UNIVERSITÉ DE NANTES

Master 1 ALMA

Projet de Réseaux et Protocoles Internet

Bomber Unicorn

Étudiants :
Coraline MARIE et Vincent RAVENEAU

Encadrant :
Pierrick PASSARD

21 mars 2014

Table des matières

Introduction

Dans un monde où les connexions sont de plus en plus présentes, il est indispensable, en tant que futurs ingénieurs en informatique, de connaître et de comprendre les différentes structures de réseaux existantes sur lesquelles notre monde évolue.

Dans le cadre du cours *Réseaux et Protocoles Internet*, nous avions pour objectif la création d'une application réseau de type *client/serveur*. Pour ce faire, nous avons créé un jeu multijoueur inspiré de `Bomberman` avec de nouveaux graphismes : `Bomber Unicorn`.

Ce rapport présente donc la création de `Bomber Unicorn` au travers de nombreuses étapes de travail, ainsi que les différentes fonctionnalités de ce jeu. Pour cela nous verrons d'abord la conception de l'application, puis son développement.

Conception

Cahier des charges

Même si une très grande liberté nous était laissée dans le cadre de ce projet, le sujet nous imposait quelques éléments.

Ainsi, le programme à développer devait posséder une architecture de type *client/serveur*, et être capable de fonctionner sur des machines différentes. La communication nécessaire entre les différents composants du programme devait donc reposer sur des échanges entre sockets.

De plus, la partie communication entre les sockets devait être codée en `langage C`. Cette contrainte avait pour but de nous faire manipuler les sockets à bas niveau, afin de bien comprendre leur fonctionnement.

Deux versions du programme étaient demandées, l'une fonctionnant selon le modèle `1 client/1 serveur`, l'autre selon le modèle `n clients/1 serveur`. Nous avons cependant choisi d'ignorer cette contrainte, comme nous l'expliquerons dans la partie consacrée à nos choix d'implémentation.

Fonctionnalités du logiciel

Comme nous l'avons annoncé dans l'introduction, le programme réalisé dans le cadre de ce projet est inspiré du jeu Bomberman [1]. Bien qu'ayant donné lieu à énormément d'adaptations et de copies, le jeu original est sorti en 1983. Il a été développé par *Hudson Soft* et édité par *Konami*.

Les règles de notre version sont très simples :
- 2 à 4 joueurs s'affrontent sur une carte quadrillée de 9x9 cases, certaines libres, certaines contenant des obstacles.
- Le jeu se déroule au tour par tour, ce qui signifie qu'un joueur doit attendre que tous les autres aient joué avant de pouvoir effectuer une action.
- Lors de son tour, un joueur peut se déplacer vers une case libre adjacente ou poser une bombe.
- Chaque joueur peut avoir 3 bombes posées simultanément sur le terrain.
- Une bombe explose après que le joueur l'ayant posée ait joué 6 tours. Tous les joueurs situés dans le rayon de l'explosion (y compris le joueur ayant posé la bombe) perdent 1 point de vie.

1. http://fr.wikipedia.org/wiki/Bomberman

- Chaque joueur démarre avec 2 points de vie.
- La partie se termine lorsqu'un seul joueur possède au moins 1 point de vie.

Choix d'implémentation

Afin de donner un sens à l'architecture *client/serveur* imposée, nous avons décidé d'effectuer toute la logique de la partie du côté du serveur. Le client a donc pour seul but de récupérer les actions de l'utilisateur, puis d'afficher les différents états du jeu qui lui sont communiqués.

De plus, le serveur a été conçu de façon à pouvoir gérer plusieurs parties simultanées, comme on peut s'y attendre dans le cadre d'un jeu en réseau.

Initialement prévu pour être joué en temps réel (tous les joueurs effectuant leurs actions en même temps), le protocole UDP a été choisi pour la communication *client/serveur*.

Ce choix a été fait de part la nature du protocole, qui offre une communication en mode déconnecté. Ce type de connexion permet d'avoir une meilleur vitesse d'échange de messages qu'avec une connection en mode connecté telle que présente avec le protocole TCP. En revanche, aucune garantie n'est présente concernant la façon dont les différents messages seront délivrés (ordre, intégrité, ...).

Ce manque de garantie ne représente cependant pas vraiment un handicap pour un jeu temps réel. En effet, si une action d'un joueur est ponctuellement perdue, les conséquences sur la partie ne sont pas graves.

Cependant, faute de temps pour mettre un tel système en place, nous avons dû modifier nos plans et passer à un jeu en tour par tour. La perte d'une action devient dans cette situation beaucoup plus grave, puisqu'elle bloque complètement la partie.

Dans ces conditions, la communication via le protocole TCP serait donc plus adaptée, mais le temps nous a manqué pour effectuer les changements nécessaires.

Compte tenu de la nature de notre application, nous avons également décidé de ne pas réaliser de version 1 client/1 serveur.

En effet, cette version aurait nécessité la mise au point d'une intelligence artificielle pouvant servir d'adversaire. Lui faire faire une action aléatoire à chaque tour n'est pas long à mettre en place, mais n'aurait pas d'intérêt. En effet, il aurait toutes les chances de se suicider en faisant des allers-retours sur ses propres bombes, entraînant sa mort avant même que le joueur ne l'atteigne.

Ne disposant pas du temps nécessaire à l'élaboration d'un ennemi plus performant, nous nous sommes donc concentré sur la version n clients/1 serveur.

Dépendances externes

En plus de la bibliothèque standard `C`, nous avons décidé d'utiliser la `CSFML` pour mettre en place l'affichage du jeu et la communication entre les sockets.

La CSFML est le binding C officiel de la SFML, une bibliothèque destinée au langage C++ conçue pour faciliter la création d'applications multimédia. La version que nous avons choisi d'utiliser est la 1.6. Ce choix s'explique par sa présence dans les dépôts Ubuntu (système linux utilisé pour développer le logiciel), ce qui facilite son installation.

Interface graphique

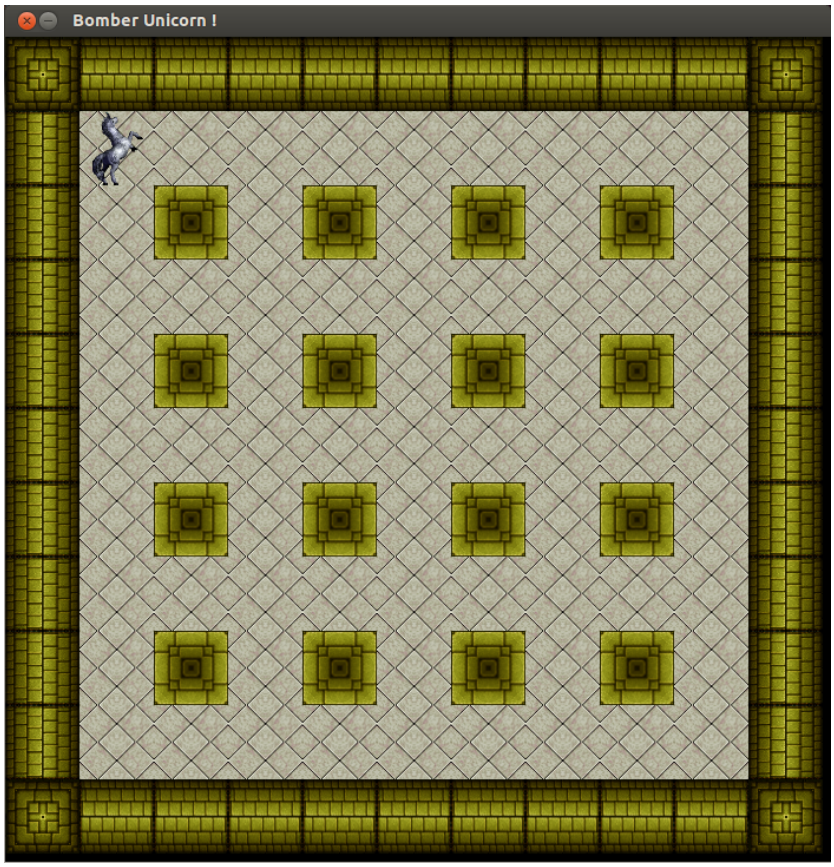
Dans le but d'offrir à l'utilisateur principal (le client) un rendu clair et agréable, nous avons opté pour une interface graphique plutôt qu'un affichage dans une console.

Celle-ci a été entièrement construite avec la CSFML, et est composée d'un ensemble d'éléments graphiques, appelés *sprites* placés sur une grille.

Nous avons créé l'intégralité de nos sprites nous même à l'aide d'une base de donnée d'images libres et d'un logiciel infographique.

Voici quelques exemples de sprites :

Voici la carte créée avec les sprites :


Bomber Unicorn !

Fonctionnement

Les clients et le serveur communiquent en s'envoyant divers messages. Ces messages concernent l'état des joueurs (position, points de vie, nombre de bombes, ...) ou des informations sur la partie (démarrage, début du tour d'un joueur, ...). Ces messages prennent la forme de tableaux de caractères et sont interprétés par le client ou le serveur.

Serveur

Fonctionnement général

L'exécution du serveur prend deux paramètres, le nombre de parties simultanées maximum autorisé (entre 1 et 100), et le nombre de joueurs par partie (entre 2 et 4).

Une fois lancé, il réserve la place mémoire nécessaire pour le nombre de parties demandé, puis écoute les demandes de connexion des clients sur le port 5000.

Lors de la réception d'un message sur ce port, il cherche à déterminer si une partie est capable d'accueillir un joueur supplémentaire.

Pour ce faire, il parcourt la liste des parties, à la recherche d'une n'ayant pas assez de joueurs pour démarrer. Dès qu'il trouve une partie dans cet état, il arrête la recherche et ajoute un joueur à la partie. Il envoie ensuite au client un message contenant le numéro de la partie en question et le numéro de joueur du client au sein de cette partie.

Si aucune partie ne manque de joueurs, et que le nombre maximal de parties n'est pas atteint, le serveur crée une nouvelle partie. Il y ajoute ensuite un joueur, et envoie au client un message de la forme décrite précédemment.

Enfin, si aucune partie ne manque de joueurs et que le nombre maximal de parties simultanées est atteint, le serveur envoie au client un message indiquant qu'aucune partie n'est disponible.

Gestion d'une partie

Afin de ne pas créer de conflit et pour éviter les blocages ou arrêts du serveur, chaque partie est gérée dasn un thread qui lui est propre. Ainsi, dans le cas d'un problème tel que le blocage de la partie, le serveur est toujours en mesure de gérer l'arrivée de nouveaux clients.

La mise en place d'un tel mécanisme a cependant nécessité de pouvoir contrôler les accès concurrents aux parties lors du parcours visant à déterminer si un

nouveau joueur peut être accepté ou non. Pour ce faire, nous avons utilisé les mutex tels qu'implémentés par la `CSFML`.

À sa création, une partie écoute les messages envoyés par ses joueurs sur un port qui lui est propre. Le numéro de ce port est déterminé selon la formule suivante : $5100 + numeroPartie * (joueursParPartie + 1)$, ce qui donne par exemple 5100, 5105 et 5110 pour 3 parties de 4 joueurs.

Une fois tous les joueurs présents, un message leur est envoyé pour indiquer le début de la partie, et le premier joueur reçoit également le message indiquant qu'il doit effectuer une action.

La partie exécute alors la boucle suivante, jusqu'à ce qu'il ne reste qu'un seul joueur en vie :

- Attente de l'action du client du joueur courant.
- Vérification de la validité de l'action du client. Si celle ci n'est pas valide, un message lui est envoyé demandant de refaire une action. Si celle ci est valide, on passe à l'étape suivante.
- Réalisation de l'action (déplacement ou pose d'une bombe).
- Avancement de l'état des bombes posées par le joueur courant. Si l'une d'elles explose, les dégâts infligés sont calculés.
- Envoi à tous les joueurs du nouvel état du jeu.
- Indication au joueur suivant que c'est à lui de jouer.

Une fois la partie finie, un message est envoyé aux joueurs pour le leur indiquer. La partie est ensuite réinitialisée, et prête à accueillir de nouveaux joueurs.

Client

Avant de lancer les divers clients, il faut s'assurer que le serveur tourne déjà. Lorsque un client est exécuté, il doit avoir en paramètres : une adresse (l'adresse ip du serveur) et une chaîne de caractères (le nom du joueur). Grâce à ces paramètres, le client demande une connexion au serveur qui, s'il n'est pas surchargé, répond sur le port 5001 en lui renvoyant le numéro de la partie qui lui a été attribuée et son numéro de joueur. Si le serveur est surchargé, il répond qu'aucune partie n'est disponible.

Déroulement d'une partie

Une fois la connexion acceptée et le client ajouté à une partie, les messages du serveur sont reçus sur un port dépendant du numéro de partie et du numéro du joueur dans cette partie, calculé de la façon suivante : $5100 + numeroPartie * (joueursMaxParPartie + 1) + numeroDeJoueur$. Ceci permet de lancer autant de client que possibles sur une seule machine sans avoir pour autant de problème pour envoyer le bon message au bon client.

Lorsque le client est connecté au serveur et qu'il possède un numéro de partie et un numéro de joueur, alors la partie peut commencer. Le joueur dispose de deux états, mis à jour par le serveur : l'un lorsque c'est à son tour de jouer, et

l'autre quand ça ne l'est pas.

Si c'est au tour du client de jouer, alors celui-ci récupère une action de l'utilisateur et l'envoie au serveur. Le joueur ne peut faire que deux actions différentes, et une seule par tour : se déplacer ou poser une bombe. Lorsque le serveur à reçu l'action du joueur par le client, il doit alors lui renvoyer un message qui peut être :

- Soit un message disant que l'action de l'utilisateur est réfusée (déplacement incorrect ou impossibilité de poser une bombe). Dans ce cas le client récupère une nouvelle action de l'utilisateur, la renvoie au serveur et reattend un nouveau message du serveur.
- Soit un message avec les nouvelles coordonnées des joueurs et des bombes, qui signifie que l'action à été acceptée, que le client doit mettre à jour l'interface graphique, et que ce n'est plus à son tour de jouer.

Si ce n'est pas au tour du client de jouer, alors celui-ci peut recevoir deux messages différents du serveur :

- Soit un message lui annonçant que tous les joueurs ont fini de jouer et que c'est à son tour.
- Soit un message avec les nouvelles coordonnées des joueurs et des bombes pour que le client mette à jour l'interface graphique.

Ces étapes sont répétées en boucle jusqu'au message du serveur indiquant que la partie est terminée.

Le client détermine alors, selon ses points de vie restants, s'il a gagné ou non.

Mise à jour graphique et fin de partie

A chaque fois que le client reçoit un message du serveur contenant des informations sur les autres joueurs, il met à jour l'interface graphique. Pour cela il fait d'abord des tests sur les données des autres joueurs afin de déterminer :

- la position actuelle de tous les joueurs.
- la direction du regard des joueurs en fonction de leur mouvement.
- l'emplacement des bombes.
- l'état des bombes : juste posée, en décompte ou en train d'exploser.
- les points de vie restant aux joueurs et les joueurs mort.
- le statut de la partie : si il y a un vainqueur ou non.

Ensuite, lorsque toutes les données ont été analysée, le client choisit les sprites correspondantes et les places dans la fenêtre. Cette dernière est ensuite actualisée et affichée sur l'écran du joueur qui a exécuté le client.

Conclusion

Nous avons présenté, tout au long de ce rapport, notre application `Bomber Unicorn`. Malgré la simplicité des règles de ce jeu, le projet fut relativement difficile à réaliser. En effet, nous avons utilisé les nouvelles notions de communications réseaux étudiées en cours, dans un langage de programmation bas niveau, ce qui complexifie les choses. De plus, l'utilisation de plusieurs processus (multithreading) sur des ordinateurs peu puissants nous a par moment causé des problèmes de performances et d'efficacité.

Malgré ces difficultés, nous avons réussi à mener le projet dans un état fonctionnel, mais il reste cependant quelques points que nous aurions souhaité améliorer, si nous en avions eu le temps. En effet, l'ajout d'informations dans l'interface graphique du client serait un gain de clarté pour l'utilisateur (description des tours de jeu, nombre de points de vie restants, etc...).

De plus, certains problèmes restent à corriger, comme l'absence de timeout pour gérer le cas où un client ne répondrait plus en cours de jeu (ce qui bloque actuellement une partie). Des erreurs de segmentation surviennent également aléatoirement côté serveur lors de la connexion du premier client au serveur, probablement dûes au lancement d'un thread. Ces erreurs semblent cependant être beaucoup moins présentes si le serveur est exécuté via un débuger tel que gdb.

`Bomber Unicorn` nous a permis d'utiliser de nouvelles connaissances étudiées en *Réseaux et protocoles Internet*, mais pas seulement. Nous nous sommes servis pour ce projet d'autres notions vues dans d'autres modules d'enseignement.