreur EINTR).
L'opération "alarm(0)" positionnée après l'appel système permet d'annuler la temporisation si jamais l'appel système réussit avant le délai de 60 secondes.

Conclusion

Lors de la réalisation de ce projet, nous nous sommes efforcés au maximum d'utiliser Linux (distribution Ubuntu 7.10), du développement à la rédaction du rapport, afin de nous familiariser avec l'environnement. Travailler sur ce projet nous a donc permis d'améliorer nos connaissances sur Unix mais également de mieux comprendre le cours de quatrième année, voir, parfois, de l'anticiper.
Notre but était de réaliser un jeu fonctionnel qui pourrait être utilisé régulièrement par n'importe quel joueur d'échecs, du débutant grâce à l'aide au déplacement, au joueur avancé grâce à la gestion de timers de jeu ou de règles complexes comme la prise en passant.
Il est important d'ajouter que bien que ça ne soit pas le but initial du projet, nous avons pu améliorer nos connaissances en C++ et de façon plus générale nos connaissances en méthode orientée objet. En effet, la réalisation d'un jeu est une excellente opportunité pour implémenter une structure orientée objet et à l'heure actuelle nous pourrions sans problème réutiliser certaines classes pour développer un jeu de dames ou d'othello par exemple.
Pour conclure, on peut dire que bien que le nombre d'heures (non quantifiables) passées sur ce projet soit important, il reste malgré tout certains bogues et certaines fonctionnalités que nous n'avons pas eu le temps d'implémenter. Malgré tout nous sommes satisfait du résultat et utilisons d'ores et déjà TeQChess pour nos parties personnelles plutôt que le traditionnel glChess qui est plus lent (fourni avec Ubuntu).

ANNEXE A

Liens utiles

A.1 Webographie

- Site officiel de Trolltech http://trolltech.com/
- Documentation de Qt http://doc.trolltech.com/4.3/
- Forum officiel de Qt http://www.qtforum.org/index.php
- Site de la communauté francophone de Qt http://www.qtfr.org/
- Site de QDevelop http://qdevelop.free.fr/index.php
- Documentation sur GNU Chess http://www.tim-mann.org/xboard/engine-intf.html
- Système de notation de GNU Chess http://www.iechecs.com/notation.htm
- Règle de jeux d'échecs http://fr.wikipedia.org/wiki/R%C3%A8gles_du_jeu_d%27%C3%A9checs

A.2 Bilbiographie

- LINUX, programmation sytème et réseau, 2ème édition, Joëlle Delacroix, DUNOD
- Qt4 et C++, interface GUI, TrollTech

Interface graphique d'un jeu d'échecs

Département Informatique
4ème année
2007-2008

Rapport du projet d'Unix

Résumé: L'objectif de ce projet d'Unix est d'apprendre à maitriser la programmation système et réseau d'un environnement Unix. Le sujet du projet est la réalisation d'une interface graphique d'un jeu d'échecs. Cela nécessite le développement d'un moteur de jeu utilisant les règles de base et les règles avancées des échecs. Trois types de parties sont disponibles: locale, réseau, ou contre une intelligence artificielle (GNU Chess). Les outils utilisés pour développer ce jeu sont Qt pour la bibliothèque graphique, C++ pour le langage de programmation, QDevelop comme environement de développement et QDesigner pour la réalisation des fenêtres de jeu annexes.

Mots clefs: Unix, Jeu d'échecs, Interface graphique, Commandes système, Qt, GNU Chess, Jeux en réseau

Abstract : The goal of this Unix project, is to learn and control the system and network programmation of an Unix environment. The subject is the creation of a graphic interface for a chess game. It requires the development of a chess motor using chess basic rules and advanced rules. Three kind of games are available : local, network, or artifical intelligence (GNU Chess). The tools we use for the development of this game are Qt, a graphic library, C++, a programmation language, QDevelop, an integrated development environment (IDE) and QDesigner for the creation of games windows.

Keywords : Unix, Chess game, Graphic interface, System commands, Qt, GNU Chess, Network game

---

Étudiants :
Jonathan COURTOIS
jonathan.courtois@etu.univ-tours.fr
Florent RENAULT
florent.renault@etu.univ-tours.fr

Encadrants :
Cédric PESSAN
cedric.pessan@univ-tours.fr
Patrick MARTINEAU
patrick.martineau@univ-tours.fr
Université François-Rabelais, Tours