Projet d'année - Lombric à Brac

Ouriaghli Ismaël Dupuis Emma Renard Simon
Grimau Florent Stevens Quentin

1er mars 2020

Table des matières

Préface

Avant de commencer notre récit épique, nous tenons à remercier feu nos camarades qui nous ont quitté trop tôt durant cette aventure extraordinaire. Ils n'ont malheureusement pas survécu au péripéties sans relâche qui nous ont guettés à chaque *commit* lors de l'avancement du projet.

Alexis, notre cher et tendre menu designer sans qui cette aventure serait restée une simple idée.

Vassilena, pleine de motivation mais surchargée de travail. Courage pour ses autres projets.

Diego, sa présence aux réunions nous était chère, son humour et surtout, sa volonté de se battre.

1 Introduction

1.1 But du projet

L'objectif de ce projet est de créer un jeu d'artillerie et de stratégie multijoueur en réseau du nom de "Lombric à brac", où chaque utilisateur devra se connecter au jeu afin de pouvoir créer ou bien rejoindre une partie. Il aura aussi la possibilité d'être spectateur d'une partie en cours et pourra envoyer des messages depuis le jeu.

Avant de jouer les joueurs pourront nommer leurs lombrics ainsi que leur équipe, l'hôte de la partie pourra également s'occuper de quelques paramètres tels que le mode de jeu (match à mort, match à mort par équipe) ou également le nombre de lombrics par joueur (8 maximum).

Une fois la partie lancée chaque lombric sera dispersé aléatoirement sur le terrain, la partie pourra ensuite commencer. Les joueurs joueront chacun à leur tour et auront la possibilité de faire différentes actions, ils pourront déplacer leur lombric sélectionné ainsi qu'utiliser un outil leur permettant d'attaquer un autre lombric ou de se repositionner.

Pour qu'une partie se finisse, il faut qu'il ne reste que les lombrics d'une seule équipe sur le terrain, ce qui fera gagner cette dernière, ou bien que tous les lombrics soient morts, mettant fin à la partie par une égalité.

Toutes ressemblance au jeux

1.2 Glossaire

pseudo : Pseudonyme

Macro : nom donné à une valeur pour faciliter la lecture du code

bug : bogue

spawn : apparition

print : affichage en terminal d'un caractère unicode

Sourire : important putain !

Sprite : Objet destiné à être affiché

1.3 Historique du document

Version	Auteur	Date	Description
0.1	Simon R., Emma D.	25/11/19	Besoin utilisateur
0.2	Alexis M., Ismaël O., Emma D., Simon R., Florent G., Quentin S., Vassilena S., Diego B.A.	26/11/19	Création du squelette
0.3	Alexis M., Ismaël O.	30/11/19	Besoins du système
0.4	Florent G., Quentin S.	08/12/19	Modules
0.5	Diego B.A	08/12/19	But du projet

2 Besoin utilisateur

2.1 Exigences fonctionnelles

Des diagrammes UML sont utilisés pour décrire les différentes exigences fonctionnelles de l'utilisateur. Ces exigences sont séparées en 3 diagrammes distincts. Ces 3 diagrammes représentent respectivement les interaction d'un utilisateur dans le menu principal, dans le salon d'attente et dans une partie.

Menu principal

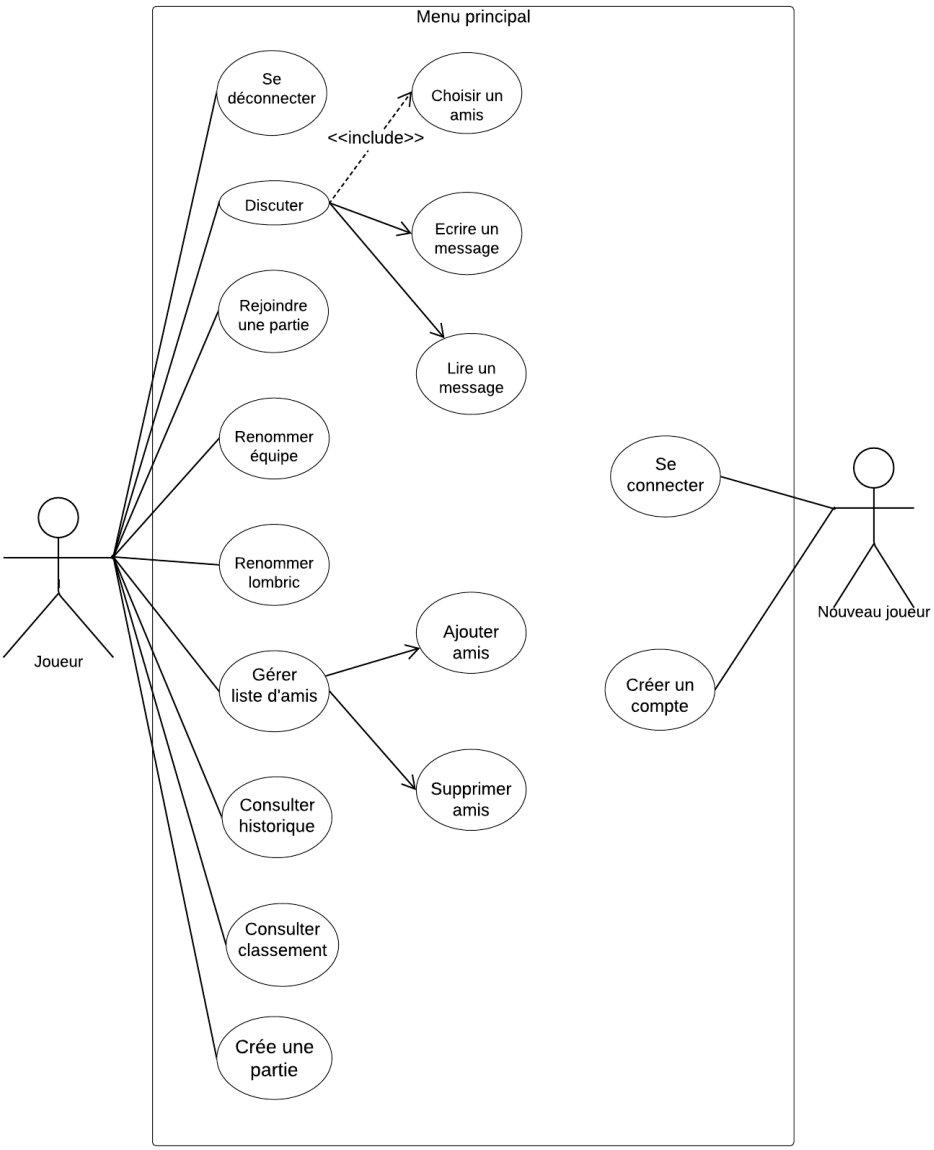

FIGURE 1 – Menu principal

Toutes les actions autre que se connecter et créer un compte impliquent que l'utilisateur est connecté. Nous considérons une personne comme étant un joueur

lorsque que celle-ci se connecte. Elle a donc accès aux fonctionnalités du jeu.

	Pré-conditions	Post-conditions	Cas général	Cas exceptionnels
Se connecter	— Avoir un compte — Être déconnecté	Le joueur est connecté	L'utilisateur rentre un nom et un mot de passe corrects	— Le joueur n'est pas connecté au serveur — Le nom ou le mot de passe est incorrect
Se déconnecter	Être connecté	— L'utilisateur est déconnecté — L'utilisateur est ramené à la fenêtre de connexion	L'utilisateur se déconnecte	L'utilisateur ferme la fenêtre sans se déconnecter
Créer un compte	L'utilisateur n'est pas connecté	Le joueur est connecté sur son nouveau compte	Le nom d'utilisateur n'existe pas encore	Le nom d'utilisateur est déjà pris
Ajouter amis	L'utilisateur n'a pas encore en ami la cible	L'utilisateur est ami avec la cible	Le nom cible est un nom d'utilisateur existant	Le nom de l'utilisateur cible n'existe pas
Supprimer un ami	L'utilisateur a choisi un de ses amis	L'utilisateur n'est plus ami avec la cible	L'ami est supprimé de la liste de l'utilisateur	Pas d'amis à supprimer
Renommer l'équipe	None	L'équipe a un nouveau nom	L'équipe de l'utilisateur est renommée	None
Renommer un lombric	— Un lombric a été choisi — Un nom a été choisi	None	Le lombric choisi a été renommé	Le lombric n'avait pas encore été nommé
Consulter historique	Un joueur est sélectionné	None	L'historique des parties est affiché	Le joueur sélectionné n'existe pas
Consulter le classement	None	None	Le classement est affiché	None
Recevoir un message	None	None	Le message est affiché	None
Envoyer un message	— Un ami est choisi pour recevoir le message — Un message est écrit	None	Le message est envoyé	None

FIGURE 2 – Tableau du use case du menu principal

Salon d'attente

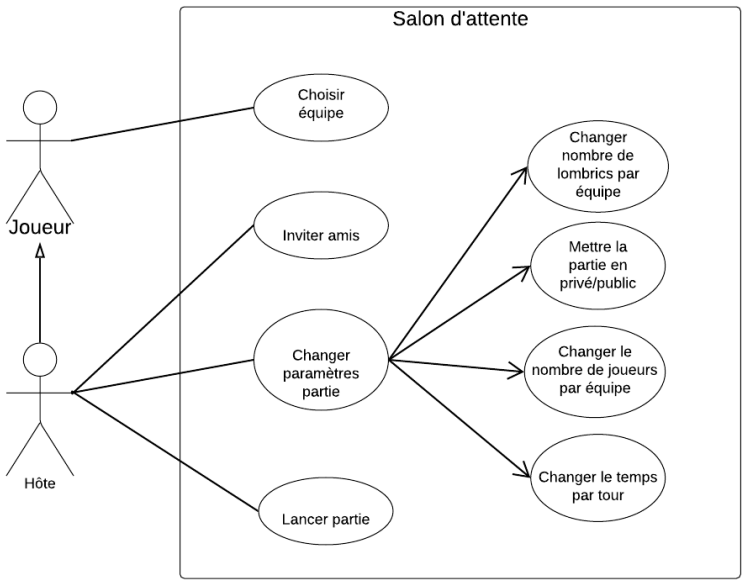

FIGURE 3 – Salon d'attente

Toutes les actions impliquent que la partie n'a pas encore été lancée.

	Pré-conditions	Post-conditions	Cas général	Cas exceptionnels
Choisir une équipe	Équipe pas encore formée	Équipe formée des lombrics choisis	Joueur choisi ses lombrics	None
Changer les paramètres d'une partie	None	La partie est configurée	Le joueur choisit les paramètres désirés	None
Inviter des amis	Avoir des amis	Amis à rejoint la salle d'attente	Le joueur veut jouer avec un ami	Le joueur ami n'existe pas
Lancer la partie	— Les paramètres de la partie sont configurés — Le joueur à créé son équipe — Au moins deux joueurs ont rejoint la partie	La partie est lancée	Le joueur configure la partie et veut commencer à jouer	Il n'y a pas assez de joueurs pour lancer une partie

FIGURE 4 – Tableau du use case du salon d'attente

En partie

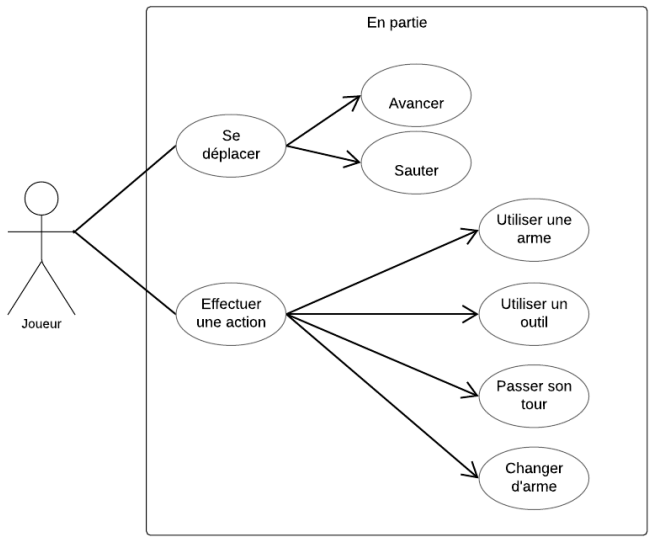

FIGURE 5 – En partie

Toutes les actions autre que discuter s'effectuent uniquement lors du tour du joueur.

	Pré-conditions	Post-conditions	Cas général	Cas exceptionnels
Se déplacer	Déplacement choisit est accessible	— Le joueur se trouve à une autre position — Le tour du joueur est fini	Joueur veut se déplacer sur le terrain	Si le joueur sort du terrain, il meurt
Utiliser une arme	— Avoir un lombric capable d'attaquer — Avoir au moins une arme	— Perte de point de vie de l'adversaire attaqué — Le tour du joueur est fini	Le joueur veut infliger des dégâts à l'adversaire	None
Utiliser un outil	— Avoir au moins un outil	Le tour du joueur est fini	L'outil est utilisé	None
Passer son tour	None	Le tour du joueur est fini	Joueur à fini son tour	None

FIGURE 6 – Tableau du use case d'une partie en cours

2.2 Exigences non fonctionnelles

Le langage imposé pour ce projet est le c++. Le jeu doit au minimum tourner sur les ordinateurs de l'ULB (sous Linux).

En cas de perte de connexion d'un des joueurs, il faut faire en sorte que le cas soit traité étant donné que d'autres joueurs attendent ce-dernier. Lorsqu'un joueur se déconnecte dans le salon d'attente, celui-ci est supprimé du salon. Lorsqu'un joueur se déconnecte en partie, celui-ci perds automatiquement la partie.

2.3 Exigence du domaine

3 Besoin du système

3.1 Exigences fonctionnelles

Création de compte et gestion de compte

Au lancement du jeu,le programme propose à l'utilisateur de soit créer un compte ou de se connecter a un compte déjà existant, si l'utilisateur décide de créer un compte alors le programme envoie un formulaire d'inscription demandant le nom du nouveau compte (pseudo) ainsi que le mot de passe qui lui sera associé, une fois son compte créé l'utilisateur n'aura qu'a rentrer son pseudo et son mot de passe afin de se connecter.

Création de son équipe de lombrics

Lors de la première connexion, le nouveau joueur doit donner un nom à son équipe de huit lombrics.

Jouer une partie

Création d'une partie et configuration de la partie : Le joueur crée une partie en choisissant le mode de jeu qui lui convient ainsi que les différents paramètres modifiables.

Gestion des invitations : Le joueur peut aussi inviter des personnes faisant partie de sa liste d'amis afin de jouer ensemble.

Trouver une partie : Le joueur va rejoindre aléatoirement une partie qui est en attente de joueur afin que, si le nombre de joueur max est atteint, la partie se lance.

Gestion liste d'amis

Le joueur peut envoyer une demande d'ami à un autre joueur, si l'autre joueur l'accepte, ils deviennent amis et pourront dès lors s'envoyer des invitations. Le joueur peut consulter sa liste d'amis.

Gestion discussion des amis

Si deux joueurs sont amis, ils peuvent discuter ensemble.

Consultation de l'historique des parties d'un joueur

Un joueur peut consulter l'historique d'un autre joueur ou le sien ainsi que son classement dans chacune des parties effectuées.

consultation du classement des joueurs

Un joueur peut consulter une liste des différents joueurs classés selon une mesure (par exemple, le nombre de victoire).

3.2 Exigences non fonctionnelles

Besoin d'une connexion internet

Comme le jeu se joue en réseau, une bonne connexion internet est requise afin de communiquer avec le serveur.

Mise à jour de la base de donnée

Le programme doit pouvoir enregistrer des (nouvelles) données comme le profil de l'utilisateur (son mot de passe, pseudo) , son nom d'équipe , sa liste d'amis ou encore les noms de ses lombrics.

Gestion des erreurs

Quand le programme rencontrera une erreur, il ne s'arrêtera pas brusquement et gérera l'erreur en question.

Fluidité

Le programme doit être optimisé afin que pendant une partie il n'y ai que peu de latence et que la connexion entre l'utilisateur et le serveur ne soit pas lente. Le client et le serveur font chacun les calculs d'affichage pour que le jeux soit le plus fluide possible. Une fois un tour terminé, le serveur impose sa vision de la partie à tous le monde pour éviter les problèmes de désynchronisation.

Optimisation

Il faut veiller a ce que le programme(serveur) ne soit pas trop gourmand en ressource (éviter toute fuite mémoire) pour que l'on puisse accueillir le plus grand nombre de joueur possible

Anti-cheat

Le serveur doit faire les calculs considéré important(trajectoire,dégâts) pour évité toute triche potentiel

Sécurité

le programme doit hasher le mot de passe avant de les rentré dans la base de données pour éviter que si une personne tierce arrive avoir accès a notre serveur(machine) , il ait accès a tout les mot de passe en clair qui est un soucis pas que pour notre jeu (si mot de passe utilisé ailleurs)

4 Communication Client/Serveur

4.1 Protocole

En utilisant la librairie protocol-buffers, l'envoi de donnée est simplifié. Nous avons créé plusieurs prototypes, correspondant chacun à des actions particulières. Chacun de ces prototypes est associé à une macro, qui est utilisée pour spécifier quel prototype est envoyé par le réseau, et ainsi pouvoir le récupérer correctement de l'autre côté.

5 Modules

5.1 Module Client

Dépendance

Pour l'affichage sur le terminal, nous utilisons la librairie *ncurses*. Cette librairie nous permet, entre autre, de gérer plus facilement les inputs de l'utilisateur. Pour l'affichage sur le terminal, *ncurses* fait juste une succession de print. On peut définir où faire ces prints, par exemple en bougeant le curseur ou créer une fenêtre.

Fonctionnement des menus

Pour naviguer entre les différents menus nous utilisons une variable *res* qui contiendra l'*id* de la fenêtre à ouvrir. Pour ce faire, nous avons prédéfini une liste des différentes fenêtres avec leur *id* respectif et chaque fenêtre, retourne l'*id* de la prochaine fenêtre à afficher. Donc le main est juste une boucle infini qui appelle la bonne fenêtre (le bon objet) en fonction de ce que contient *res*.

Menu entrer

Elle correspond à la première fenêtre auquel l'utilisateur sera confronté. Il aura trois choix devant lui :

- Se connecter
- S'enregistrer
- Quitter

Il pourra donc en effet se connecter ou se créer un compte. Si il y a un quelconque problème avec le pseudo ou le mot de passe entré, une fenêtre avec un message d'erreur approprié s'affichera et le redirigera sur le Menu Entrer.

Menu Jeu

Ici, l'utilisateur est dans le grand menu du jeu. Il aura plusieurs choix qui lui seront proposés (voir ci-dessous) et pourra effectuer différentes actions, tel que se déplacer dans le menu et dès qu'il appuiera sur la touche de confirmation,

une nouvelle fenêtre s'affichera avec les bons éléments.

— Trouver partie
— Créer une partie
— Créer/Modifier équipe
— Voir ses invitations
— Classement
— Amis
— Historique
— Se déconnecter

Créer une partie

En appuyant sur créer une partie dans le menu jeu, l'utilisateur devra alors alors entrer les différents paramètres de la partie dans la nouvelle fenêtre qui s'est affichée. Des messages d'erreur seront afficher si il ne respecte pas les restrictions imposées (par exemple si il donne en input un *string* au lieu d'un *int*, ou bien si le *int* entré dépasse la limite autorisé).
À savoir, pour pouvoir récuperer l'entré utilisateur dans la *box* nous utilisons la fonction *mvwgetnstr()* de *ncurses* qui enregistre les entrées dans le tableau de char. Après avoir réceptionné le message, la fonction *stoi()* pour transformer les entrées en *int* pour pouvoir testé la valeur entré. Un *catch* de *std : :std invalid argument* est effectué après le *stoi()* (dans le cas où un *string* est entré au lieu d'un *int*).
Après cela, l'utilisateur aura officiellement créer une *room* et se trouvera dans un salon d'attente où il verra les autres utilisateurs connectés à sa *room*. Dans le salon d'attente il peut soit :

— Annuler
— Options (changer les paramètres de la partie)
— Inviter amis (dans le salon)
— Lancer la partie

Invitation et Ami

À plusieurs endroit, l'utilisateur peut interagir avec les invitations reçues et ses amis/son ami.
Dans un premier temps, dans le Menu Jeu en appuyant sur *Voir ses invitations* il pourra voir les invitations reçues (une distinction est faite entre invitation de jeu et une demande d'ami). Il peut soit accepter l'invitation (si il s'agit d'une invitation de jeu, il est alors redirigé dans le salon d'attente), soit renier l'invitation ou ne rien faire du tout.
Dans un second temps, dans l'onglet *Ami* (à savoir qu'il y aura une notification si l'utilisateur a un message non lu qui, si oui une asterix s'affichera *Ami (*)*), l'utilisateur aura le choix entre soit :

— Envoyer demande ami
— Consulter liste amis
— Discuter avec ses amis
— Revenir au menu

Si il veut envoyer une demande d'ami, il doit écrire dans la *box* indiqué le pseudo de la personne à qui envoyer l'invitation. La deuxième option est de consulter sa liste, cela veut dire qu'il pourra voir ses amis et/ou supprimer si il le souhaite un ami(à noter qu'il faut au préalable mettre la flèche devant l'ami à supprimer en appuyant sur q). Et enfin la dernière option ,mais non des moindres, le tchat. Dans cette option, il pourra choisir un ami à qui envoyer un message, cette étape effectuer, une fenêtre s'affichera avec les 5 derniers messages échangés et une box où sera écrit le message. De plus en appuyant sur r l'utilisateur pourra *refresh* la fenêtre pour voir si possiblement des nouveaux messages ont été reçu.

5.2 Affichage de la partie

Gestion de la partie

Durant la partie, le joueur peut déplacer la caméra si la console est plus petite que la taille de la carte. Quand c'est à son tour, la fenêtre qui contient les informations de la partie change pour voir quelle arme est sélectionnée et comment elle est paramétrée. Il voit également le temps restant du tour.

Fonctionnalité

Chaque sprite peut subir un mouvement, soit un mouvement parabolique, soit un mouvement rectiligne uniforme. Les explosions font des dégâts circulaire sur la carte et expulse les lombrics touchés par l'explosion. Quand un lombric est soumis à un mouvement, il ne peut pas être contrôlé par le joueur.

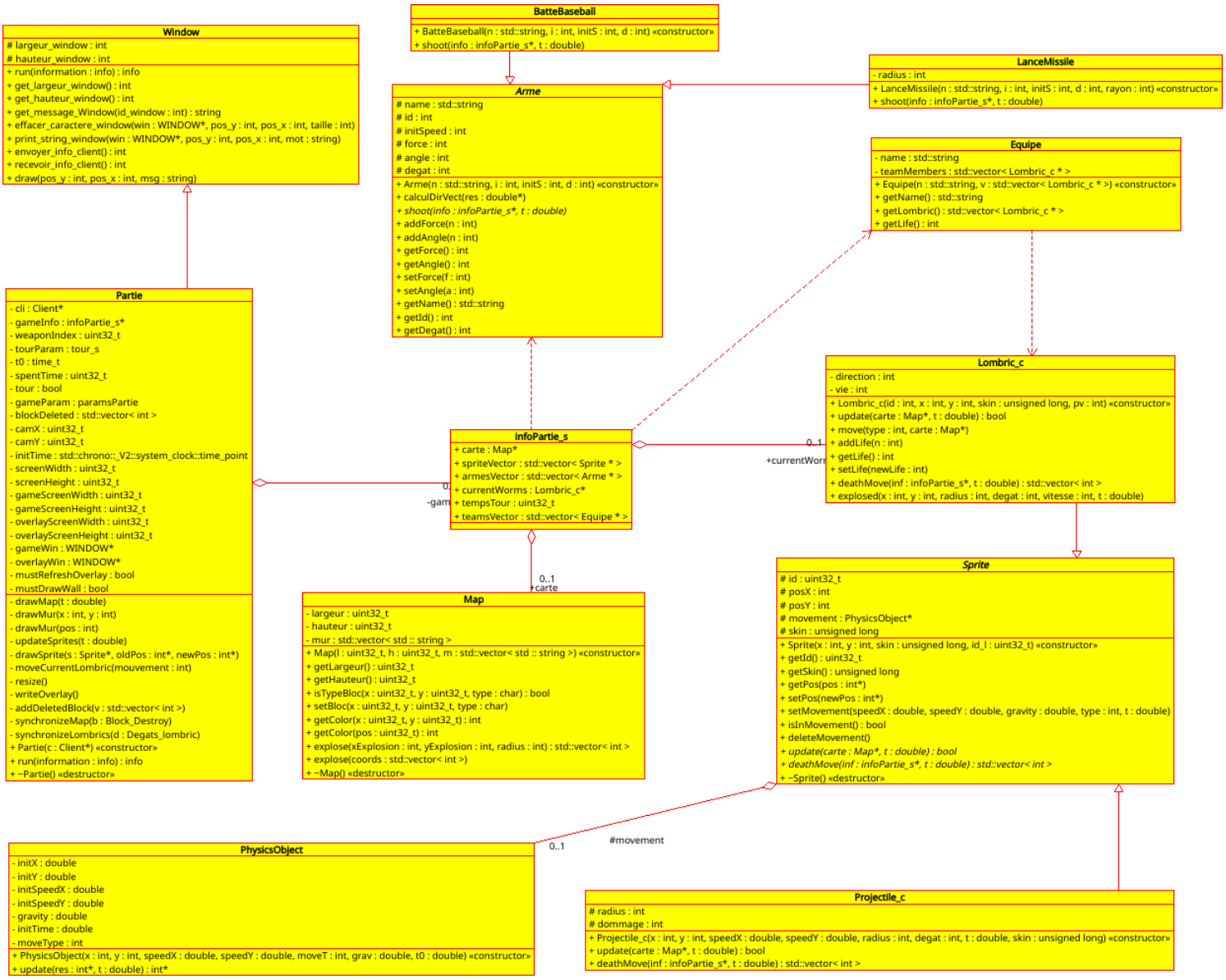

FIGURE 7 – Diagramme de classe de la partie

5.3 Communication avec le serveur

Le module client s'occupe de faire la passerelle entre l'affichage et le serveur. Il reçois les messages du serveur et créé à partir de ces messages des structures adéquates pour l'affichage.

Thread

Le module client est un thread secondaire.

Soit le client réceptionne des messages dits inattendus (que l'affichage n'a pas expressément demandé) et les stocke dans des vecteurs jusqu'à ce que l'affichage demande au client s'il y a des nouveaux messages à afficher.
L'affichage récupéra ce vecteur et le client pourra donc le vider. C'est le cas des messages du chat, des invitations d'amis, des invitations à des parties.

Soit le client réceptionne des messages qu'il a demandé. L'affichage fait donc appel a une méthode du client qui va interroger le serveur puis répondre à l'affichage.
Lorsque l'affichage appelle une de ces méthode, il passe en paramètres les informations dont le serveur a besoin pour traiter la demande. Il s'agit des valeurs reçues en input et d'une macro indiquant au serveur de quelle type de demande il s'agit.
Le client convertira alors une structure contenant ces informations en string et les enverra au serveur.
S'il s'attend à une réponse, il se bloquera jusqu'à la réception de cette réponse. Une fois débloqué, il reconvertira le string reçu avec les informations demandées par l'affichage en un objet proto-buff qu'il transformera ensuite en une structure utilisable par l'affichage.
S'il n'attends pas de réponse, il se bloquera juste le temps de l'envoi du message. C'est le cas quand un tire est effectué, l'affichage envoie les paramètres du tir mais n'attends pas de réponse.
Les méthodes du client sont donc bloquantes pour s'assurer de la réception de la réponse à la demande. L'affichage ne peut rien faire entre sa demande et la réponse qui lui est donnée. C'est le cas quand il faut afficher le nom des lombrics ou encore lorsque l'utilisateur se connecte.

Librairie utilisée

Afin d'envoyer des messages structurés entre le serveur et le client, nous utilisons la librairie protocol-buffers. Cette librairie permet de définir des structures qui après compilation donnent une classe en $C++$. Cette classe permet de remplir les champs de la structure et peut simplement être convertie en *string*. Ce *string*, une fois envoyé au serveur peut être reconverti en la classe. Cette librairie est donc également utilisée par le serveur.

5.4 Module serveur

Dépendances

Le serveur utilise les librairies suivantes :

— Google's Protocol Buffers : Une librairie utilisée pour facilement transférer des données au travers du réseau.

— Sqlite3 : Une librairie utilisée pour stocker des données à long terme sur le serveur, de les récupérer et les trier facilement.

— Bcrypt : Une librairie de hachage de mots de passe, qui utilise l'algorithme Blowfish.

— ZeroMQ : Une librairie qui permet la communication inter-threads, au moyen d'un système de "publisher/subscriber", qui permet à des threads de s'abonner à des canaux de communication, ou à y envoyer un message.

Découpe en threads

La serveur est découpé en plusieurs threads :

— Thread principal : Ce thread est à la base de tous les autres, c'est celui qui attend des connections de la part des clients, et qui génère tous les autres threads. Dès qu'un client se connecte au serveur, il se réveille et assigne un thread (expliqué au point suivant) à ce client.

— Client : Un thread créé par le thread principal qui gère la communication avec un seul client à la fois. Il peut notamment interagir avec la base de donnée, lorsque l'utilisateur souhaite envoyer un message, récupérer ses messages reçus, voir sa liste d'amis, le classement général, son historique de partie, ou encore modifier les noms de ses lombrics.

— Broker : Le broker est un thread qui permet la communication entre les threads clients, ou entre les threads client et leurs salons respectifs. Il agit comme une sorte de service postal, il reçoit des messages de tout le monde, qui sont soit destinés à un autre utilisateur, auquel cas il les redirige, soit destinés au broker lui même (Par exemple, un utilisateur peut demander au broker de créer un thread partie, ou autre). Ce thread est lancé au lancement du serveur par le thread principal, avant que celui-ci ne se mette en attente de connexion des clients.

— Partie : Un thread qui gère l'entièreté d'une partie, la communication avec les joueurs connectés à cette partie, le salon d'attente (avec modification des paramètres de partie, invitation de joueurs, ...)

Module Communication

Le module communication du serveur est représenté dans ce projet par une classe : "Listener". Cette classe possède plusieurs méthodes, en rapport avec la manière dont le client et le serveur communiquent. Elle permet entre autres de

récupérer le type de message qui va arriver, et du préparer l'environnement à la réception du message en lui-même. Ce module permet également d'envoyer des messages, en précisant encore une fois le type de message que l'on désire envoyer.

Module Salon/Partie

Ce module est géré par un thread à part qui est lancé par le Broker lorsqu'un client souhaite lancer une partie. Le thread qui gère ce module est une sorte de mini broker, qui réceptionne tous les messages que les clients lui envoie, et les transfère aux autres clients, ou les interprètent. Il est donc abonné à son propre canal ("room/<id>/client") pour les messages de la part des clients sur la room <id>.

Le module salon gère l'attente des joueurs avant la partie. Il permet au modérateur de la partie (celui qui l'a créée, de modifier les paramètres de la partie qui suit. Un salon contient des équipes, lesquelles contiennent des joueurs. Ces derniers possèdent leur équipe de lombrics. C'est durant cette phase que les joueurs peuvent en inviter d'autres à jouer.

Le module partie représente une partie jouée. Le module gère la carte, les interactions entre la carte et les joueurs le temps alloué à chaque joueur pour jouer son tour et l'apparition de bonus. Il s'occupe aussi de calculer tout ce qui est trajectoire des missiles ainsi que des dégâts pour éviter que le client puisse tricher sur les informations envoyées au serveur.

Module Base De Donnée

Ce module gère les interactions avec la base donnée, notamment lorsqu'un joueur s'inscrit, ou qu'une partie se termine. Il permet aux autres modules de faire des requêtes pour trouver des joueurs/parties/..., ou d'écrire dans la base de donnée au moyen de méthodes propres. Pour éviter les soucis de concurrence dans l'écriture en base de donnée, ce module, représenté par un objet, est déclaré globalement, et chacun peut y accéder, mais, via un mutex, on peut garantir qu'à aucun moment il n'y aura deux partie du code qui utiliseront cet objet en même temps.

Diagrammes

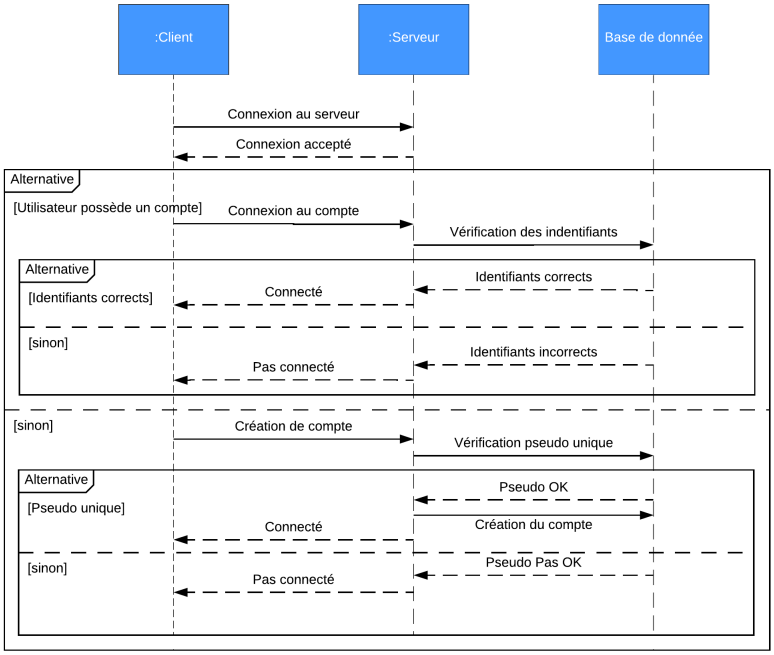

FIGURE 8 – Diagramme serveur client pour la connexion/inscription

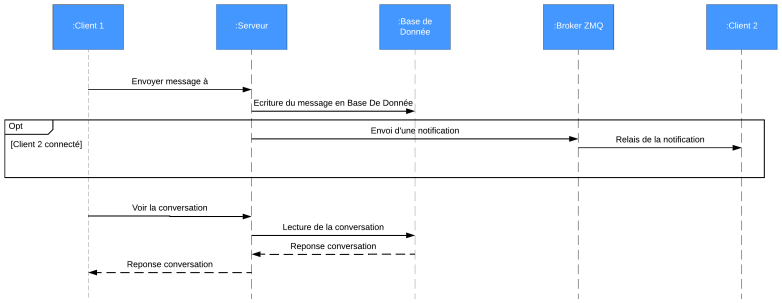

FIGURE 9 – Diagramme envoi de message