UNIVERSITÉ
DE REIMS
CHAMPAGNE-ARDENNE

# STARLYZE

## INFO601/604 – Jeu multijoueur en réseau

HADID Hocine et CHEMIN Pierre | 02/04/2023
Dirigé par **RABAT Cyril et DELISLE Pierre**

# TABLE DES MATIERES

# 1. Introduction

Nous avons entrepris la conception d'un projet appelé Starlyze, qui consiste à développer un jeu de plateforme multijoueur en ligne. L'objectif est de créer un monde virtuel dans lequel chaque joueur peut contrôler un personnage et interagir avec d'autres joueurs de la même partie.

Le concept de base du jeu est simple : chaque joueur peut créer son propre monde et y jouer avec d'autres joueurs. Cela signifie qu'il n'y aura pas de monde prédéfini, et chaque joueur aura la liberté de créer le monde qu'il souhaite avec ses propres niveaux, décors, obstacles et ennemis. Ainsi, chaque partie de Starlyze sera unique et offrira une expérience de jeu différente à chaque fois.

Le projet est divisé en trois parties principales : l'éditeur, le serveur et le client. L'éditeur permet aux joueurs de créer leur propre monde et de le personnaliser selon leurs préférences. Les joueurs peuvent ajouter des objets, des niveaux et des décors à leur monde. Le serveur est la partie du jeu qui gère la mise en relation des joueurs et la gestion de la partie. Il est responsable de la création de parties, de la gestion des joueurs et de la communication avec les joueurs. Enfin, le client est l'application qui permet aux joueurs de visualiser et de jouer au jeu. Il est conçu pour être facile à utiliser et à comprendre, avec une interface utilisateur intuitive. Le client affiche la partie en temps réel et permet aux joueurs de contrôler leur personnage.

De plus, les applications « serveur » et « client » communiquent tout au long de la partie via différents protocoles afin de garantir la fluidité de l'expérience de jeu et une interaction en temps réel entre les joueurs.

Dans le but de vous exposer la construction de ce projet avec clarté, nous allons tout d'abord nous pencher sur la conception de l'éditeur de monde. Ensuite, nous explorerons en détail les interactions entre les applications client-serveur, en examinant leurs échanges de données et leurs fonctionnements respectifs. Enfin, nous aborderons la gestion complète de la partie.

## 2. Editeur de monde

Dans cette partie nous allons aborder l'application permettant d'éditer les mondes et niveaux du jeu. Il faut effectivement d'abord créer son monde et ses niveaux avant de pouvoir jouer. Nous verrons comment nous avons géré la sauvegarde des mondes, l'édition de ces derniers et comment leurs niveaux sont représentés dans l'éditeur ainsi que son fonctionnement.

### A. La représentation d'un monde dans l'éditeur

Nous aborderons ici la représentation d'un monde et sa gestion uniquement du côté de l'éditeur.

Tout d'abord pour ce qui est de la représentation d'un monde. Il est composé d'une multitude de grille que l'on appellera niveau ou « level ». Un niveau est tout simplement une grille de 60x20, c'est-à-dire une matrice de 60 cases de largeur et de 20 cases de hauteur. Un monde proprement construit et complet est composé de plusieurs niveaux dont seulement un possède la « porte d'entrée » à savoir là où les joueurs apparaissent dans le monde et un autre niveau qui détient la « porte de sortie », c'est-à-dire la fin du monde, là où le joueur doit se rendre pour gagner.

#### 1) Les éléments d'un niveau

Chaque niveau possède des éléments appelés « sprite », chacun d'eux possède un identifiant, la largeur ainsi que la hauteur, la position X, la position Y et la spécificité. Ce dernier attribut n'est valable que pour les portes, les portails et les clés. En effet, ces éléments ont besoin d'informations complémentaires qu'il est essentiel de conserver pour permettre le bon fonctionnement du jeu. Une porte a besoin d'un identifiant spécifique pour savoir vers quelle autre porte elle mène, le même principe s'applique aux portails et clés qui fonctionnent de pairs.

Lorsqu'un élément est enregistré dans un fichier, on n'écrira que son identifiant et sa spécificité puisqu'avec l'identifiant on retrouve le type d'élément de la case permettant alors de définir sa largeur et sa hauteur qui n'ont donc pas besoin d'être sauvegardé.

#### 2) Le monde dans l'éditeur

Pour représenter notre monde dans l'éditeur nous avons créé une structure intitulée « game_level » qui possède plusieurs paramètres :

- ***width*** **:** un entier représentant la largeur du niveau
- ***height*** **:** un entier représentant la hauteur du niveau
- ***elements_map*** **:** une matrice en deux dimensions contenant des pointeurs vers les éléments du niveau

Le fait de représenter le niveau sous forme d'une matrice de pointeur nous permet de faciliter la gestion mémoire des éléments en évitant de multiples copies ou encore des oublis de libération de mémoire lors de la suppression ou modification d'un élément. Par exemple, pour ajouter un élément il suffit que chaque case occupée par cet élément pointe vers ce dernier. C'est pourquoi pour le supprimer ou le modifier on ne modifie que le pointeur de l'élément pour que cela soit effectif sur toutes les cases. Lors de la suppression toutes les cases occupées par l'élément sont mises à **NULL** et on libère la mémoire occupée par l'élément.

### 3) Le monde dans un fichier

Pour pouvoir sauvegarder au mieux un monde dans un fichier, il faut que ce dernier soit structuré. Par conséquent, le fichier possède des tables d'adresses qui sauvegarde la position de la représentation des niveaux dans le fichier. On retrouve également des tables de vides qui permettent de savoir où on peut insérer un niveau sans problème. Comme on peut le voir sur le schéma suivant, chaque table d'adresse possède 10 entrées, la première étant réservée pour pointer vers la table précédente et la dernière pour la table suivante, ce qui fait qu'il reste 8 adresses utilisables pour renseigner nos niveaux. Chaque entrée possède deux attributs, le premier est la position de l'élément pointé et le deuxième sa taille.

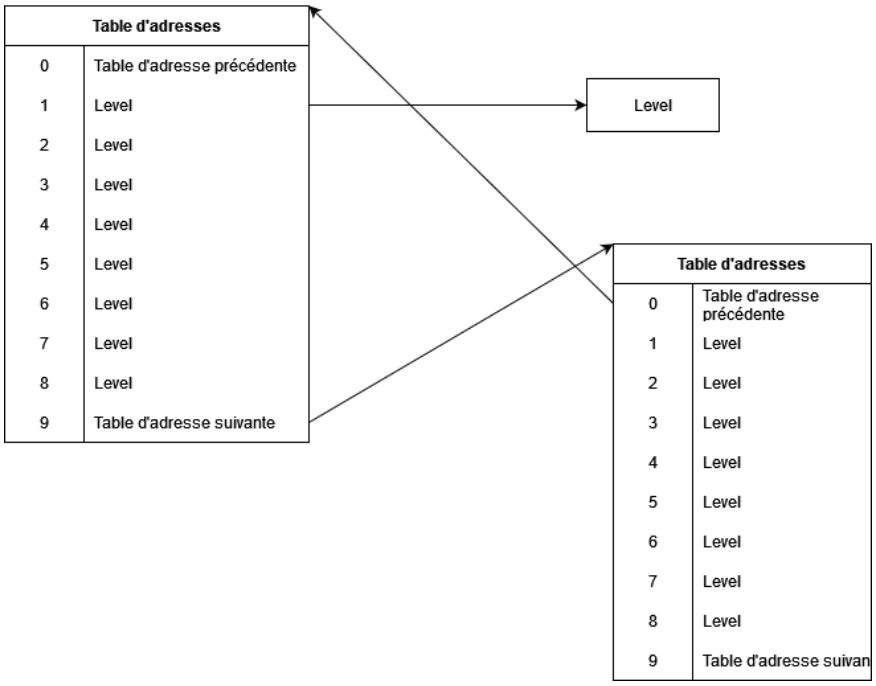


Figure 1 - Schéma de fonctionnement des tables d'adresse

On rappelle que pour un monde il faut un fichier qui sera composé au minimum d'une table d'adresses, d'une table de vide et d'un niveau. Il ne peut pas y avoir plus d'un seul monde dans le même fichier. Le nom du fichier est le nom du monde et son extension est « world ».

Ensuite, pour minimiser les appels systèmes « read », « write » et « lseek », il a fallu réfléchir à une structure et une méthode spécifique pour enregistrer un niveau. En effet, nous évitons d'enregistrer toutes les informations de chaque élément car certaines peuvent être retrouvé notamment la largeur et la hauteur qui sont spécifique au type de sprite. C'est pour cela qu'un élément du niveau n'a pas la même représentation dans le fichier que dans l'éditeur. Dans un fichier nous enregistrons seulement son type suivit de sa spécificité. De plus, les éléments sont enregistrés en parcourant chaque case de la matrice représentant le level. C'est-à-dire que l'on enregistre en premier l'élément à la case {0,0}, ensuite la case {0,1} et pour finir à la dernière case {m,n} où m représente le nombre de ligne et n le nombre de colonne. S'il n'y a aucun élément dans la case alors on écrit des 0 pour le type et -1 pour la spécificité.

Avec cette représentation, il est maintenant facile d'aller lire ou écrire un élément spécifique pour le modifier ou le supprimer sans avoir à réécrire tout le niveau dans le fichier, ce qui permet de limiter le nombre d'appel à « read » et « write ».

### B. Le fonctionnement de l'éditeur

Pour lancer l'éditeur il suffit d'utiliser la commande « ./bin/editor [nom du fichier monde] » dans un terminal. Si le fichier existe alors l'éditeur l'ouvre sinon il le crée avec un premier niveau vide. Notre fichier monde contient alors une table d'adresse avec une adresse vers le premier niveau suivit d'une table de vide qui est vide et enfin le premier niveau entièrement vide. A partie de là l'édition du monde peut commencer.


Figure 2 - L'éditeur de monde

Tout d'abord, il va falloir se familiariser avec l'interface utilisateur. Dans la fenêtre principal (la plus grande) il y a la représentation du niveau actuel, dans la fenêtre secondaire qui se situe à la droite de la fenêtre principal ce sont les différents outils permettant d'éditer le monde, et enfin dans la dernière fenêtre située tout en bas nous avons les différentes informations données par l'éditeur.

Dans la fenêtre des outils il est possible de sélectionner des spécificités seulement pour les portes, gates et clés, il s'agira de la couleur pour les deux derniers éléments leur permettant de fonctionner ensemble. Une fois l'outil s'électionné il suffit de cliquer dans le niveau à l'endroit où l'on souhaite placer l'élément. Si des sprites sont déjà présents sur la zone occupée par le nouvel élément, ces derniers seront automatiquement supprimés.

En ce qui concerne la sauvegarde, celle-ci s'effectue automatiquement dès qu'il y a un changement sur le niveau. De plus, la modification ne se fait que sur les éléments qui ont été modifié permettant de limiter les appels système superflus.

## 3. Les applications client-serveur et leurs échanges

Au sein de ce projet, le serveur ainsi que le client occupent une place primordiale, en effet, ce sont ces derniers qui gèrent la gestion et le bon fonctionnement de la partie. C'est pourquoi dans cette partie, nous allons aborder différents points relatifs au serveur et au client, comme par exemple les différentes communications entre ces derniers ou encore leur implémentation.

### A. Lancement des applicatifs

Afin de lancer le serveur il suffit d'utiliser la commande « bin/server [n° de port] » dans un terminal. Dans un premier temps, le serveur effectue toutes ses communications avec le client via le protocole UDP. C'est pourquoi au lancement, ce dernier crée une socket UDP avec le numéro de port spécifié en paramètre lors du lancement de celui-ci. Une fois la socket UDP mise en place et fonctionnel, le serveur se met alors en écoute des diverses requêtes qu'il peut recevoir par le client.

En ce qui concerne le client, pour le démarrer, nous devons exécuter la commande suivante : "bin/client [adresse IP du serveur] [numéro de port du serveur]". Si les informations d'adresse IP et de port du serveur sont correctement renseignées et correspondent à ceux du serveur démarré, le client sera alors lancé.

### B. Communication entre les deux applications

Afin de communiquer entre eux, le client et le serveur vont utiliser deux protocoles : UDP et TCP. L'usage du protocole UDP se fait dans le contexte du paramétrage des clients ainsi que des parties crées et qui attendent d'être lancés.

Afin de standardiser l'échange entre les deux entités, nous avons mis en place deux structures : *request_client_udp_t* et *response_server_udp_t*. La première est utilisé lors de l'envoi d'une requête depuis le client vers le serveur tandis que la deuxième est utilisé pour le chemin inverse, soit l'envoie de la réponse à la requête reçu.

La structure *request_client_udp_t* est formatée de la manière suivante :

```
typedef struct
{
    int type_request;
    content_request_client_udp_t content;
} request_client_udp_t;
```

Figure 3 : Structure request_client_udp_t

Cette structure comporte un élément de type union, ce qui évite la création de multiples structures pour chaque requête. En utilisant cette approche, nous réduisons considérablement la surcharge inutile de notre requête. Si nous avions opté pour une structure standard à la place de l'union, cela aurait engendré une charge supplémentaire à nos requêtes, étant donné qu'a requête toutes les champs devait être rempli.

C'est pourquoi notre variable *content_request_client_udp_t* est formatée comme suit :

```
typedef union
{
    char pseudo[MAX_MSG];
    int id_client;
    int settings_game[3]; // [0] : name_world | [1] : nb_players_game | [2] : id_client
    int choice_game[2];   // [0] : id_game | [1] : id_client
} content_request_client_udp_t;
```

Figure 4 : Union content_request_client_udp_t

Quant à la réponse du serveur aux différentes requête du client, nous retrouvons la même architecture que *request_client_udp_t* dans *response_server_udp_t*. Cependant, le contenu diffère, en effet, nous avons l'union suivante au sein de cette dernière :

```
typedef union
{
    int nb_clients;
    int id_clients;
    int port_tcp;
    list_world_response_t list_world;
    list_game_without_pointers_t list_game;
} content_response_server_t;
```

Figure 5 : Union content_response_server_t

Au sein de cette union, nous retrouvons trois variables de type entier, ainsi qu'une variable structurée contenant la liste des mondes existants, et enfin une dernière variable structurée contenant la liste des parties en cours.

Et en ce qui concerne le type de la requête, au sein de notre fichier contenant les structures précédemment évoquées, nous avons créé une liste de constantes qui nous permet d'identifier la requête reçue, ainsi nous pouvons déterminer le traitement à effectuer par la suite.

### C. La gestion des parties et des joueurs

La gestion des parties et des clients se fait principalement au sein du serveur. En effet, c'est ce dernier qui met en lien toutes les parties de jeu et les clients qui ont lancé l'application.

#### 1) La gestion des parties jeux

Afin de gérer les différentes parties de jeux qui peuvent être lancée sur le serveur, nous avons décidé d'opter pour une liste chainée contenant une structure représentant les différents paramètres d'une partie de jeu. Cette liste chainée est allouée dès le lancement du serveur et à chaque partie créée, celle-ci est directement enregistrée au sein de la structure de donnée.

Ainsi, une partie de jeu est représentée par la structure suivante :

```
typedef struct
{
    int id;
    int nb_participants_final;
    int nb_participants_actual;
    char name_world[MAX_MSG];
    list_info_client_t *list_players;
    struct game_t *next;
    struct game_t *prev;
}game_t;
```

Figure 6 : Structure représentant une partie

Nous trouvons dans cette structure un ensemble d'informations essentielles : un identifiant unique pour la partie, le nombre de participants finaux attendus, le nombre actuel de participants, qui est incrémenté dès qu'un joueur rejoint la partie, le nom du monde sélectionné, ainsi qu'une liste chaînée de clients représentant les joueurs présents dans la partie en question.

2) La gestion des clients

Pour assurer la gestion des différents clients se connectant au serveur, comme pour les parties de jeux, nous avons également mis en place une liste chainée contenant une structure. Cette structure est composé de différents éléments utiles à l'administration des différents clients. Tout comme les partie de jeux, cette structure de donnée est alloué dès le lancement du serveur. Et dès lors qu'un client se connecte, il est automatiquement ajouté dans cette dernière.

Ainsi, la liste des clients connectés au serveur sont décrits par la structure suivante :

```
typedef struct
{
    int id;
    char pseudo[MAX_MSG];
    char *client_address;
    int port;
    struct info_client_t *next;
    struct info_client_t *prev;
}info_client_t;
```

Figure 7 : Structure représentant un client

Dans cette structure, nous pouvons identifier plusieurs éléments : tout d'abord, un identifiant unique qui permet d'identifier chaque client. Nous trouvons également le pseudo associé à ce client ainsi que son adresse IP et le port correspondant, ce qui nous permettra ultérieurement de lui envoyer le numéro du port de la socket TCP lorsque la partie sera lancée. Enfin, les pointeurs **next* et **prev* permettent la mise en place d'une liste chaînée, facilitant ainsi la gestion de ces informations.

Cette structure est utilisé par les deux parties, en effet, le serveur et le client ont recours à cette dernière afin de gérer les informations du joueur.

## D. Les différents échanges entre le serveur et le client via le protocole UDP

Durant cette partie, nous allons voir les différentes communications entre le serveur et le client effectuée par le biais du protocole UDP, protocole permettant des échanges fluides et rapides entre deux entités.

### 1) Première connexion d'un joueur au serveur

Maintenant que nos deux applications sont en cours d'exécution et que les formats des requêtes et les différentes structures de gestion ont été définis, la première étape pour le client sera de saisir le pseudonyme qu'il souhaite utiliser tout au long de la partie.

Après avoir saisi le pseudo, une première communication UDP est établie avec le serveur. En effet, le client envoie une requête soigneusement composée, incluant le type de requête ainsi que le pseudo choisi. A la réception de cette requête, le serveur répond à cette dernière en envoyant l'id généré au sein de son système de gestion des clients et enregistre le client au sein de la structure adéquate, structure que nous avons évoqué auparavant.

Nous avons donc le diagramme d'échange suivant, représentant la communication précédemment décrite :

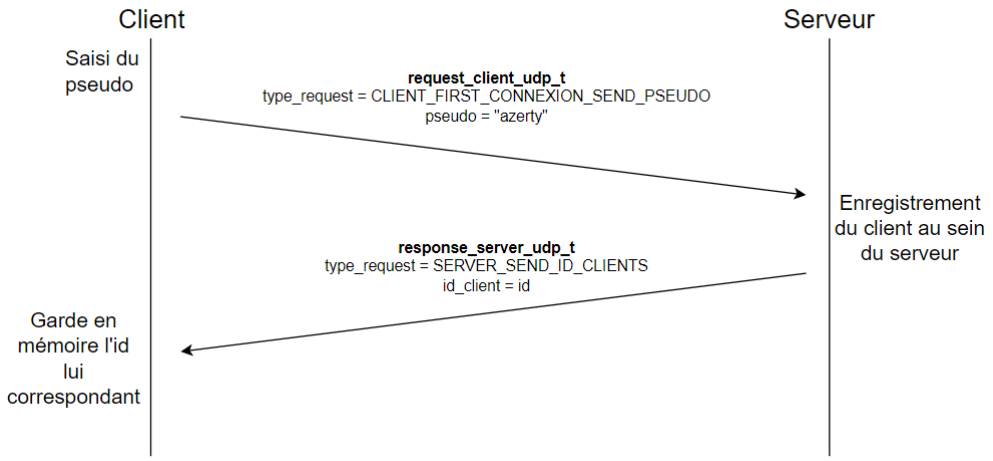


Figure 8 : Diagramme d'échange correspondant à l'enregistrement du client au sein du server

### 2) Récupération du nombre de joueurs actuels

Une fois que le joueur a saisi son pseudonyme et que toutes les étapes de gestion ont été effectuées, un menu s'affiche côté client. Ce menu permet au joueur de créer une partie, de rejoindre une partie existante ou encore de connaître le nombre de joueurs actuellement connectés sur Starlyze. Pour obtenir cette information, une communication avec le serveur est alors nécessaire. À cet effet, en utilisant la même socket de communication que celle initiée lors de la première connexion, le client envoie une demande de récupération du nombre de joueurs actuels. Le serveur répond alors en envoyant le nombre correspondant au client.

Nous avons donc le diagramme d'échange suivant, qui représente cette communication :

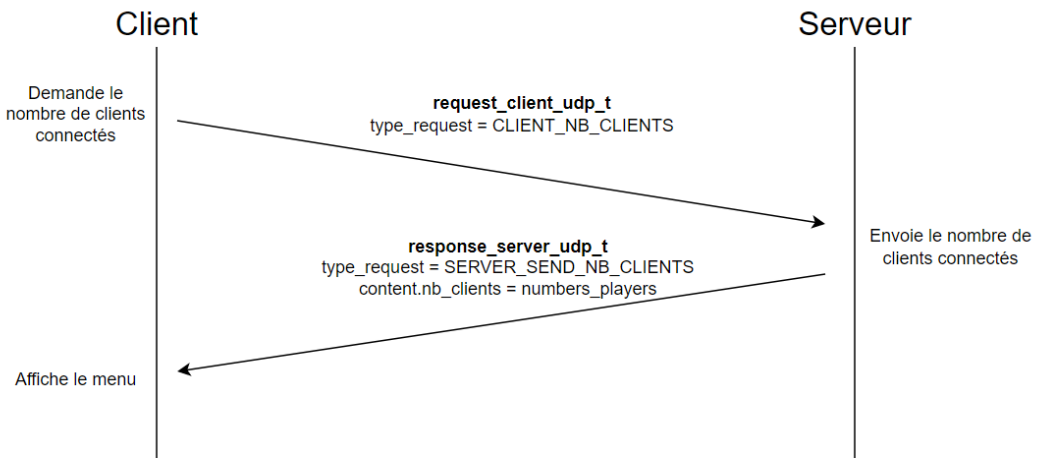


Figure 9 : Diagramme d'échange correspondant à la récupération du nombre de joueurs connectés sur Starlyze

Suite à cette réponse, le client peut alors afficher le nombre de joueurs connectés sur Starlyze au sein du menu, comme nous pouvons le voir ci-dessous :


Figure 10 : Menu coté client

### 3) Créer une partie de Starlyze

Comme nous pouvons l'observer sur la figure n°8, le menu proposé côté client permet différentes options, dont celle de créer une partie de Starlyze. Si le joueur sélectionne cette option, cela déclenche une série de communications avec le serveur.

En effet, dans un premier temps, pour lancer une partie, il est nécessaire de récupérer les différents mondes disponibles dans le jeu. Pour ce faire, en utilisant la même socket UDP que celle utilisée lors de la première connexion, le client envoie une première requête au serveur afin signaler sa demande de récupération de la liste des mondes disponibles. Le serveur ayant chargée la liste des mondes au sein d'une liste chainée lors de son lancement renvoie celle-ci au client intéressé. Une fois cette liste récupérée, le client l'affiche alors à l'utilisateur.

Puis dans un second temps, une fois que le joueur a sélectionné le monde auquel il souhaite jouer et précisé le nombre de joueurs nécessaires pour que la partie soit lancée, une deuxième communication est établie avec le serveur. Cette communication permet d'envoyer au serveur les différentes informations concernant la partie qui vient d'être créée, afin qu'il puisse l'enregistrer au sein du système de gestion des parties. Cette étape est essentielle pour permettre à d'autres joueurs de rejoindre la partie nouvellement créée.

Ainsi, afin de créer une partie, nous avons donc les échanges suivants qui sont effectués :

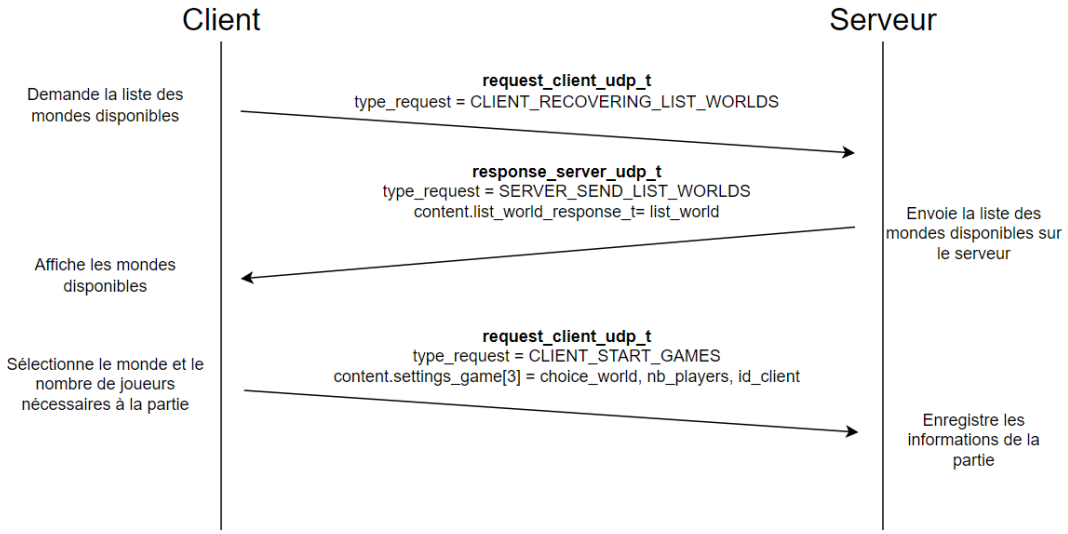


Figure 11 : Diagramme d'échange correspondant à la création d'une partie de Starlyze

### 4) Rejoindre une partie créée

Hormis le fait qu'un joueur peut créer une partie, il peut également en rejoindre une déjà existante. Tout comme la création d'une partie, ce choix entraine une série de communication avec le serveur.

Dans un premier temps, afin de rejoindre une partie, il est nécessaire d'obtenir la liste des parties en attente et qui ne sont pas encore démarrées. C'est pour cela que le client envoie une première requête afin de pouvoir de récupérer la liste des parties en attente. Le serveur ayant une liste de toutes les parties en attente renvoie celle-ci, afin que le client puisse l'afficher.

Une fois que la liste des parties en attente s'affiche, le joueur peut alors effectuer sa sélection. À cet instant précis, une seconde requête est envoyée par le client au serveur, permettant ainsi de communiquer le choix du joueur quant à la partie qu'il souhaite rejoindre. Au sein cette requête, qui inclut non seulement l'identifiant de la partie sélectionnée, mais également l'identifiant du joueur lui-même. Cette information permet au serveur d'ajouter plus facilement le joueur sans qu'il ait à envoyer toutes ces informations telles que son pseudo, son adresse IP ou encore son port. En effet, grâce à l'identifiant fourni, le serveur peut directement récupérer le joueur concerné dans son système de gestion de client.

Suite à cela, le serveur effectue une vérification cruciale afin de s'assurer que le nombre de joueurs requis pour la partie est atteint. Si tel est le cas, le serveur crée une socket TCP et envoie le port TCP correspondant à tous les joueurs. Cette étape est essentielle pour assurer une connectivité stable et fluide pendant toute la durée de la partie ainsi que le début de la partie. Nous aborderons cette étape en détails dans les prochaines sections de ce rapport.

Ainsi, afin de rejoindre une partie, nous avons donc la série d'échange suivante qui sont effectués :

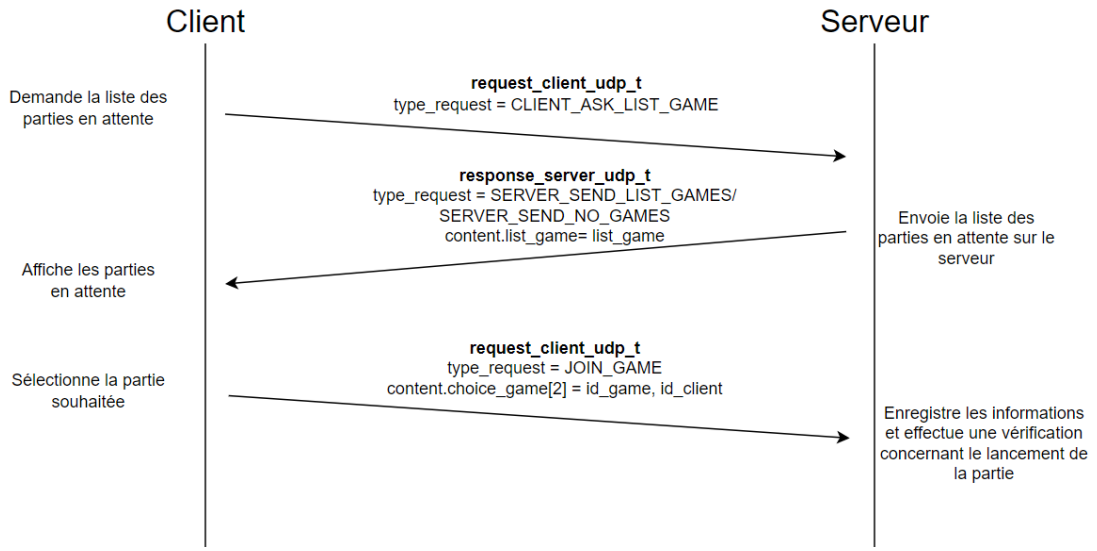


Figure 12 : Diagramme d'échange correspondant à la connexion d'un joueur à une partie

Il est important de noter que le diagramme illustre clairement que, lors de la réponse du serveur pour l'envoi de la liste des parties en attente, le type de requête peut prendre deux valeurs différentes. Si la valeur est SERVER_SEND_LIST_GAMES, cela signifie qu'il y a effectivement des parties en attente et que le joueur peut en choisir une pour y participer. En revanche, si la valeur est SERVER_SEND_NO_GAMES, cela indique qu'il n'y a aucune partie en attente, et que le joueur devra donc créer une partie pour pouvoir jouer à Starlyze.

### 5) Envoie du port de la socket TCP

Une fois qu'une partie a atteint le nombre de joueurs requis, nous créons donc une socket TCP au sein du serveur afin que celle-ci servent au déroulement de la partie. En effet, le déroulement de la partie se fait exclusivement en TCP. Nous ne rentrerons pas dans les détails de cela au sein de cette sous-partie, nous aborderons ce point ultérieurement au sein de ce rapport.

Afin que le client se connecte à la socket TCP, nous envoyons le port de notre socket via UDP vers le client. En ce qui concerne l'adresse IP, il utilisera la même que celle qu'il a utilisé pour UDP. Ce port a été généré automatiquement par le système d'exploitation. Une fois le numéro de port reçu, le client crée alors une socket TCP de son coté. Ainsi, à partir de cette étape, tous les échanges entre le serveur et le client se feront exclusivement en TCP.

Nous avons donc le diagramme d'échange suivant :

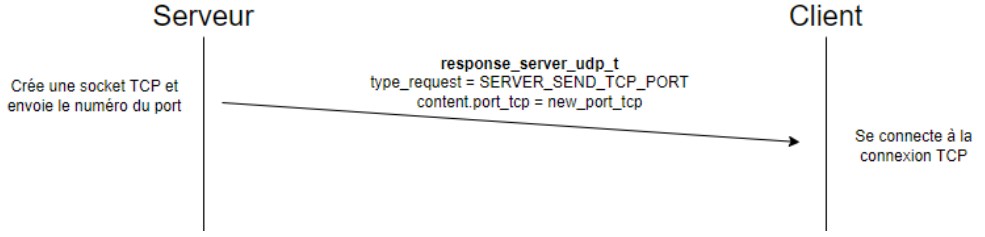


Figure 13 : Diagramme d'échange représentant l'envoie du port TCP

### 6) Déconnexion du client

Enfin, une fois qu'un joueur a créé ou rejoint une partie et qu'elle s'est terminée, ou bien si le joueur décide d'arrêter le jeu avant même d'avoir lancé une partie, il est essentiel de le déconnecter du serveur. Cette étape est importante pour garantir que le joueur ne soit plus comptabilisé dans la liste des joueurs connectés, évitant ainsi une surcharge du serveur. Cette fonctionnalité de déconnexion assure également une gestion efficace des ressources du serveur et une meilleure expérience de jeu pour tous les utilisateurs.

Dès qu'un joueur effectue l'une des actions mentionnées précédemment, la procédure de déconnexion est lancée. Pour cela, le client envoie une requête au serveur pour signaler qu'il souhaite se déconnecter, en incluant son identifiant dans la requête. Lorsque le serveur reçoit cette requête, il entame alors la procédure de déconnexion, qui consiste à supprimer toutes les informations relatives au joueur déconnecté qu'il avait sauvegardées jusqu'à présent.

Nous avons donc le diagramme d'échange suivant, représentant la communication précédemment décrite :

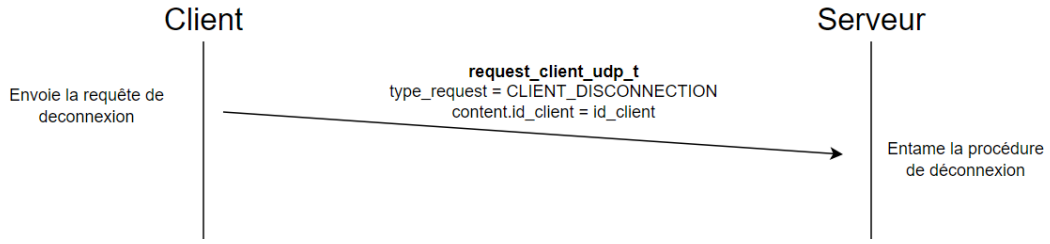


Figure 14 : Diagramme d'échange représentant la déconnexion d'un client

### E. Les différents échanges entre le serveur et le client via le protocole TCP

Lors du déroulement d'une partie, les échanges entre le client et le serveur se font via le protocole TCP. En terme de type de requête il en existe peu puisque les seuls échanges nécessaires sont ceux de l'actualisation du level, les actions réalisées par un joueur, la fin de partie et si un joueur est immobilisé.

Tout d'abord, pour l'actualisation d'un level, nous envoyons une requête de type REFRESH_LEVEL depuis le serveur vers le client concerné. Cette requête est composé des champs type_request, level_display (le level à afficher), et player (le joueur concerné). Etant donné que l'actualisation du coté du client doit se faire quasiment instantanément il s'agit d'une requête qui est envoyé toutes les 0,1 secondes. Du côté client, lors de la réception de cette requête, il affiche juste les nouvelles informations reçu et n'émet aucune réponse si ce n'est l'acquittement du protocole TCP comme on peut le voir sur le diagramme d'échange suivant.

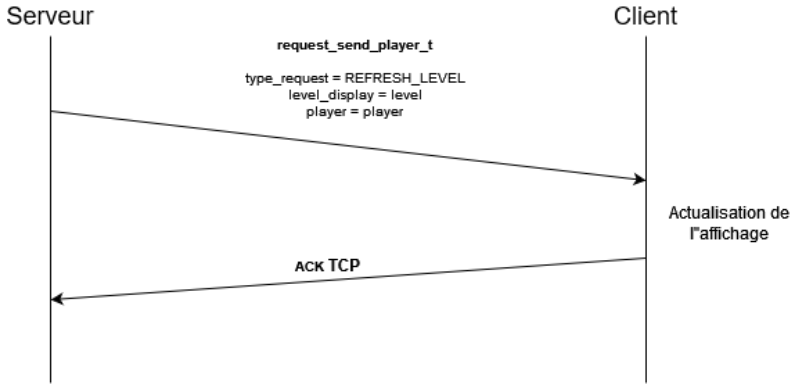


Figure 15 - Diagramme d'échange représentant l'actualisation de l'affichage du client

Toujours du côté serveur, nous allons aborder la requête permettant d'immobiliser un joueur, cette requête est nécessaire lorsque le joueur est immobilisé pour pouvoir en informer le client puisque pendant le temps d'immobilisation il faut éviter que le client envoi des requêtes de déplacement. Pour cette requête d'immobilisation, nous avons les mêmes champs que précédemment auxquels on rajoute le champ second_freeze qui indique le nombre de secondes d'immobilisation du joueur, le champ type_request est à la valeur FREEZE_PLAYER. Le joueur n'a pas besoin de réponse à cette requête ce qui fait que nous nous retrouvons avec le diagramme d'échange suivant, dans lequel on peut voir qu'à la réception, le client actualise tout de même l'affichage et qu'il envoi l'acquittement TCP.

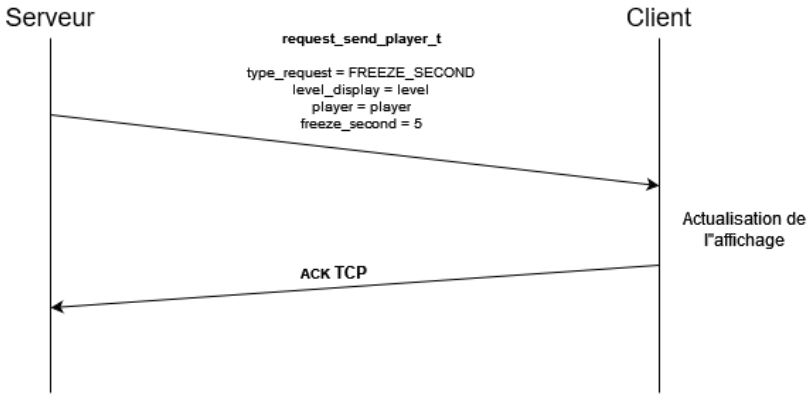


Figure 16 - Diagramme d'échange représentant la requête FREEZE_SECOND

Maintenant, la dernière requête du serveur est celle concernant la fin de la partie. En effet, lorsque la partie arrive à son terme il faut en informer les clients pour qu'ils puissent arrêter d'envoyer des requêtes sur le serveur de jeu et informer le joueur de la fin de la partie. Pour cette requête, nous aurons besoin des champs type_request, message et player. Pour le type de requête, il sera END_GAME, le message et la raison pour laquelle la partie est arrivé à son terme (vitoire d'un joueur) et précise si le joueur qui reçoit la requête a gagné ou non. Le champ player, quant à lui précise le joueur et ses informations. Nous obtenons donc le diagramme d'échange suivant, et encore une fois le client n'a pas besoin de répondre à cette requête.

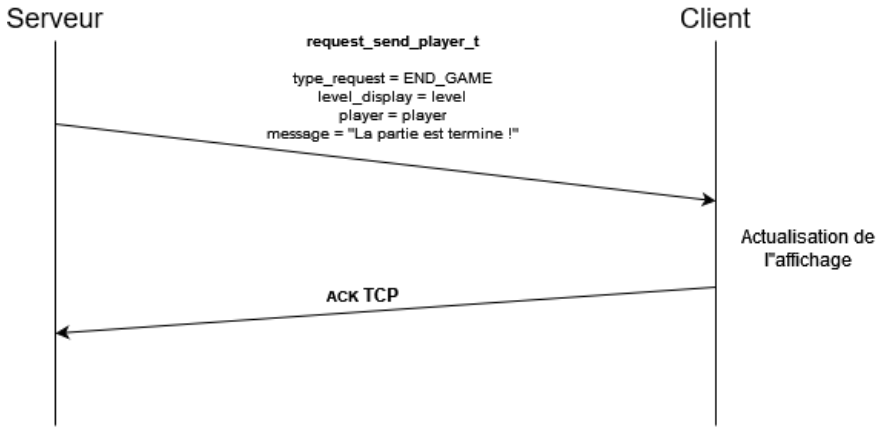


Figure 18 - Diagramme d'échange représentant la requête de fin de partie

Passons à la requête du client, il n'y aura ici qu'une seule requête envoyée par le client et il s'agit de ses actions. La requête du client est très simpliste puisqu'il ne s'agit que d'un caractère qui est envoyé. Ce caractère précise l'action effectué par le joueur. En termes d'échange TCP nous restons sur la même complexité qu'avant. Etant donné que l'actualisation s'effectue automatique toutes les dixième de seconde, nous n'avons ici pas besoin de répondre avec le nouvel affichage puisque nous sommes sûrs qu'il sera effectué.

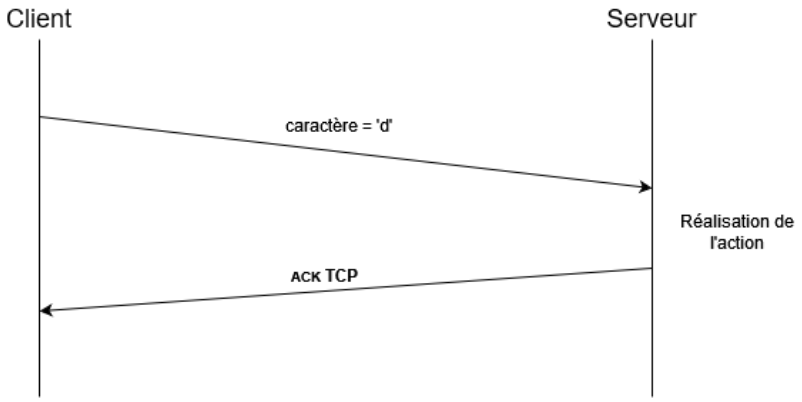


Figure 17 - Diagramme d'échange représentant la requête d'action du joueur

## 4. La gestion d'une partie

Nous allons ici aborder la gestion d'une partie de Starlize, entre le chargement d'un monde, la gestion des déplacements et des événements ainsi que les problèmes rencontrés et les choix de développement pris.

### A. Lancement d'une partie

Comme nous avons pu le voir dans les parties « rejoindre une partie créée » et « envoie du port de la socket TCP », lorsque tous les clients requis sont présents alors le serveur va créer une socket TCP dont nous laissons le système choisir le port. Ce dernier est ensuite envoyé aux clients de la partie. Le serveur va également créer un nouveau processus qui sera chargée de suivre et contrôler la partie. Ce processus fils a besoin du nombre de joueur, de la socket TCP de la partie et du nom du monde choisit pour pouvoir être lancé.

Le processus commence par charger toutes les informations du monde en mémoire, puis initialise les variables nécessaires pour la partie telles que la variable de condition de victoire, le mutex de victoire et la variable indiquant l'identité du joueur gagnant.

Le gestionnaire de partie attend ensuite la connexion des joueurs sur sa requête TCP, et une fois le nombre de joueurs requis atteint, il lance les différents threads des joueurs, des ennemis et des pièges. Le thread principal du processus se met alors en attente sur la variable de condition de victoire. Lorsqu'un joueur atteint la porte de sortie, un signal est lancé sur la variable de condition, débloquant ainsi le thread principal et mettant fin à la partie. Le processus nettoie alors toute la mémoire, en indiquant aux différents threads de se terminer et en libérant les ressources prises.

### B. Le chargement du monde

Au démarrage de la partie, notre première tâche est de charger le monde correspondant à son nom. Pour ce faire, nous procédons à une lecture du fichier monde en récupérant toutes les informations essentielles dès le début afin de minimiser le nombre d'appels système et d'accélérer le déroulement du jeu. Ces informations sont ensuite stockées dans une structure de données spécifique, la world_info_t, qui nous permet de les organiser efficacement.

```
typedef struct
{
    int doors_level[NUMBER_DOOR][2];

    int total_level;
    level_info_t *levels;

    int start_level;
    int exit_level;
} world_info_t;
```

Figure 19 - Structure world_info_t

Cette structure contient plusieurs informations cruciales. Tout d'abord, elle inclut un tableau à deux dimensions contenant toutes les informations relatives aux portes du monde. Le premier niveau de ce tableau correspond à l'identifiant de chaque porte, tandis que le deuxième niveau répertorie les identifiants des levels dans lesquels ces portes se situent.

De plus, une variable est dédiée à la sauvegarde du nombre total de levels présents dans le monde.

Ensuite, nous disposons d'un tableau de type "level_info_t" qui contient toutes les informations relatives à chaque level du monde. Enfin, l'identifiant du level de début ainsi que celui du level contenant la porte de fin sont également enregistrés. Avec ces informations, il est possible de naviguer facilement à travers les différents levels du monde et de permettre aux joueurs de passer d'un level à l'autre grâce aux portes.

Lorsque l'on charge un monde nous commençons par la lecture des différents levels contenus dans le fichier monde. Ces levels sont également stockés sous forme de la structure suivante :

```
typedef struct
{
    sprite_t map[WIDTH_LEVEL][HEIGHT_LEVEL];

    int number_enemy;
    int robot[NUMBER_ROBOT];
    int probe[NUMBER_PROBE];
    list_block_trap_t list_block_trap;

    int number_mutex_zone;
    pthread_mutex_t *mutex_zone;
} level_info_t;
```

Figure 20 - Structure représentant un level

Dans cette structure, on peut observer que chaque level est représenté par un tableau à deux dimensions de sprites, dotés d'un attribut de type et d'un attribut de spécification, comme pour la partie de l'éditeur. En outre, chaque level comprend un certain nombre d'ennemis, de tout type, et ceux-ci sont stockés dans des tableaux portant le nom de leur type. Dans ces tableaux, les positions des ennemis sont enregistrées sous la forme d'un entier unique, résultant d'une formule qui tient compte de leur position X et Y sur la carte. Quant aux pièges, ils sont stockés sous forme d'une liste de blocs de pièges, pour être gérés par bloc plus facilement. Enfin, afin de gérer les déplacements des entités et les événements de jeu tout en évitant des problèmes de synchronisation, chaque level dispose d'un tableau de mutex, chaque mutex gérant une zone du level. Dans notre projet, les zones de mutex mesurent 5 cases sur 5, ce qui représente un total de 48 mutex par level.

Ainsi, pour charger un level, il suffit de lire le fichier monde normalement en parcourant les tables d'adresse. Comme pour l'éditeur, chaque sprite est enregistré dans le tableau de sprite. Une fois cette étape terminée, il suffit de parcourir le level pour y enregistrer les différentes informations. Pour les ennemis, dès que nous en trouvons un sur la carte, nous calculons sa position X et Y en un seul entier que nous stockons dans le tableau de son type. Une fois cette opération effectuée, nous le supprimons de la carte, car cette variable ne stocke que les informations statiques du level, c'est-à-dire les éléments qui ne se déplacent pas, comme les portes, les portails ou les bombes.

Pour faire apparaître et disparaître les pièges par bloc, nous les stockons sous forme de listes chaînées de blocs de pièges. Un bloc de pièges est en fait une liste chaînée de pièges, dotés des attributs posX, posY et succ, qui indique le piège suivant de la liste. Un bloc de pièges représente donc les pièges collés entre eux sur la carte, soit horizontalement, soit verticalement. Chaque bloc est ensuite ajouté à la liste des blocs du level.

Une fois toutes ces étapes réalisées, notre monde est chargé en mémoire. Il ne reste plus qu'au gestionnaire de partie de lancer les différents threads sur les entités nécessitant un changement d'état, comme les ennemis et les pièges.

### C. Le déplacement des entités

Lorsqu'une entité souhaite se déplacer sur un niveau, elle appelle la méthode "move_level". Cette méthode est cruciale en termes d'actions réalisées, car elle effectue plusieurs vérifications, telles que les collisions, la victoire d'un joueur, ou encore la récupération d'objets.

Tout d'abord, nous calculons la position finale de l'entité sur laquelle nous effectuons les vérifications. Si la position finale est en dehors du niveau, le déplacement n'est pas autorisé. En revanche, s'il est possible de se déplacer dans une zone, nous bloquons le mutex de la zone initiale et celui de la zone de destination, si le déplacement amène l'entité dans une zone différente. Une fois les mutex bloqués, nous passons à la vérification du déplacement. Cette fonction détermine si le déplacement n'est pas bloqué par la présence d'un bloc, d'un vide, ou encore d'un piège. Pour les robots, nous vérifions également que les cases situées en dessous d'eux sont un bloc ou une échelle. En revanche, pour les joueurs, nous effectuons une vérification supplémentaire pour les portails, car chaque joueur a besoin d'une clé spécifique par portail. Nous vérifions donc sa présence dans l'inventaire du joueur.

Une fois l'étape de vérification de déplacement effectuée, nous pouvons déplacer l'entité. Si c’est un ennemi, nous vérifions s'il entre en collision avec un joueur. Si c'est le cas, nous retirons une vie au joueur. S'il n'a plus de vie, il recommence la partie depuis le début avec son inventaire vide. En revanche, si c'est un joueur qui se déplace, nous vérifions s'il est invincible. S'il ne l'est pas, nous vérifions les éventuelles collisions avec les ennemis du niveau. Si le joueur est en état de collision, nous effectuons la même opération que si c'était un ennemi qui entrait en collision avec le joueur. S'il n'y a pas de collision, nous vérifions si un objet est présent à la position du joueur et si c'est le cas, nous le récupérons. S'il s'agit d'une bombe, nous en rajoutons entre une et trois au joueur. S'il s'agit d'un objet de vie, nous réinitialisons la vie du joueur au maximum. S'il s'agit d'une clé, nous l'ajoutons à l'inventaire du joueur. Lorsqu'un joueur récupère une vie ou une bombe, l'objet disparaît pendant un certain temps. Pour cela, nous lançons un thread qui se charge de replacer l'objet après un certain temps. Nous passons à ce thread la position X et Y, le type et le niveau de l'objet à replacer.

La possibilité d'utiliser les portes est une autre option offerte aux joueurs, simplement en appuyant sur la touche ‘e’. Les opérations à effectuer pour ce faire sont similaires à celles décrites précédemment, mais la zone de destination se trouve dans un autre level. Lorsque le joueur franchit la porte, nous procédons au changement de level en modifiant la variable « level » du joueur, de sorte que le nouveau level soit pris en compte. Cette transition est transparente pour le joueur, qui peut ainsi explorer de nouveaux niveaux sans interruption.

Une fois que le déplacement et la récupération des objets éventuels est terminée, nous vérifions si le joueur est en état de victoire ou non. Si c'est le cas, nous modifions la variable "id_player_winner" pour ensuite envoyer un signal sur la variable de condition de victoire. Cela permet d'informer tous les joueurs que la partie est terminée et qu'il y a un gagnant. Enfin, nous terminons tous les threads et libérons la mémoire.

### D. Les threads des ennemies

En ce qui concerne la gestion des ennemis dans notre jeu, chacun d'eux est doté d'un thread dédié qui prend en charge son déplacement. L'algorithme de déplacement est simple mais efficace : les robots se déplacent uniquement horizontalement et, dès qu'ils rencontrent un mur ou un piège, ils changent de direction. Quant aux probes, ils se déplacent de manière aléatoire sur une case adjacente.

Durant leurs déplacements, des vérifications de collisions sont effectuées avec les joueurs. Si un ennemi entre en collision avec un joueur, ce dernier subit des dégâts et bénéficie d'un court temps d'invincibilité de trois secondes pour se rétablir.

### E. Les threads des pièges

En matière de gestion des pièges, nous avons opté pour une méthode organisée et efficace. En effet, nous avons prévu un thread dédié à chaque niveau, dont la mission sera de superviser les pièges correspondants. Ce faisant, il examinera chaque bloc de pièges et ajustera la spécification de chacun d'eux pour qu'elle s'accorde avec les autres pièges du même bloc.

Pour ce faire, notre thread se basera sur la première spécification de chaque bloc. Si elle est égale à -1, il choisira une valeur aléatoire entre 0 et 15 et la modifiera, soit en la rendant positive, soit en la rendant négative. Dans le cas où la spécification est négative, le piège sera masqué. En revanche, si la spécification est déjà positive ou négative, et qu'elle est différente de -1, notre thread incrémentera ou décrémentera la valeur pour la faire revenir à -1. Cette opération sera répétée tous les dixièmes de seconde pour chaque bloc.

Par ailleurs, comme pour la gestion des ennemis, notre thread vérifiera avant de boucler s'il doit s'arrêter.

### F. Les threads des joueurs

Chaque joueur connecté à la partie est associé à un nouveau thread, qui gère les échanges réseaux avec le client. Avant de commencer, le thread récupère les informations nécessaires pour son fonctionnement : le gestionnaire de jeu, la socket du client, et l'identifiant du joueur qui est extrait du tableau de joueurs.

Une fois ces informations collectées, le thread lance un autre thread qui se charge d'envoyer des informations de rafraîchissement au client. Ces informations incluent l'état actuel du niveau, les informations du joueur (inventaire, vie et level actuel) et les éventuels événements tels que l'immobilisation du joueur ou la fin de la partie.

Pendant ce temps, le thread principal est chargé de lire les informations envoyées par le client via la socket. Ces informations sont les actions que le joueur souhaite effectuer.

Lorsque le thread du joueur est arrêté, il envoie une dernière demande pour signaler la fin de la partie et arrête également le thread chargé de l'envoi des informations au client. La socket est alors fermée.

Ce processus permet de gérer efficacement les échanges entre les joueurs et le serveur, en utilisant des threads pour répartir la charge de travail et éviter les conflits lors de l'accès aux ressources partagées, tout en assurant une expérience de jeu fluide pour les joueurs.

### G. Le fonctionnement des bombes

Lorsqu'un joueur met la main sur un item de bombe, il a la possibilité d'en récupérer entre une et trois, qui seront immédiatement ajoutées à son inventaire. Pour poser une bombe, une simple pression sur la touche ‘g’ suffit. Une fois posée, elle apparaît en rouge et un thread spécialement créé pour gérer son explosion est instantanément mis en place.

Ce thread se charge de calculer les zones à verrouiller avant de patienter le délai d'explosion de la bombe, qui est déterminé aléatoirement entre quatre et dix secondes. Une fois le temps écoulé, le thread verrouille toutes les zones à proximité de la bombe pour affecter toutes les entités se trouvant dans son rayon d'action, qui s'étend sur cinq blocs.

Si un joueur se trouve dans ce rayon d'action, il perd une vie, tandis que les ennemis sont simplement immobilisés pendant cinq secondes. Après l'explosion, le thread replace l'élément qui occupait la place de la bombe dans le level, avant de déverrouiller les mutex.

## 5. Conclusion

Ainsi, ce projet de développement d'un jeu multijoueur en réseau, qui clôture notre licence, nous a permis de mettre en pratique les différentes notions abordées en cours, telles que la programmation en C et la gestion de sockets TCP. Nous avons ainsi pu développer une architecture client-serveur efficace, permettant à plusieurs joueurs de jouer ensemble en temps réel.

Nous avons également abordé des notions plus avancées, telles que la gestion de threads et la synchronisation d'accès aux ressources partagées, ainsi que la gestion d'un système de jeu complet, incluant la gestion de niveaux, de pièges, d'ennemis et d'objets.

Grâce à ce projet, nous avons pu travailler en équipe et collaborer efficacement, en utilisant des outils de développement collaboratifs tels que GitHub ou encore Trello. Nous sommes fiers du résultat obtenu et nous pouvons considérer que nous avons atteint nos objectifs de développement.

Pour conclure, ce projet qui a exigé de nous de nombreuses heures de travail acharné, s'est avéré être passionnant jusqu'à son achèvement.