Rapport de Projet

Modèle Client/Serveur, TCP/IP, Sockets Unix

Wumpus

COUTABLE Guillaume, RULLIER Noémie
6 avril 2013

UNIVERSITÉ DE NANTES

Table des matières

1 Introduction

L'objectif de ce projet fut de développer une application avec échange de données entres machines distantes. Il devait permettre la compréhension du fonctionnement des interfaces systèmes/-réseaux et des applications clients/serveur.

Afin de créer cette application que nous avons appelé *Wumpus* , nous avons établit plusieurs étapes dans l'avancement du projet. Ce rapport présentera ces étapes les unes après les autres.

2 Présentation de l'application

Notre application est basé sur un jeu : le *Wumpus* .

Le *Wumpus* est un jeu ou un personnage, ici le client, devra parcourir une tour souterraine d'étages en étages à la recherche de trésors. Munie de ses flèches il devra tuer le *Wumpus* pour rester en vie.

2.1 Règles du jeu

2.1.1 Version mono-client

La tour souterraine est composée d'étages. Chaque étage est représenté par une grille qui inclut une échelle, un trésor, un trou hérissé de lance et du *Wumpus* . L'échelle permet au joueur de descendre à l'étage inférieur lorsque le trésor a été trouvé. Au cours de son exploration, si le joueur tombe sur le *Wumpus* ou tombe dans le trou, la partie est terminée.

Le joueur possède une unique flèche qui lui est donnée à chaque nouvel étage. Cette flèche permet au joueur de tuer le *Wumpus* .

Au cours de ces précédentes aventures le personnage à acquis un certain nombre de senseurs. Ces senseurs donne la capacité au personne de ressentir son environnement proche :

WIND : Ce senseur est activé lorsque le personnage se situe sur une case adjacente au trou.

TWINKLE(scintiller) : Ce senseur s'active lorsque le joueur détecte la présence du trésor.

STINK(empester) : Ce senseur est activé lorsque le joueur ressent la présence du *Wumpus* .

Une partie est terminée, si le joueur est tué par le *Wumpus* , s'il tombe dans un trou, ou si le joueur décide qu'il est temps de remonter à la surface. Lorsque la partie est terminée, le joueur se voit attribuer un score en fonction du nombre d'étage parcourus, du nombre de trésor trouvés et du nombre de *Wumpus* tués.

2.1.2 Version multi-client

Au passage de la version mono-client à la version multi-client, certaines règles ont évolué pour instaurer de la compétitivité entre les joueurs :

- Les joueur se déplacent toujours dans le même environnement, mais ne connaissent pas la position des autres joueurs.
- Lorsque un des joueurs de la partie passe change d'étage, les autres joueurs de la même partie passent automatiquement à l'étage suivant.
- Les points acquis à la découverte du trésor ne sont plus attribués au moment ou le joueur trouve le trésor, mais au moment ou il passe à l'étage suivant. De cette manière si deux joueurs trouvent le trésors, seul le plus rapide des deux est récompensé.
- Si un des joueurs meurt, il perd des points en fonction de ce qui l'a tué.
- Si l'un des joueurs d'une partie meurt, il n'a plus la possibilité d'effectuer d'actions. À moins que l'un des joueurs change d'étage , auquel cas la partie continue, ou que tous les autres joueurs de la même partie meurent aussi. Dans se dernier cas on considère que la partie est terminée et que le joueur ayant le plus de points à gagné.

3 Coté serveur

Dans cette partie nous verrons comment sont traitées les commandes, comment est faite la communication client/serveur et comment est effectuées la gestion des clients connectés.

3.1 Commandes clients

Au démarrage, le serveur est mis en attente de connexion de clients. Une fois le joueur connecté, il a la possibilité d'entrer des commandes. Ces commandes correspondent aux actions possibles du joueur.

3.1.1 Descriptions des commandes

À chaque commande coïncide une fonction. Voici une brève explication de chacune de ces fonctions.

Commande quit : La commande *quit*, quand elle est reçu, supprime le joueur qui l'a envoyé de la partie.

Commande move : La commande *move* met à jour la position du joueur en fonction de la direction pointé par celui-ci(NORD, SUD, EST, OUEST). Avec la nouvelle position du joueur, la commande vérifie si le joueur est sur une case à événement, c'est à dire soit sur une case trésor, soit sur une case trou, soit sur une case *Wumpus* . Puis met à jour les senseurs du joueur et envoi les nouvelles informations au client.

Commande turnright : La commande *turnright* met a jour la direction du joueur. La direction évolue dans le sens des aiguilles d'une montre. Puis envoi les nouvelles informations au client.

Commande turnleft : La commande *turnleft* à le même comportement que la commande *turnright*, à la différence que la direction évolue dans le sens inverse des aiguilles d'une montre.

Commande shot : La commande *shot* correspond à l'utilisation de la flèche par le joueur. Cette commande parcours les cases de la position du joueur jusqu'au bord de la carte dans la direction regarder par le joueur. Si le *Wumpus* est rencontré, cela signifie que la flèche à tuer le *Wumpus* , auquel cas la carte du jeu ainsi que les données du joueur(score) sont mis à jour. Les nouvelles données sont ensuite envoyé au client.

Commande down : La commande *down* ne pourra être envoyé par le client que si ce client à trouvé le trésor. Si cette commande est reçu par le serveur, un nouvel étage est généré, le score du joueur qui à envoyé la commande *down* est mis à jour puis la position et la direction de tout les joueurs est remise à zéro. Ainsi les joueurs évoluerons sur une nouvelle carte.

Après cette brève présentation des commandes clients, voyons comment sont traitées ces commandes.

3.1.2 Traitement des commandes

Au démarrage du serveur les commandes sont déclarées en tant que pointeur sur fonction :

```
/*Type T_FONC_ACTION*/
typedef void T_FONC_ACTION();

/*Prototypes fonctions*/
T_FONC_ACTION quit;
T_FONC_ACTION move;
T_FONC_ACTION turn_right;
T_FONC_ACTION turn_left;
T_FONC_ACTION shot;
T_FONC_ACTION down;
```

Un type *Action* est définie pour faire le lien entre la commande reçu par le serveur et la fonction qui correspond à la commande :

```
typedef struct {
    char* command;
    T_FONC_ACTION* action;
} Action;
```

Chaque *Action* est initialisée et conservée dans un tableau :

```
//Number of all commands available
#define NBACTION 6

//Simplify action access
#define A_QUITTER 0
#define A_AVANCER 1
...

//Initialize all available actions
Action* initialisationActions()
{
    Action* actions = (Action*)malloc(NBACTION*sizeof(Action));

    //quit action
    Action* quitAction = (Action*)malloc(sizeof(Action));
    quitAction->command = "quit";
    quitAction->action = quit;
    actions[A_QUITTER] = *quitAction;
    ...

        return actions;
}
```

Après que la commande ai été reçue, on récupère le pointeur de fonction correspondant à la commande à l'aide de la fonction *findActionFromCommand* :

```
Action* findActionFromCommand(Action* action, char* command);
```

Cette fonction parcours le tableau contenant toutes les actions disponibles, test la chaîne de caractères donnée en entrée et retourne l'*Action* correspondante et permet ainsi l'exécution de la fonction adéquat.

3.2 Communication client/serveur

Lors de la première connexion d'un client, le serveur envoi une requête de pseudonyme au client. Une fois ce pseudonyme reçu le serveur instancie un nouveau joueur ainsi qu'un thread. Le serveur affecte alors le joueur au thread ainsi qu'à un jeu(cf.3.3). Au final pour chaque connexion un thread sera créé.
Chaque thread ainsi que le serveur seront en mode "écoute" du réseau. Chaque thread sera en attente d'une commande de son client, et le serveur sera en attente de la connexion d'un nouveau client.

Entrons un peu plus dans les détails. Lorsque le client envoi des données, ces données représentent une simple chaîne de caractère qui correspond à une commande. Nous avons vu dans la section "3.1.2 **Traitement des commandes**" comment le serveur récupérait ces données et ce qu'il faisait pour exécuter la bonne fonction. Par contre lorsque le serveur veut envoyer des données, il envoi les données sous la forme d'une structure :

```
//Different type of structure which could be send
#define STRUCTACQUITEMENT 0
#define STRUCTMESSAGE 1
#define STRUCTMOVING 2
#define STRUCTDOWN 3

//Maximum size of a sending structure
#define TAILLEMAX (TAILLE_MAX_NOM+sizeof(int))*(NBPLAYERSPERGAME)+sizeof(int)

typedef struct
{
    int type;
    char name[15];
    char structure[TAILLEMAX];
} sendToClient;
```

Dans cette structure le paramètre le plus important est le troisième paramètre car il correspond aux données envoyées. Ces données sont encapsulées dans une structure qui dépend du type. Au final le serveur peut envoyer quatre type de structures :

- Une structure d'acquittement
- Une structure qui contient un message

– Une structure contenant des informations relatives au mouvement d'un joueur.
– Une structure utilisée à la réception de la commande *down*

Il est a noter qu'à chaque envoi de données, que ce soit depuis le serveur ou depuis le client, l'émetteur attend une trame d'acquittement(sauf s'il s'agit de la trame d'acquittement elle même).

3.3 Gestion des clients

Pour une meilleur gestion des clients nous avons mis en place une structure qui réparti les joueurs entre les différentes parties en cours. Le comportement de cette structure se base sur l'ajout et la suppression de joueur. Nous verrons donc le comportement de la structure mise en place lors de la connexion d'un joueur, ou lorsque celui si termine la partie sur lequel il joue.

3.3.1 Ajout d'un joueur

Un joueur est ajouté dans la structure lorsqu'il demande une connexion au serveur et renseigne son pseudonyme. Dans ce cas la structure cherche une partie en cours qui respecte les conditions suivantes :

– La partie ne doit pas être complète.
– Pour éviter un déséquilibre au niveau des points, la partie doit être sur le premier étage.

Si une des conditions précédentes n'est pas respectée, la structure créer une nouvelle partie et y ajoute le nouveau joueur.

3.3.2 Suppression d'un joueur

La suppression d'un joueur par la structure survient, soit lorsque le joueur se déconnecte, soit lorsque la partie est terminée (tous les joueurs sont morts ou on quitter la partie). Maintenant que le joueur a été supprimé, on vérifie qu'il reste des joueurs sur la partie. Si ce n'est pas le cas on supprime la partie. Si c'est la cas un autre joueur pourra alors remplacer le joueur supprimé. Le remplacement tiendra compte des conditions nécessaire à l'ajout d'un joueur décrit dans la partie "3.3.1 **Ajout d'un joueur**".

4 Coté client

Pour qu'un client puisque communiquer avec le serveur, il doit tout d'abord commencer par établir une connexion.

4.1 Etablir une connexion

Afin que cette connexion soit établie, le client doit fournir les bons paramètres. Soit l'adresse de la machine hôte sur laquelle le serveur est lancé ainsi que le port. Lorsque la connexion est établie, un thread est lancé. Celui-ci est en fait un listener qui va écouter et recevoir tout ce que le serveur va envoyer au client. On demande au joueur d'entrer un pseudo qui sera envoyé dès que la connexion est établie.

Le client va ensuite pouvoir jouer au *Wumpus* . A chaque action qu'il fera une commande sera envoyée au serveur sous forme de char*. Le serveur va traiter cette commande et envoyer une réponse au client si nécessaire !

5 Jeux d'essais

6 Conclusion