Miquel Adrien
Lambla Damien

Projet Othello

Sommaire :

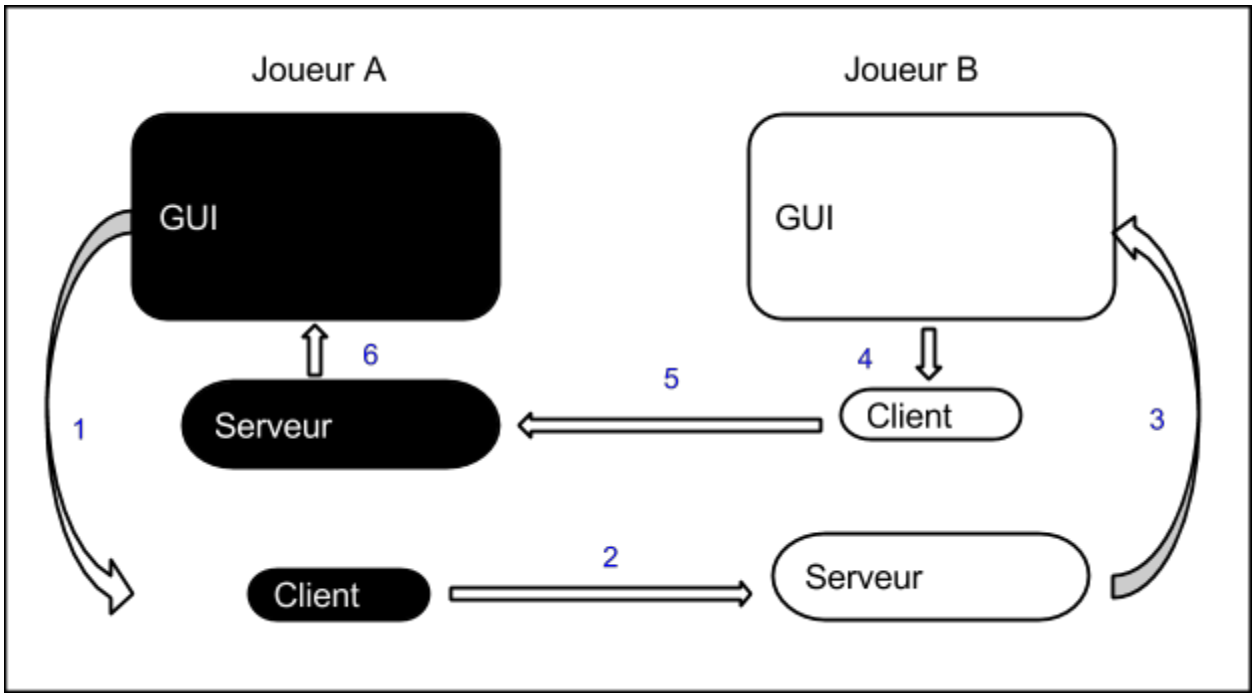
Quelques définitions :

Nous avons choisi de diviser notre code en 3 grandes parties : L'interface graphique (GUI), le serveur, et enfin le client. On retrouve ces trois parties pour tous les joueurs.

- Le serveur ne sert qu'a recevoir les paquets envoyés par le joueur adversaire et les transmettre à l'interface graphique.
- Le client, lui, envoie des requêtes au joueur adversaire.
- L'interface graphique va faire le lien entre client et serveur (en quelque sorte).

Il faut donc bien différencier ce que nous appellerons "joueurs" de ce que nous appelons "clients" et "serveurs".

Pour illustrer cette disposition, voici un schéma explicatif d'échanges de requêtes entre 2 joueurs :



1 : Le joueur A effectue une action, ou requête (demande de connexion, mouvement de jeu, …) et transmet cette action au client.
2 : le client l'envoie au serveur de l'adversaire.
3 : le serveur du joueur B reçoit la requête et la transmet à l'interface graphique qui l'interprète.
4 : Une fois que la GUI a interprété l'action et que le joueur B a donné une réponse (acceptation de connexion, mouvement de jeux, …), la GUI transfert l'action du joueur B vers le client.
5 : le client du joueur B envoie la requête vers le serveur du joueur A.
6 : le serveur du joueur A reçoit la requête et la transmet à l'interface graphique.
etc, etc, ...

Comment lancer Othello :

L'exécution de ce jeu n'est possible que sur une machine disposant d'un système d'exploitation de type UNIX (Ubuntu par exemple), soit en OS principal, soit en machine virtelle.

1. L'arborescence

Tout d'abord, 2 dossiers doivent être créés. Le premier, que l'on nommera *Othello*, contient les fichiers et dossiers suivants (1 dossier = 1 instance de joueur) :

- le dossier **UI_Glade** : il contient les images et les fichiers nécessaires à l'établissement de l'interface graphique
- le fichier **othello_GUI.c** : l'intelligence de l'interface
- **modele_serveur.c** : création et fonctionnement du serveur associé à un joueur
- **modele_client.c** : création et fonctionnement du client associé à un joueur
- le fichier **makefile** : qui se charge de compiler le tout, de supprimer les fichiers "temporaires" (liés au fonctionnement du jeu) et de copier les exécutables dans …

Un deuxième dossier *OthelloBis*, lui ne contiendra que le dossier UI_Glade. D'autres fichiers viendront s'ajouter dans ce dossier, mais aucunes manipulations de votre part n'est nécessaires. Pour augmenter le nombre de joueurs, faites des copies d'*OthelloBis*.

Si vos deux dossiers n'ont pas le même parent, alors il faut copier et compiler les fichiers sources dans le deuxième dossier (étape 3 de la compilation).

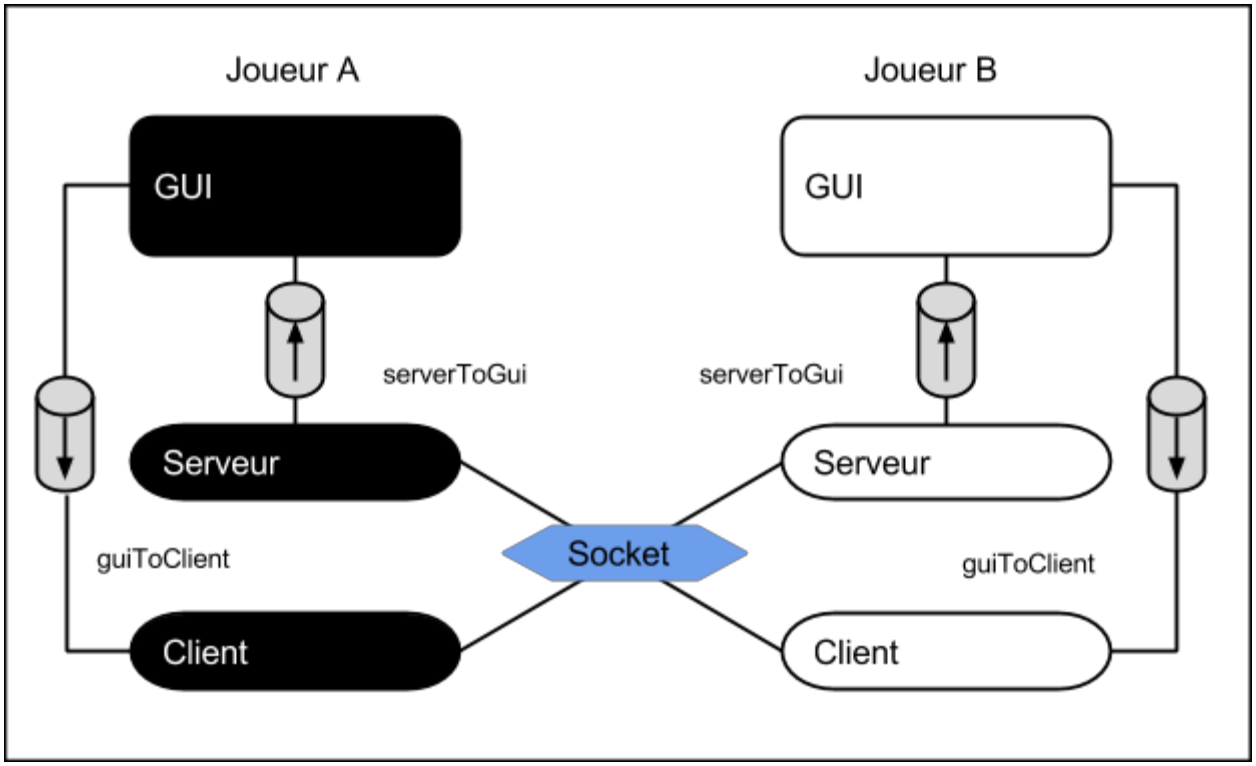
2. La compilation

- Ouvrez un 1er terminal
- Déplacez vous jusqu'à atteindre le dossier *Othello*
- Exécutez l'instruction : `make`
- Exécutez l'instruction : `./othello_gui.o X` ⇒ **l'interface graphique apparaît**
  (où `X` symbolise un numéro de port assez grand pour ne pas être utilisé par un processus système, par exemple : 55555)
- Ouvrez un 2ème terminal
- Déplacez vous jusqu'à atteindre le dossier *OthelloBis*
- Exécutez l'instruction : `./othello_gui.o Y` ⇒ **l'interface graphique apparaît**
  (où `Y` symbolise un numéro de port assez grand pour ne pas être utilisé par un processus système et différent de `X`, par exemple : 55555).
- Ouvrez un 3ème terminal
- Déplacez vous jusqu'à atteindre le dossier *Othello*
- Exécutez l'instruction : `./players_server.o`

3. Quitter Othello :

- Quitter d'abord l'interface des joueurs (avec la croix, pas Ctrl+C),
- puis le serveur de joueurs (Ctrl+C),

Choix d'architecture :



Modèle client-serveur pour chaque joueur (communication entre le client et le serveur : par des pipes nommés et des processus lourds -fork-)

1. Pourquoi avoir choisi des fork plutôt que des threads ?

Nous avons choisi les fork afin de ne pas bloquer l'interface graphique lorsque le serveur est en attente de paquets (recv).
Nous voulions également séparer l'interface graphique des "controller", un peu comme avec le design pattern MVC.
Pour séparer les fichiers sources nous avons utilisé la fonction *execlp*. Cette fonction nous permet également de vider la mémoire que le fork a copié. En effet, nous n'avons plus besoin de la mémoire qu'a copié le fork (cette zone mémoire est une copie de la zone mémoire originale du processus parent : on y retrouve le descripteur de la socket principale, toute l'interface graphique, et bien plus encore…), c'est pour cela que nous lançons un tout nouveau processus avec une toute nouvelle zone mémoire. En faisant cela nous n'avons pas besoin de fermer la socket principale ou quoi que ce soit qui est copié par le fork.

Attention nous utilisons également les threads pour gérer l'ouverture et la lecture des pipes.

2. Structure joueur dans le serveur de joueur :

La structure d'un joueur dans le serveur de joueur est définie de la manière suivante :

```
struct player
{
        char ip[50];
        char port[50];
        char login[50];
        int status;
};
```

On stock donc l'ip du joueur (même si nous ne jouons qu'en local - 127.0.0.1 -), le port du joueur, son login, et enfin son statut. Son statut permet de définir si le joueur est en jeu, en attente d'un adversaire, et potentiellement d'autre statut qui viendraient plus tard.

Pour l'instant le serveur de joueur ne stock que 10 joueurs maximum.
Pour allouer dynamiquement la mémoire en fonction du nombre de joueurs nous aurions pu utiliser une libraire comme glib (http://www.gtk.org/api/2.6/glib/glib-Arrays.html) ou bien implémenter notre propre ArrayList, à l'image de Java : http://stackoverflow.com/questions/3732856/arraylist-in-c

Choix des formats des messages :

Une **en-tête** est toujours présente pour savoir de quel type de message il s'agit. Les messages envoyés par les sockets sont découpés suivant le format CSV. Le séparateur utilisé est donc la virgule.

1. Échange de contenu entre les joueurs (P2P) :

★ Demande de connexion (*othello_GUI.c*) :

En-tête + Port	ex : `"ack,55555"`

```
char message[30];
strcpy(message, "demande,");
strcat(message, portInChar);

// if its a demand send this
if (strcmp(argv[3], "0")==0)
{
	send(sockfd, message, strlen(message), 0);
}
//else send acknoledgment
else if (strcmp(argv[3], "1")==0)
{
	send(sockfd, "ack", 3, 0);
}
```

★ Transmission de coordonnées (coup joué par l'adversaire) ⇒ via socket :

En-tête + Colonne + Ligne	ex : `"c-42"`

```
// we send the movement to the other player
char message[5];
strcpy(message, "c-");

char position[5];
char ligInChar[2];
char colInChar[2];

memset(position, 0, sizeof(position));

sprintf(ligInChar, "%d", lig);
sprintf(colInChar, "%d", col);
```

```
strcat(position, colInChar);
strcat(position, ligInChar);
strcat(message, position);
```

★ Transmission au client via le pipe :

```
write(descGuiToClient, message, strlen(message));
```

★ puis, envoi à l'autre joueur (*modele_client.c*):

```
stringToRead[nbBRead] = '\0';
send(sockfd, stringToRead, strlen(stringToRead), 0);
```

2. Échange de contenu entre le serveur principal et un joueur (client-serveur) :

★ Enregistrement auprès du serveur de joueur, envoi de la requête suivante (*players_server.c*):

En-tête + IP + Port + Login	ex : `"c,127.0.0.1,55555,pseudo"`

```
// we send our ip, port and login
char message[100];
if (strcmp(lecture_addr_serveur(), "")==0 ||
strcmp(lecture_port_serveur(), "")==0 || strcmp(lecture_login(), "")==0)
{
        printf("a field is empty %s, %s, %s,\n", lecture_addr_serveur(),
        lecture_port_serveur(), lecture_login());
        fflush(stdout);
        return NULL;
}

strcpy(message, "c,");
strcat(message, lecture_addr_serveur());
strcat(message, ",");
strcat(message, lecture_port_serveur());
strcat(message, ",");
strcat(message, lecture_login());
strcat(message, ",");
```

★ Transfert au client via le pipe :

```
send(sockfd, message, strlen(message), 0);
```

★ puis, envoi à l'adversaire :

```
stringToRead[nbBRead] = '\0';
send(sockfd, stringToRead, strlen(stringToRead), 0);
```

★ Le serveur envoie ensuite la liste des joueurs de la façon suivante :

En-tête + (IP + Port + Login) * nbDeJoueurs + En-tête

```
char message[100];
strcpy(message, "c,");
int i = 0;
for (i=0;i<next_player_number;i++){
        strcat(message, players_list[i].ip);
        strcat(message, ",");
        strcat(message, players_list[i].port);
        strcat(message, ",");
        strcat(message, players_list[i].login);
        strcat(message, ",");
}
strcat(message, "c");
```

★ Pour signaler un changement d'état d'un joueur (lorsqu'un joueur commence une partie) !! partie non encore implémente !! :

En-tête + PortJA + PortJB

3. Échange de contenu entre inter-processus (pipes) :

Les pipes (ou tubes nommés) sont utilisés pour les communications suivantes :

- serveur → interface graphique (GUI)
- interface graphique → client

Différents types de message sont envoyés, c'est pour cela qu'un système d'en-tête de message a été implémenté. En effet,

les messages commençant par **c-** *sont en rapport avec les déplacements de l'adversaire, ceux commençant par* **j-** *sont concernent les demandes de connexion (acceptation ou refus).*

Le serveur et le client s'occupent respectivement de la transmission des messages à l'interface graphique et de l'envoi des informations au serveur adverse. Ces informations ne sont interprétées qu'une fois que la GUI les ait reçues.

Voici des exemples de code pour l'utilisation des tubes nommés :

★ *Envoi d'un message de l'interface graphique vers le client*

```
char guiToClient[] = "guiToClient.fifo";
descGuiToClient = open(guiToClient, O_WRONLY);
// descGuiToClient est une variable globale,
// descripteur du pipe GUI → client, ouvert en écriture
[...]
char message[5];
strcpy(message, "c-");
[...]
write(descGuiToClient, message, strlen(message));
```

★ *Réception d'un message par le client depuis la GUI*

```
char guiToClient[] = "guiToClient.fifo";
descGuiToClient= open(guiToClient, O_RDONLY);
// descGuiToClient est une variable globale,
// descripteur du pipe GUI → client, ouvert en lecture
[...]
char buf[7];
read(descGuiToClient, stringToRead, buf);
```

★ *Interprétation d'un message par la GUI, reçu du serveur*

```
char stringToRead[5];
int nbBRead;

while(1)
{
	if((nbBRead = read(descServerToGui, stringToRead, 5-1)) == -1)
	{
		[...]
	}
	else if(nbBRead > 0)
	{
		stringToRead[nbBRead] = '\0'; // make the string readable

		// Separate the header from the content
		char* token = strtok (stringToRead,"-");
		char* header = token;
		token = strtok(NULL, stringToRead);
		char* content = token;
		token = strtok(NULL, stringToRead);

		// Message about the other player
		if(strcmp(header, "j") == 0)
```

```
                {
                        if(strcmp(content, "J2") == 0)
                        {
                                [...] // Suggestion to play ? (to be J2)
                        }
                        else if(strcmp(content, "ok") == 0)
                        {
                                [...] // J2 accepts to play ! :)
                        }
                        else if(strcmp(content, "no") == 0)
                        {
                                [...] // J2 refuses to play ... :(
                        }
                }
                // Message about a movement from the opponent
                else if(strcmp(header, "c") == 0){
                        [...]  // handle the movement and allow current
                               // player to play.
                }
        }
}
```

Difficultés rencontrées :

Au début du projet, l'erreur suivante apparaissait en console au lancement de l'exécutable :

*(othello_GUI.o:2788): IBUS-WARNING **: Unable to connect to ibus:* Le délai d'attente est dépassé

Ajouter dans le *.bashrc* :

```
export GTK_IM_MODULE_FILE=/usr/lib/gtk-3.0/3.0.0/immodules.cache
```

Au moment d'implémenter la terminaison d'une partie, il était nécessaire de killer les processus associé à l'exécution du jeu, l'affichage en console donnait :

```
Complété
```

Il ne fallait pas tuer les processus de pid 1 ou 0.

```
(othello_GUI.o:4562): Gdk-CRITICAL **: gdk_window_invalidate_rect_full:
assertion `GDK_IS_WINDOW (window)' failed

(othello_GUI.o:4562): Gdk-CRITICAL **: gdk_window_invalidate_rect_full:
assertion `GDK_IS_WINDOW (window)' failed
```

Il s'agit de problèmes dus à la librairie GTK, il aurait fallu la downgrade ou bien modifier le code en profondeur.

```
[xcb] Unknown sequence number while processing queue
[xcb] Most likely this is a multi-threaded client and XInitThreads has
not been called
[xcb] Aborting, sorry about that.
mktrayicon: xcb_io.c:274: poll_for_event: Assertion
`!xcb_xlib_threads_sequence_lost' failed.
```

Inclure la librairie X11 par l'instruction :

```
#include <X11/Xlib.h>
```

A l'initialisation de GTK dans la fonction `main` de *othello_GUI.c* :

```
/* Initialisation de GTK+ */
XInitThreads();
gtk_init (& argc, & argv);
```

http://stackoverflow.com/questions/18647475/threading-problems-with-gtk
http://stackoverflow.com/questions/24989432/linking-error-dso-missing-from-command-line

Parfois, la lecture des paquets reçus par socket ou pipe était illisible.

"Fermer" le buffer avec un '\0' afin de signaler la fin de la chaîne de charactère :

```
stringToRead[nbBRead] = '\0';
```

Management du projet :

1. Ordre de développement :

1. création de l'architecture multi-processus (fork)
2. connexion des sockets
3. début rédaction du rapport
4. création des pipes
5. connexion des pipes (threads)
6. mise en place d'une stratégie de communication entre pipes et socket (entête des paquets)
7. développement des algorithmes de jeu
8. développement du serveur de joueur
9. solution de stockage des structures joueurs
10.

2. Outils utilisés :

- Trello : un "tasker", cet outil permet de renseigner différentes tâches selon plusieurs catégories et statuts (à définir).
- GitHub : inutile de présenter ce gestionnaire de version très répandu parmi la communauté informatique.
- Slack : Slack est une plateforme pour la communication d'équipe : tout est regroupé en un seul endroit, avec des mises à jour instantanées, peut importe où l'on se trouve. Cet outil nous a permis de centraliser les informations issues de Trello & GitHub.
- Drive : pour la rédaction de ce rapport.

Ce qu'on a rajouté :

1. Texte en gras :

Afin de signaler au joueur que c'est a lui de jouer, nous avons rajouté une mise en gras du nom du joueur dans l'interface graphique.

2. Boîte de dialogue supplémentaire :

Des boîtes de dialogue alertent l'utilisateur sur ses actions, nous vous laissez les découvrir....

Ce qu’on a pas pu finir :

- Signaler le changement d'état d’un joueur auprès du serveur de joueurs.
- L’adresse IP n’est pas pris en compte partout.
- Le code ne compile pas avec l’option -Wall (warnings Gtk)
- Les pipes et les socket ne sont pas fermés correctement à la fin de d’une partie.
- Certaines erreurs ne sont pas gérés lorsque le joueur fait des choses inattendu.
- Lorsqu'une erreur fait planter l’interface graphique, les sockets et pipes ne sont pas bien fermés et supprimés.