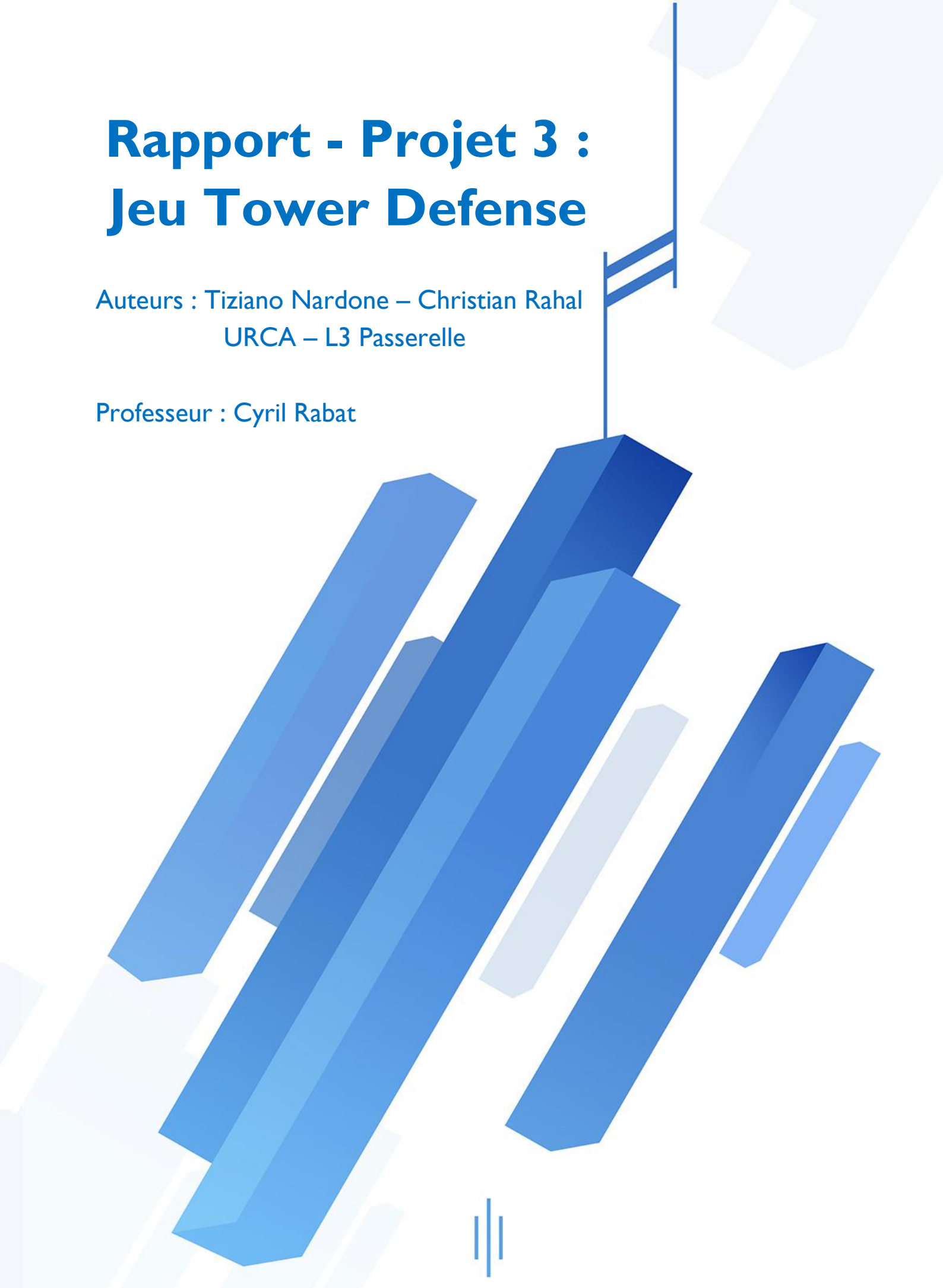
Rapport - Projet 3 : Jeu Tower Defense

Auteurs : Tiziano Nardone – Christian Rahal
URCA – L3 Passerelle

Professeur : Cyril Rabat



I. Table des matières

II. Avancement du projet au moment du rendu

A. Applications

La compilation crée deux exécutables, un appelé serveur et un autre client. Ces deux applications sont exécutables à partir d'un terminal, le client peut être exécuté plusieurs fois à partir de terminal différent. Lorsque ces deux applications sont lancées le client envoie des informations au serveur et celui -ci lui répond. Ces échanges d'informations sont peuvent être découpées en deux parties. La partie menu visualisable sur le terminal qui a lancé l'exécutable client. La seconde partie, il s'agit de la partie jeu en elle-même, celle-ci aussi est visualisable sur les terminaux qui ont lancé l'exécutable client. Les informations visualisables sur le terminal qui a lancé le code serveur sont des informations avancées, interprétables par les développeurs de l'application ou après une bonne lecture de ce rapport et des codes des applications.

La relation entre le client et le serveur est fonctionnelle en mode connecté et non connecté.

B. Menu

A l'exécution du code client un menu est affiché sur le terminal, ce menu demande à l'utilisateur de choisir entre quatre demandes à envoyer au serveur :

- Demande de liste des cartes disponibles
- Demande de liste des scénarios disponibles
- Demande de démarrage d'une partie en précisant la carte et le scenario choisie
- Demande de liste des parties démarrées

Toutes ces demandes sont envoyées au serveur et ce dernier répond au client en fonction de sa demande et de l'état de la ou des parties. Ainsi la partie menu est fonctionnelle.

C. Jeu

Une partie commence lorsque quatre clients se sont connectés à cette même partie. Lorsqu'un client est connecté à une partie, le serveur envoie la carte choisie par le client et toutes les secondes le serveur envoies des unités au client en fonction également du scénario choisi. Lorsqu'une unité arrive devant le fort du client, sa vie diminue de 1 point, arrivé 0 points le client à perdu des messages d'informations de fin partie sont affiché dans la fenêtre informations et le client s'arrête.

Les trois cartes sont implémentées ainsi que les trois scénarios, les scénarios sont fonctionnelles sur les trois cartes. Toutefois concernant les scenarios, seul le déplacement des unités est fonctionnel dans une partie. Enfin les actions d'un utilisateur possible sur l'interface du terminal n'ont aucun effet sur le jeu.

III. Modélisation générale

A. Serveur

Le rôle du serveur est de créer un segment de mémoire, et de créer un fils qui va gérer les requêtes TCP de connexion. Dès lors le serveur positionne une socket sur un numéro de port précisé en argument et à une adresse aléatoire, afin de recevoir des requêtes UDP de clients. Dès que le serveur reçoit une requête d'un client, il l'à traite et envoie à ce client, une réponse adaptée en fonction de l'état du segment de mémoire et de la demande du client. Enfin le serveur intercepte le signal envoyé par le fils concernant la fin d'exécution de ce dernier.

B. Fils

Le rôle du fils du serveur est de positionner une socket TCP de connexion avec une adresse aléatoire fournie par le système et à un numéro de port aléatoire, ainsi à chaque réception d'une connexion d'un client il crée un petit fils. Enfin le fils intercepte les signaux envoyés par les petits fils concernant la fin de leur exécution.

C. Petit fils

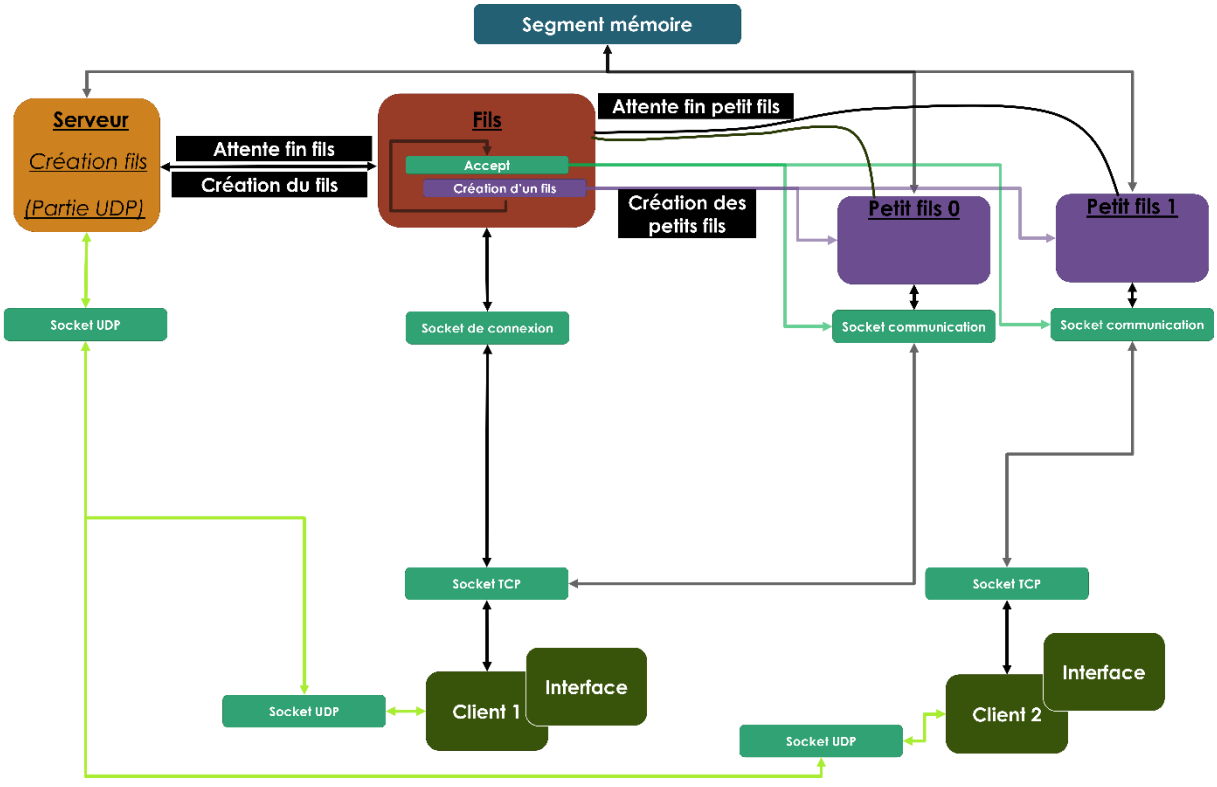
Le rôle d'un petit fils est d'envoyer des informations à son client concernant l'état d'une partie dans le segment de mémoire et l'état du scénario. Enfin lorsqu'une partie est terminée, le petit s'arrête et envoie un signal de fin à son père « le fils »

D. Client

Le rôle d'un client est double, il affiche via une interface *ncurses*, les informations de menu et de jeu à l'utilisateur et transmet les demandes aux serveur en ayant positionnée une socket UDP pour les requêtes UDP et une socket TCP, pour les requêtes TCP, et en récupérant les informations d'adresse du serveur et de numéro de port d'écoute du serveur, fournis en argument de l'exécution du code client par l'utilisateur. Lorsqu'une partie se termine, le client affiche à l'utilisateur les informations de fin de partie et s'arrête.

E. Segment mémoire

Le segment a pour rôle de contenir les informations des parties en cours, dans l'objectif de répondes aux demandes d'informations du serveur pour le client et pour le moteur de jeu et la stratégie de jeu, afin que les petits-fils puissent envoyer les informations de jeu personnalisées pour chaque client.



F. Les structures non présentées dans les parties suivantes

a) carte_t

La structure carte_t représente une carte contenue dans le répertoire « cartes ». Elle contient :

- Un identifiant
- Un nom

```
/* Carte */
typedef struct{
    int id;
    char nom[256];
}carte_t;
```

b) scenario_t

La structure scenario_t représente un scénario contenu dans le répertoire « scenarios ». Elle contient :

- Un identifiant
- Un nom

```
/* Scenario */
typedef struct{
    int id;
    char nom[256];
}scenario_t;
```

c) liste_carte_t

La structure liste_carte_t contient :

- Un tableau de « 3 » carte_t

```
/* Liste de carte */
typedef struct{
    carte_t cartes[NB_MAX_CARTE];
}liste_carte_t;
```

d) liste_scenario_t

La structure liste_scenario_t contient :

- Un tableau de « 3 » scenario_t

```
/* Liste de scenario */
typedef struct{
    scenario_t scenarios[NB_MAX_SCENARIO];
}liste_scenario_t;
```

e) jeu_t

La structure jeu_t représente les informations d'un jeu personnalisé pour un joueur elle contient :

- Un tableau à 2 dimension (15x15)
- Le nombre de vies du joueur
- La vies des adversaires
- L'argent que possède le joueur
- l'état du freeze

- L'état du unfreeze

```c
/* La structure contenant les informations sur le jeu */
typedef struct {
    unsigned char carte[15][15];   /* La carte */
    unsigned int vies;             /* Vies du joueur */
    unsigned int adv[3];           /* Vies des adversaires */
    unsigned int argent;           /* Argent */
    unsigned int freeze;           /* Etat freeze */
    unsigned int unfreeze;         /* Etat unfreeze */
} jeu_t;
```

f) ligne_scenario_t

La structure ligne_scenario_t représente les informations d'une ligne de scénario, utilisée par le petit fils pour mettre à jour le jeu du joueur et par le client pour afficher soit la carte, les informations de la fenêtre ou les autres informations, elle contient :

- Le temps d'attente d'avant la lecture de la prochaine ligne de scénario
- Le type de scénario lu
- Le message lu
- Le numéro de l'unité lu
- L'état du freeze
- L'état du unfreeze

```c
/* ligne scenario */
typedef struct{
    long temps;
    unsigned char type;
    char *msg;
    unsigned int unite;
    unsigned int freeze;
    unsigned int argent;
}ligne_scenario_t;
```

g) liste_partie_t

La structure liste_partie_t contient :

- Un tableau de « 2 » partie_t

```c
/* Liste de partie */
typedef struct{
    partie_t parties[NB_MAX_PARTIE];
}liste_partie_t;
```

IV. Réseau et exécution

A. Partie mode non connecté

1. Coté client – positionnement socket

a) Création de la socket

Le client positionne une socket UDP en précisant le domaine autrement dit la famille de protocole, *AF_INET* pour le protocole IPv4, le type *SOCK_DGRAM*, pour le mode non connecté (transmission par paquets) et le protocole, *IPPROTO_UDP*, pour le protocole UDP.

b) *Création de l'adresse réseau du serveur*

Le client crée ensuite une adresse réseau du serveur compréhensible pour par le système en précisant l'adresse et le numéro de port d'écoute du serveur fournit en argument lors de l'exécution du client.

2. Coté client – envoie requêtes

a) *Menu*

Au lancement du code client, le menu de démarrage est affiché grâce à l'interface *ncurses*, ainsi l'utilisateur à le choix entre :

- Demande de liste des cartes disponibles
- Demande de liste des scénarios disponibles
- Demande de démarrage d'une partie en précisant la carte et le scenario choisie
- Demande de liste des parties démarrées

```
                * MENU *
1. Afficher la liste des cartes
2. Afficher la liste des scenarios
3. Démarrer une partie
4. Afficher la liste des parties démarées
-> Choix :
```

Les requêtes sont directement envoyées au serveur après le choix de l'utilisateur. Toutefois, si l'utilisateur choisis de démarrer une partie, est affiché la liste de parties démarrées avec leur carte et scénario, (si l'utilisateur n'a pas demandé au préalable cette liste au serveur, la liste sera vide même des parties sont démarrées), ensuite il lui est demandé de préciser la carte et le scénario et le requête est envoyée au serveur.

```
              * LISTE PARTIES *
1. ...
2. ...
Choix carte (1.easy, 2.hard, 3.medium) : 1
Choix scenario (1.infini, 2.pognon, 3.tranquille) :
```

3. Coté client – réception requêtes

a) *Affichages*

En fonction dès requêtes envoyées par le client et de l'état du segment de mémoire les réponses sont différentes.

- Affichage des cartes

```
              * LISTE CARTES *
1. easy
2. hard
3. medium
```

- Affichages des scenarios

```
              * LISTE SCENARIOS *
1. infini
2. pognon
3. tranquille
```

- Affichages des listes de parties démarrées (cas où aucune partie n'est démarrée)

```
                * LISTE PARTIES DEMAREES *
1. ...
2. ...
```

- Affichages des listes de parties démarrées (cas avec des parties démarrées)

```
                * LISTE PARTIES DEMAREES *
1. partie1      Carte : easy
                Scenario : infini
2. partie2      Carte : hard
                Scenario : tranquille
```

- Démarrer une partie (cas où le nombre max de partie est atteint)

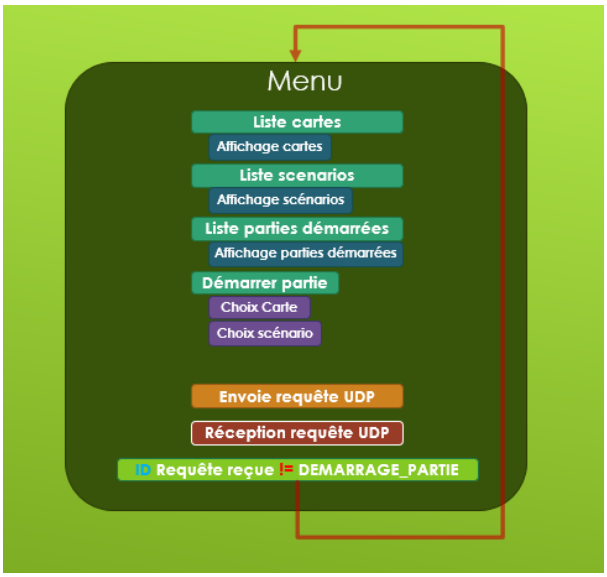
```
                * NOMBRE MAX PARTIE ATTEINT *

-> Appuyer sur 'c' pour continuer ...
```

Dans le cas où une partie démarrer un autre affichage remplace celui-ci (cf partie TCP)

b) Schéma d'exécution du client

Le client reste dans cette boucle d'envoie et de réception tant qu'il ne reçoit pas de requête dont l'identifiant est DEMARRAGE_PARTIE



4. Coté serveur – Positionnement socket

a) Création socket

Tout comme pour le client, le serveur positionne une socket UDP en précisant le domaine autrement dit la famille de protocole, *AF_INET* pour le protocole IPv4, le type *SOCK_DGRAM*, pour le mode non connecté (transmission par paquets) et le protocole, *IPPROTO_UDP*, pour le protocole UDP.

b) Création de l'adresse réseau du serveur

Le serveur crée ensuite une adresse réseau compréhensible pour par le système en précisant le numéro de port d'écoute du serveur fournit en argument lors de l'exécution du client et avec une adresse réseau aléatoire, c'est-à-dire fournit par le système.

c) Nommage de la socket

Pour nommer la socket UDP, le serveur récupère l'adresse réseau crée et la met en paramètre de la fonction *bind()*.

d) Récupération de l'adresse client

Le serveur récupère l'adresse du client qui vient de lui envoyé une requête grâce à la fonction *recvfrom()*, qui permet de récupérer une structure de type *struct sockaddr*, le serveur convertit cette structure en adresse réseau compréhensible par le système grâce à la fonction *intet_ntop()*.

5. Coté serveur – Traitement des requêtes reçues

Toutes les fonctions présentes dans cette partie et dont l'emplacement n'est pas précisé, se trouvent dans le fichier *udp_gestion_requetes.c*.

a) Requête demande liste cartes

Le serveur exécute la fonction *recuperer_liste_carte()*, (fichier.c), qui ouvre le répertoire « cartes », où se trouve les fichiers binaires et les images des cartes, grâce à la fonction *opendir()*, puis qui place la tête de lecture au début du répertoire grâce à *rewinddir()*, et lis toutes les entrées du répertoire grâce à *readdir()* tout en ne prenant pas en compte les entrées « . » et « .. » présente dans un répertoire. Enfin, les entrées récupérées dans une structure de type *liste_carte_t* puis retournées dans une requête UDP, de structure *requete_udp_t*.

b) Requête demande liste scenarios

Le serveur exécute la fonction *recuperer_liste_scenario()*, (fichier.c), qui ouvre le répertoire « cartes », où se trouve les fichiers binaires et les fichiers texte des scénarios, grâce à la fonction *opendir()*, puis qui place la tête de lecture au début du répertoire grâce à *rewinddir()*, et lis toutes les entrées du répertoire grâce à *readdir()* tout en ne prenant pas en compte les entrées « . » et « .. » présente dans un répertoire. Enfin, les entrées récupérées dans une structure de type *liste_scenario_t* puis retournées dans une requête UDP, de structure *requete_udp_t*.

c) Requête demande liste parties démarrées

Le serveur exécute la fonction *recuperer_liste_partie()* (udp_gestion_sem.c), cette fonction teste dans l'*id* des parties du segment de mémoire, quand un *id* est différent de 0, cela signifie que la partie testée est active (créée), alors l'*id*, le nom de la partie, de la carte et du scénario sont récupérées et retournées dans l'attribut *liste*_partie d'une requete UDP, de type *requete_udp_t*.

d) Requête demande démarrage partie

Dans cette partie plusieurs fonctions sont exécutées pour vérifier les informations reçues du client et l'état du segment mémoire.

- Vérification des informations reçues

Le serveur exécute la fonction *infos_parties_correctes()* (udp_gestion_sem.c), qui vérifie si le nom de la carte reçue est égale à une des cartes contenues dans le répertoire « cartes » et fait de même pour le scénario reçu en vérifiant si le nom du scénario reçu, correspond à au moins un des scénarios contenus dans le répertoire « scenarios »

- ➢ Si les informations sont conformes alors on passe à la seconde étape de vérification
- ➢ Sinon une réponse « Erreur requête » est envoyée au client

- Vérification du nombre de partie courantes (nombre de parties exécutées)

Le serveur exécute la fonction *recuperer_nombre_parties()* (udp_gestion_sem.c), qui teste si l'id des parties présentes dans le segment mémoire et retourne le résultat.

➢ Si l'*id* d'une partie est différent de « 0 » alors on incrémente de « 1 » la variable nombre *nb_partie_courantes*

- Cas où *nb_parties_courantes* vaut « 0 »

Le serveur prépare la réponse en passant l'*id* de la requête à DEMARRAGE_PARTIE, en passant le numéro de port de la socket TCP de connexion à l'attribut *num_port_connexion_tcp* de la requête et en passant « 0 » à l'attribut *num_partie*, le « 0 » permettra dans une étape plus avancée que le petit fils du serveur crée une partie dans le segment mémoire.

- Cas où *nb_parties_courantes* est différent de 0 mais inférieur à ou égale à 2

Le serveur exécute la fonction *partie_existe()* (udp_gestion_sem.c), qui vérifie si la carte et scénario reçus du client sont les mêmes que ceux des parties qui sont crées et retourne le résultat.

➢ Si num_partie_existante vaut « 0 »

Le serveur exécute la fonction *recuperer_nombre_joueur()* (udp_gestion_sem.c), qui récupère le nombre de joueur courant d'une partie grâce à la structure d'une partie qui contient un attribut *nb_joueur*_courant et retourne le résultat.

➢ Si *nb_joueur_courant* est inférieur à 4

Le serveur prépare la réponse en passant l'id de la requête à DEMARRAGE_PARTIE, en passant le numéro de port de la socket TCP de connexion à l'attribut num_port_connexion_tcp de la requête et en passant *num_partie_existante*, à l'attribut num_partie, cette variable vaut soit « 1 » ou « 2 », elle permettra au petit fils dans une étape plus avancée d'incrémenter le nombre de joueur courant de la partie lorsque le client se connectera au petits fils.

➢ Sinon

Le serveur prépare la réponse en passant l'id de la requête à DEMARRAGE_PARTIE, en passant le numéro de port de la socket TCP de connexion à l'attribut num_port_connexion_tcp de la requête et en passant « 0 » à l'attribut num_partie, le « 0 » permettra dans une étape plus avancée que le petit fils du serveur crée une partie dans le segment mémoire.

➢ Si num_partie existante vaut « 0 » et le nombre de partie courante est inférieur à 2

Le serveur prépare la réponse en passant l'id de la requête à DEMARRAGE_PARTIE, en passant le numéro de port de la socket TCP de connexion à l'attribut num_port_connexion_tcp de la requête et en passant « 0 » à l'attribut num_partie, le « 0 » permettra dans une étape plus avancée que le petit fils du serveur crée une partie dans le segment mémoire.

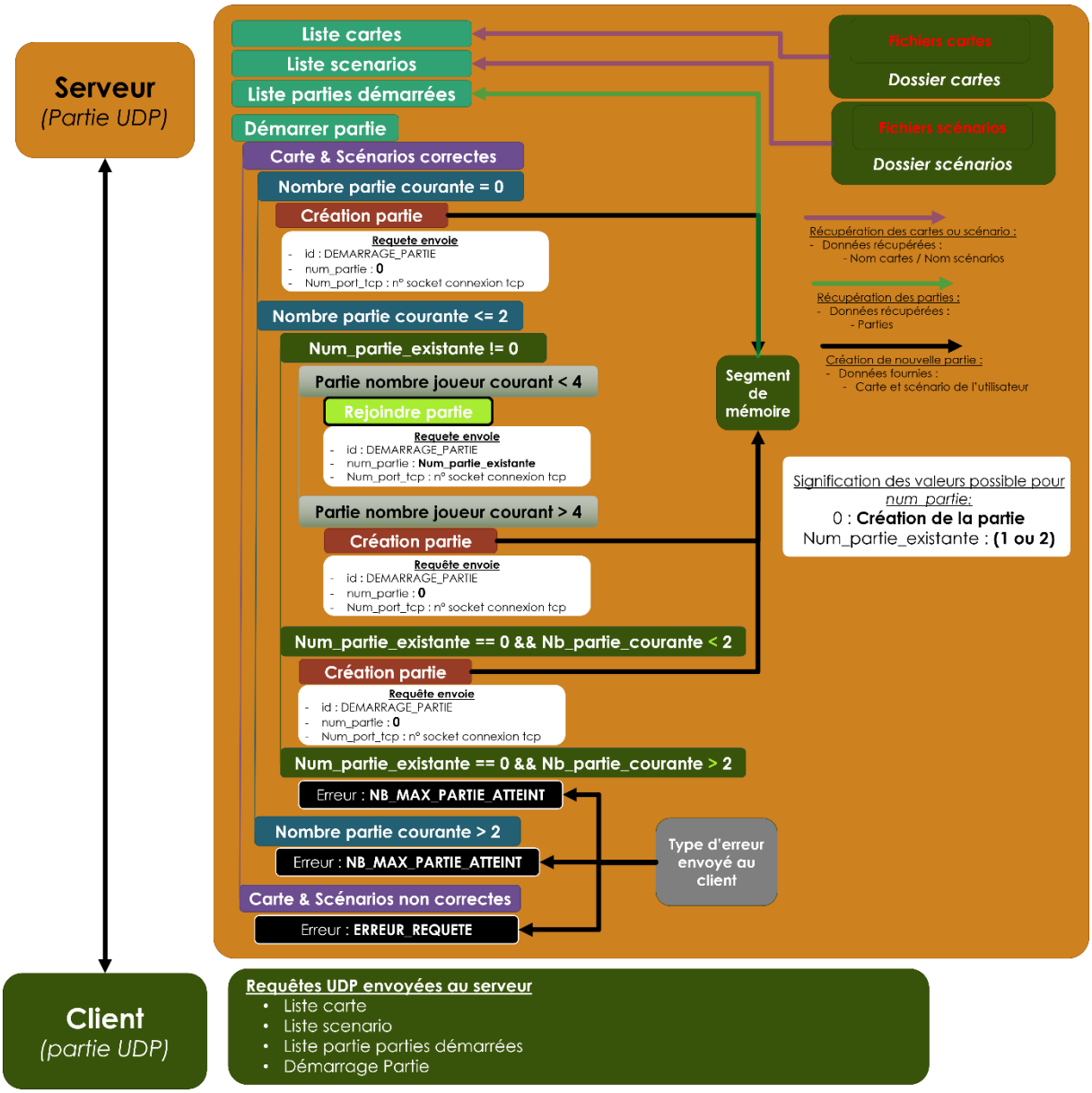
➢ Dans les cas contraire

Le serveur prépare une réponse dont l'*id* est NB_MAX_PARTIE_ATTEINT.

- Cas où *nb_parties_courantes* supérieur à 2

Le serveur prépare une réponse dont l'*id* est NB_MAX_PARTIE_ATTEINT.

Le schéma ci-dessous résume de manière graphique le traitement de requête UDP envoyées par un client, reçues par le serveur.



6. Requêtes

a) Types de requêtes

Afin de minimiser le nombre de type de requête à envoyer et recevoir, un seule type de requête est utilisé, le type *requete_udp_t*.

Cette structure contient toutes les informations qui sont échangées lors de requêtes UDP, soit :

- Un identifiant de requête

- Une structure carte de type carte_t
- Une structure scenario de type scenario_t
- Une structure liste de carte de type liste_carte_t
- Une structure liste de scenario de type liste_scenario_t
- Un numéro de port pour la connexion TCP
- Un numéro de partie

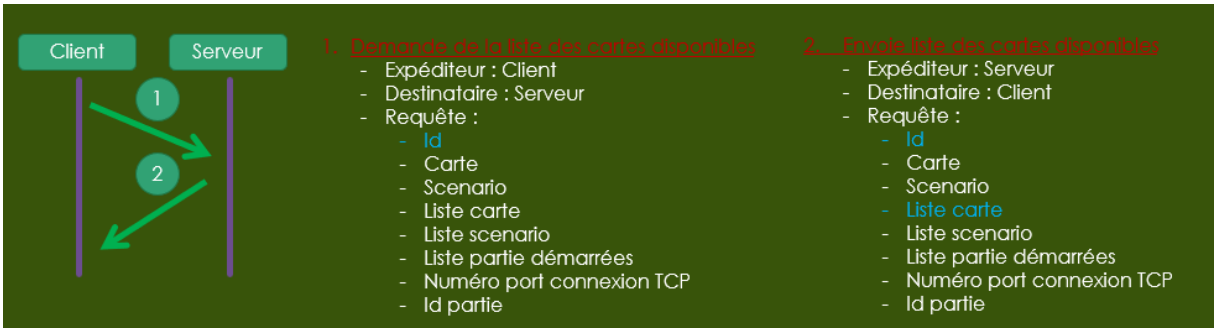
```
typedef struct{
    int id;
    carte_t carte;
    scenario_t scenario;
    liste_carte_t liste_carte;
    liste_scenario_t liste_scenario;
    liste_partie_t liste_partie;
    in_port_t num_port_connexion_tcp;
    int num_partie;
}requete_udp_t;
```

b) Diagrammes d'échanges client / serveur

Dans les diagrammes suivants les données utilisées au cours de l'échange sont écrites en cyan.

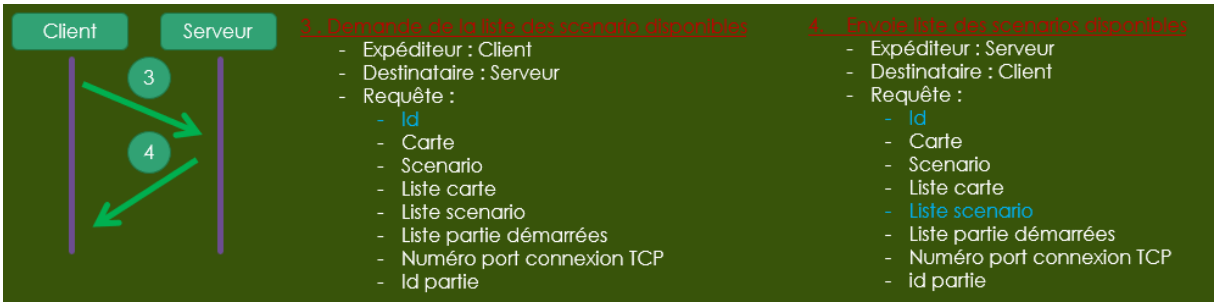
- Demande liste cartes disponibles

Le client envoie une requête dont l'identifiant correspond à la demande de liste des cartes et le serveur lui répond en envoyant une requête possédant le même identifiant accompagnée de la liste des cartes.



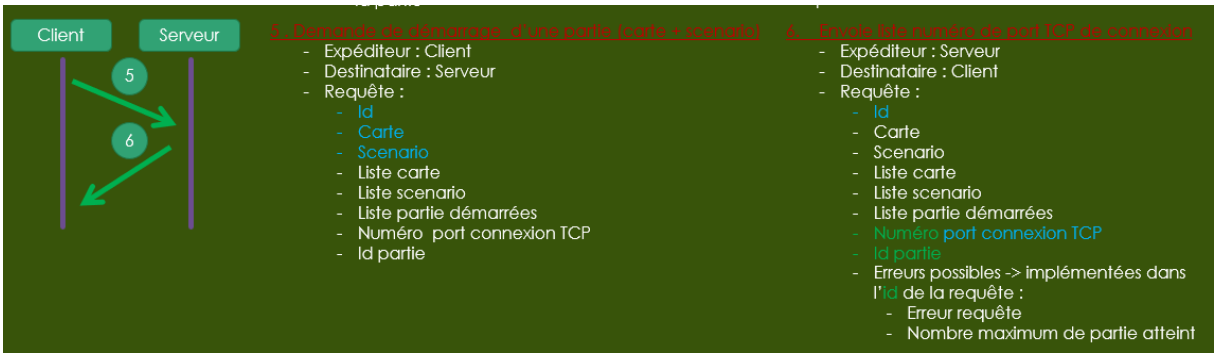
- Demande liste scénarios disponibles

Le client envoie une requête dont l'identifiant correspond à la demande de liste des scénarios et le serveur lui répond en envoyant une requête possédant le même identifiant accompagnée de la liste des scénarios.



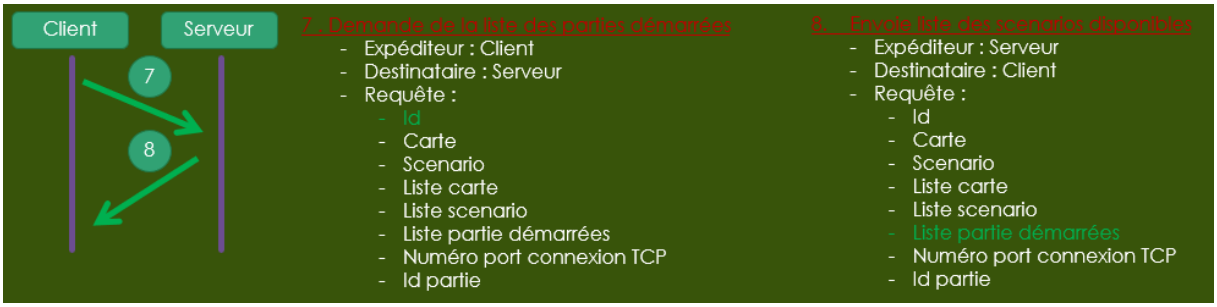
- Demande démarrage partie

Le client envoie une requête dont l'identifiant correspond à la demande de démarrage de partie accompagnée de la carte et du scénario choisie et le serveur lui répond en envoyant une requête possédant le même identifiant accompagnée du numéro de port de la socket de connexion à laquelle le client doit se connecter pour accéder à une partie et le numéro de la partie qui vaut « 0 » si le serveur a déterminé que le petit fils devra créer une partie ou « 1 » ou « 2 », si le serveur a déterminé que le client doit rejoindre une partie qui est déjà en cours.



- Demande liste des parties démarrées

Le client envoie une requête dont l'identifiant correspond à la demande de liste des parties démarrées et le serveur lui répond en envoyant une requête possédant le même identifiant accompagnée de la liste des parties démarrées qui est vide si aucune partie n'est démarrées.



B. Partie mode connecté – Connexion au fils

1. Coté client – positionnement socket

a) Création socket

Le client positionne une socket TCP en précisant le domaine autrement dit la famille de protocole, *AF_INET* pour le protocole IPv4, le type *SOCK_STREAM*, pour le mode connecté (flux d'octet) et le protocole, *IPPROTO_TCP*, pour le protocole TCP.

b) Création de l'adresse réseau du serveur

Le client crée ensuite une adresse réseau du serveur compréhensible pour par le système en précisant le numéro de port d'écoute de la socket de connexion reçue grâce à la requête UDP et l'adresse d'écoute fournit en argument lors de l'exécution du client.

c) Connexion au serveur

Le client exécute la fonction *connect()* en passant l'adresse réseau du serveur crée précédemment, cela permet débloquer la socket de connexion coté serveur.

2. Coté fils – Positionnement socket

a) Création socket

Tout comme pour le client, le serveur positionne une socket TCP en précisant le domaine autrement dit la famille de protocole, *AF_INET* pour le protocole IPv4, le type *SOCK_STREAM*, pour le mode connecté (flux d'octet) et le protocole, *IPPROTO_TCP*, pour le protocole TCP.

b) Création de l'adresse du serveur

Le fils crée ensuite une adresse réseau compréhensible pour par le système en précisant le numéro de port d'écoute du serveur reçue du père lors de sa création et avec une adresse réseau aléatoire, c'est-à-dire fournit par le système.

c) Nommage de la socket

Pour nommer la socket UDP, le serveur récupère l'adresse réseau crée et la met en paramètre de la fonction *bind()*.

d) Mise en passif de la socket

Pour mettre la socket de connexion en mode « écoute de connexion » (ou écoute de connexion), le serveur exécute la fonction *listen()*, en précisant le descripteur de fichier et le nombre de client qui peut être en attente de connexion, dans cette application ce nombre vaut « 8 » (2 parties max + 4 joueurs par partie).

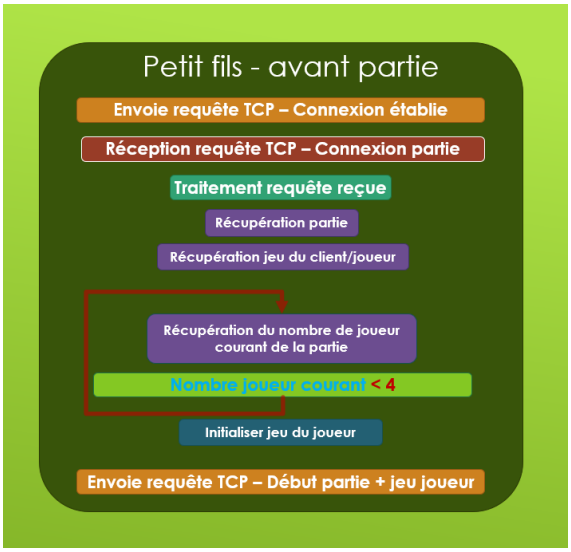
C. Partie mode connecté – Echanges avant partie

1. Coté petit fils – Schéma d'exécution

A sa création le petit fils reçoit la socket de communication du fils, qui lui permet de communiquer avec un client, celui qui s'est connecté sur le fils.

Dès sa création le petit fils envoie une requête au client pour vérifier que la connexion est bien établie entre le petit fils et client. Ensuite pour notifier que client est prêt aussi à recevoir des requête, le client lui envoie une requête de connexion à une partie avec les informations qui lui ont été fournit par le serveur (partie non connecté), soit l'*id* de partie.

Après le traitement de la requête et la mise à jour du segment de mémoire, le petit fils récupère le nombre de joueur courant d'une partie. Il attend que ce nombre ssoit égale à « 4 » pour envoyer une requête de début de partie à son client.



2. Coté petit fils – Traitement des requêtes reçues

Le petit fils récupère de la requête du client l'id de la partie fournit par le serveur et teste cet identifiant.

- Si cette id vaut « 0 »

Le petit fils crée une partie pour son client

- Si cette id vaut « 1 »

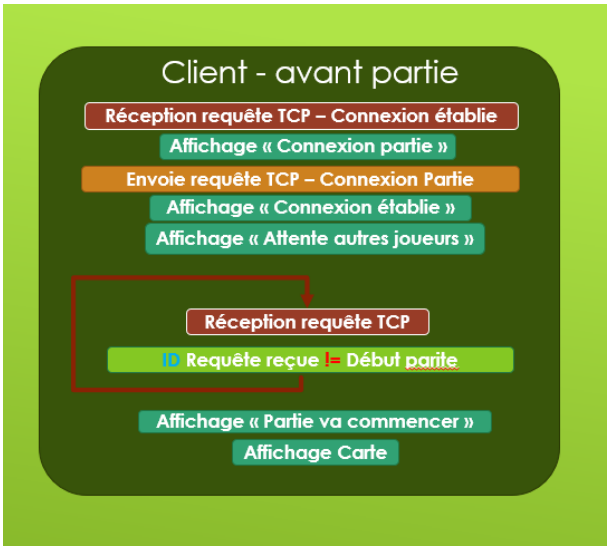
Le petit fils fait rejoindre son client à la partie 1

- Si cette id vaut « 2 »

Le petit fils fait rejoindre son client à la partie 2

3. Coté client – Schéma d'exécution

Le client reçoit la requête de connexion établie de la partie du petit fils et lui répond en envoyant une requête pour notifier qu'il est prêt à recevoir d'autres requêtes d'information, il met à jour la fenêtre d'information *ncurses*, et attend que le petit fils lui envoie une requête dont l'id est DEBUT_PARTIE puis il affiche les information de son jeu comme la carte que l'utilisateur a choisi.



4. Requêtes

a) Structure de requête

Afin de minimiser le nombre de type de requête à envoyer et recevoir, un seule type de requête est utilisé, le type *requete_tcp_t*.

Cette structure contient toutes les informations qui sont échangées lors de requêtes UDP, soit :

- Un identifiant
- Un identifiant de partie
- Un identifiant du client/joueur
- Un identifiant d'un adversaire
- Etat de connexion
- Une carte
- Un scénario
- Un jeu
- Un ligne de scénario

```
/* Structures d'une requete TCP pour le jeu*/
typedef struct{
    int id;
    int id_partie;
    int id_joueur;
    int id_adv;
    int etat_connexion;
    carte_t carte;
    scenario_t scenario;
    jeu_t jeu;
    ligne_scenario_t ligne_scenario;
}requete_tcp_t;
```

b) Diagrammes d'échanges

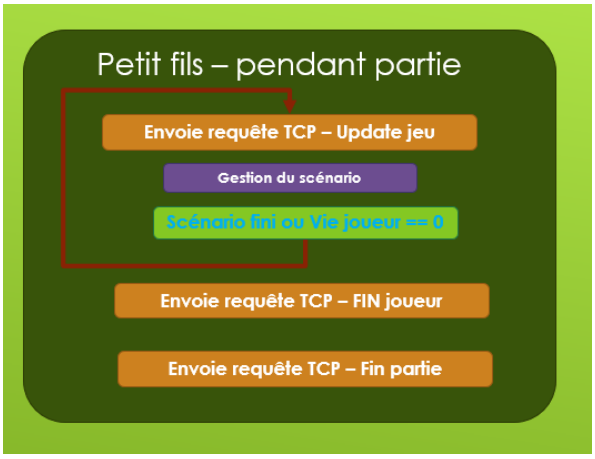
Dans les diagrammes suivants les données utilisées au cours de l'échange sont écrites en cyan.

Le petit fils envoie une requête dont l'id est CONNEXION_ETABLIE, le client répond par CONNEXION_PARTIE et enfin le petits fils répond par DEBUT_PARTIE accompagnée de la carte du scénario, et du jeu du joueur (il s'agit d'un jeu personnalisé que lui seul peut afficher sur l'interface *ncurses*).

D. Partie mode connecté – Echanges jeu

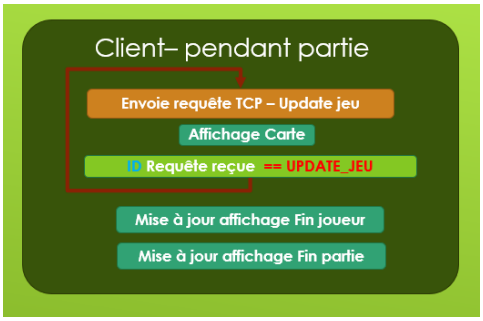
1. Coté petit fils

Dès lors que tous les joueurs sont prêts à recevoir les informations du leur jeu personnalisé (expliquées en détail dans la partie moteur). Le petit fils envoie ainsi une requête contenant le jeu du petit fils mis à jour et dont l'*id* est UPDATE_JEU. Ces requêtes sont envoyées au client jusqu'à que le client ne possède plus de vies ou que le scénario est fini. Suite à cela une requête dont l'*id* est FIN_JOUEUR est envoyé au client. Enfin le petit fils attend qu'il n'y soit plus de joueurs dans la partie pour envoyer une requête de fin de partie dont l'*id* est FIN_PARTIE, et le petit fils s'arrête.



2. Coté client

Le client reçoit son jeu qui est personnalisé et affiche sur notamment la nouvelle carte mis à jour avec les unités. Dès que l'id de la requête n'est plus UPDATE_JEU mais FIN_JOUEUR, il met à jour l'affichage puis dès qu'il reçoit la requête avec l'id FIN_PARTIE met à joueur à nouveau l'affichage, et enfin s'arrête.



3. Requête

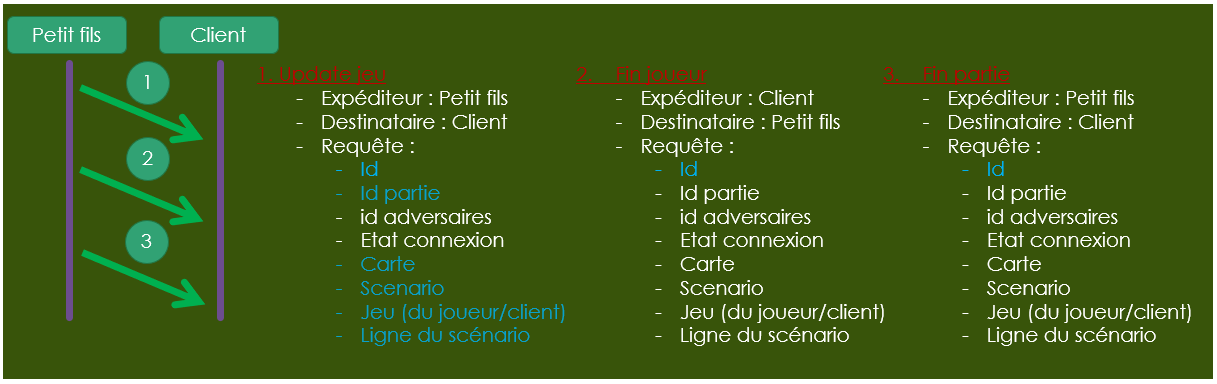
a) Structure de requête

La structure de requête échangée pendant la partie est la même à celle échangée avant la partie.

b) Diagramme d'échange

Dans les diagrammes suivants les données utilisées au cours de l'échange sont écrites en cyan.

Le petit fils envoie des requêtes de mise à jour du jeu du client avec l'*id* UPDATE_JEU et d'autres requête de pour notifier au client la fin du joueur et de la partie, ainsi le client met à jour sont affichage.



V. Processus

A. Le fils

1. Création

Le fils est créé par le serveur à l'aide de la fonction *fork()* (seveur.c), afin de libérer celui-ci de la tâche de réception des requêtes TCP de connexion. Le serveur lui passe les listes des permettant d'accéder au segment de mémoire et le numéro de port de la socket de connexion. La création du fils est visible sur le terminal qui a exécuté le code du serveur.

2. Rôle

Le rôle du fils est de créé un petit fils à chaque connexion d'un client sur la socket de connexion grâce à la fonction *fork()* (tcp_connexion.c). Chaque création de petits fils avec leur pid est visible sur terminal qui a exécuté le code serveur.

3. Fin

Le fils s'éteint lorsque **« 4 »** ce sont éteint, ainsi il envoie le signal SIGCHLD à son père, ce dernier, atteste-de sa fin à l'aide de la fonction *wait()* (serveur.c).

B. Les petits fils

1. Création

Les petits fils sont crées par le fils grâce à la fonction *fork()* (tcp_connexion.c), chaque création des petits fils sont visibles et sur terminal qui a exécuté le code serveur. Le fils fournit aux petits fils les listes qui pointes ver le segment de mémoire et la socket de communication.

2. Rôle

Le rôle des petits fils est de communiquer avec le client, mettre à jour le segment de mémoire et fournit les informations à leur client concernant leur jeu et l'état de la partie.

3. Fin

A la fin d'une partie les petits fils, envoient un signal SIGCHLD au fils qui atteste de leur fin avec la fonction *wait()*.

VI. Mémoire

A. Problématique

Le serveur a besoin de récupérer la liste des parties démarrées avec leur carte et scénario et il a besoin de connaître le nombre de joueur courant dans une partie. De plus les petits fils ont besoin de récupérer les informations de chaque joueur notamment leur jeu et mettre à jour les informations du jeu de leur client. Ainsi, pour partager ces informations, un segment de mémoire partagée est créé par le serveur à son exécution.

B. Contenu du SMP

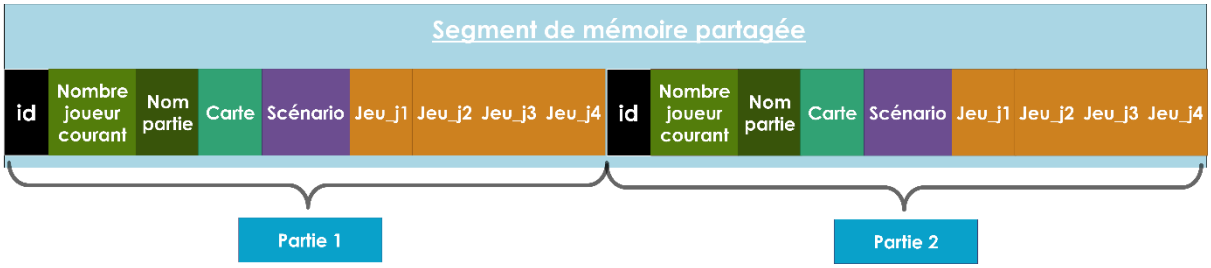
Le segment de mémoire peut contenir les informations de deux parties.

1. Structure de partie

Une partie contient toutes les informations nécessaire pour une partie de Tower Defense selon les exigences., elle contient :

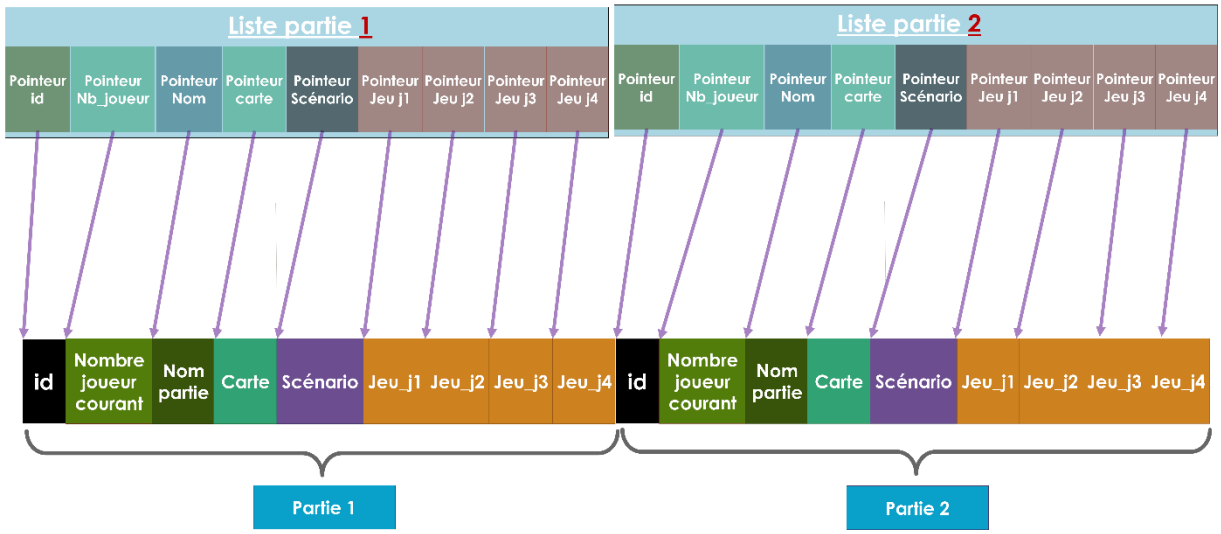
- Un identifiant - *(initialisé au début de la partie)*
- Un nombre de joueur courant – (Qui augmente à la connexion de chaque nouveau client et décrémente dès que la vie d'un client/joueur atteint 0)
- Un nom - *(initialisé au début de la partie)*
- Une carte - (initialisé au début de la partie)
- Un scénario - *(initialisé au début de la partie)*
- 4 Jeux (1 pour chaque joueur) - (initialisé au début de la partie) et (Mis à jour dans pendant la partie)

```c
/* Partie */
typedef struct{
    int id;
    int nb_joueur_courant;
    char nom[256];
    carte_t carte;
    scenario_t scenario;
    jeu_t jeu_j1;
    jeu_t jeu_j2;
    jeu_t jeu_j3;
    jeu_t jeu_j4;
}partie_t;
```



C. Accès au segment de mémoire

Pour accéder au segment de mémoire dans un chaque processus, deux listes (une pour chaque partie) sont créées lors de l'exécution du serveur, chaque élément de la liste pointe vers une adresse d'attribut dans le segment de mémoire, cela permet d'éviter de placer une pointeur à chaque lecture ou écriture dans le segment de mémoire. Ces listes sont partagées par le père au fils et aux petits fils. Les seuls utilisateurs du segment de mémoire sont le père pour la récupération d'information de parties et les petits pour la lecture d'information et la mise à jour de celles-ci.



VII. Moteur de jeu

A. Ouverture des fichiers binaires

Nous disposons de 3 scénarios qui sont contenus dans un dossier **scenarios**. Chaque scénario est un fichier binaire. Pour lire ces fichiers binaires, il faut indiquer le **pathname**, c'est-à-dire le chemin à suivre pour obtenir la localisation du fichier. On ouvre ensuite ces fichiers en mode lecture.

B. Lecture des scénarios

Chaque scénario est constitué d'une description. La description correspond à une taille (size_t) suivie d'une chaine de caractères variable de la description. En lisant tout d'abord la taille, on peut ainsi déterminer le nombre de caractères de la chaine et donc passer la description pour pouvoir ensuite lire la suite. Etant donné qu'on ne connait pas le nombre d'octets des scenarios, pour pouvoir déterminer une condition d'arrêt de la lecture du scénario, on utilise le fait que tant que le nombre d'octets lu est différent de 0, on continue de lire le scénario.

Lire une ligne d'un scénario correspond à faire cela :

- Lire le temps (long)
- Lire le type (unsigned char)
- Lire la donnée
    - Lire char[255] (Si type = 0)
    - Lire unsigned int (1 à 5) (Si type = 1)

Lire unsigned int (0 à 65535) (Si type = 2 ou type = 3)

C. Matrice

Nous avons à disposition une matrice de taille 15x15 qui contient des unsigned char. Cette matrice représente la carte du jeu. Elle indique le positionnement sur la carte, des unités, du fort, du point de départ des unités lancées par les adversaires (les chiffres 1 à 3) et du point de départ des unités correspondant au scénario (la lettre O). Les cases vides sont les 0. Cette matrice est de même dimension que la fenêtre carte dans ncurses.

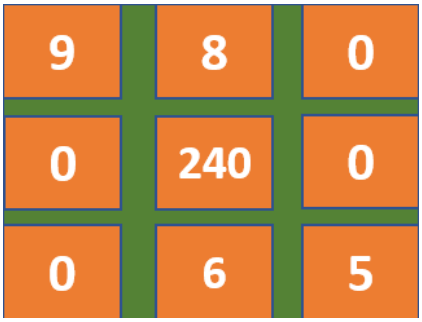
D. Déplacement des unités

Il existe 5 unités différentes, dont chacune est caractérisé par:

- Soldat : 240 (CASE_SOLDAT)
- Commando : 241 (CASE_COMMANDO)
- Véhicule : 242 (CASE_VEHICULE)
- Lance-Missile : 243 (CASE_MISSILE)
- Char : 244 (CASE_CHAR)

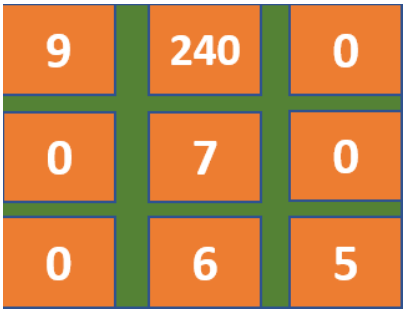
Tout au long d'une partie, les unités vont apparaitre au point de départ du scénario, c'est-à-dire le point O qui est caractérisé par l'entier 254 dans la matrice. Pour qu'une unité se déplace, on regarde tout d'abord les éléments de la matrice qui sont juste au-dessus, en dessous, à droite et à gauche de celle-ci qu'on stocke dans un tableau de 4 entiers. Après cela, on détermine la valeur la plus grande autour de l'unité en triant le tableau et en récupérant la dernière valeur de celui-ci. L'unité se déplace alors sur la case dont la valeur est la plus grande. L'ancienne valeur de la case ou était l'unité, est alors remplacée par la valeur que l'unité a remplacée moins 1.

On peut prendre l'exemple suivant :



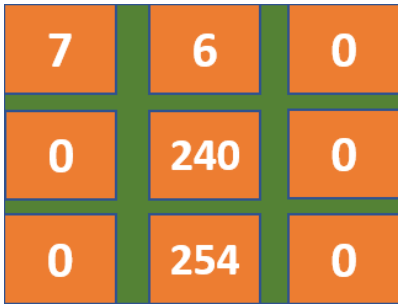
On observe que le soldat, c'est-à-dire le nombre 240 est au centre. Les éléments aux alentours du soldat sont 8, 0, 0 et 6 puisqu'on ne prend pas en compte les diagonales. La valeur la plus grande est alors 8, et donc cette valeur sera égale à 240. Et la valeur de la case du milieu donc celle ou le soldat était auparavant sera elle égale à la valeur la plus grande donc 8 moins un, donc 7.

Voici ce que l'on obtient après déplacement du soldat d'une case :



Cette méthode de déplacement des unités par la recherche de la valeur la plus grande est efficace. Cependant, il faut rajouter certaines conditions pour qu'elle puisse fonctionner dans tous les cas. En effet, il y a certains cas, ou celle-ci ne fonctionnera pas correctement ou bien pas de la manière dont on le souhaite.

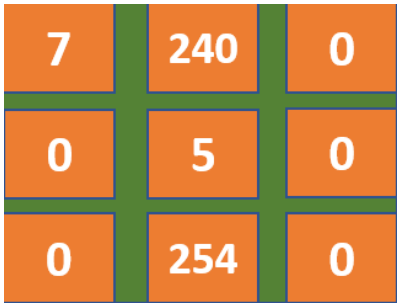
Prenons l'exemple suivant :



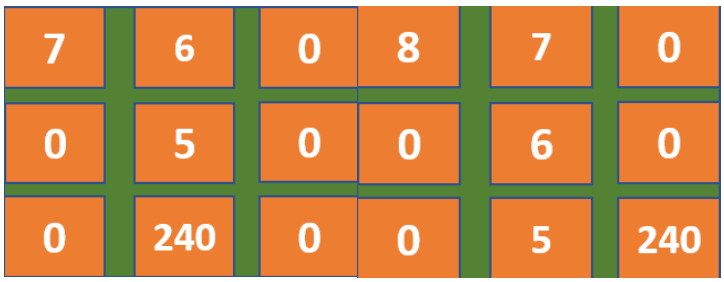
La case 254 correspond à la case ou les unités du scénario apparaissent. Le soldat va dans cet exemple, se déplacer d'une case. On applique la même méthode, c'est-à-dire qu'on recherche la valeur max aux alentours du soldat. Ici, cela correspond à 254. Nous ne voulons pas que le soldat se déplace sur cette case puisqu'il est apparu ici. Cela reviendrait à ce qu'il fasse demi-tour. Dans ce cas, on rajoute alors la

condition suivante : Si jamais la valeur la plus grande est 254, alors se déplacer sur la 2ème valeur la plus grande, donc 6. En général,

On obtient finalement ceci :



Si jamais l'unité est présente sur une arête de la carte, la procédure est quasiment la même sauf qu'ici , on ne déterminera la plus grande valeur d'entre trois valeurs. Si l'unité est sur un sommet de la carte, il n'y aura que 2 valeurs à étudiées.



VIII. Concurrence

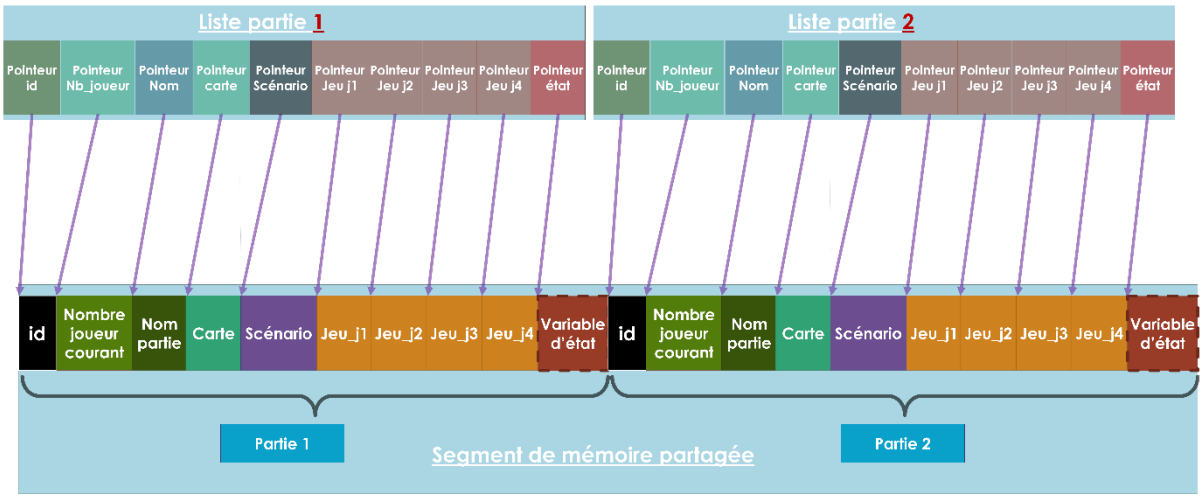
La gestion de la concurrence n'est pas implémentée dans cette version de l'application, toutefois, elle a été modélisée.

A. Problématique

Le segment de mémoire étant partagée par le serveur et les petits fils, il risque d'y avoir des problèmes de concurrence, notamment lorsqu'un petit fils met un jour le segment, par exemple il crée une deuxième partie et le serveur répondant à un client qu'il n'y a qu'une seule partie en mémoire. Par conséquent il est nécessaire d'implémenter un système de gestion de la concurrence.

B. Solution modélisée

Une solution alternative aux sémaphores a été étudiée, il s'agit d'intégrée aux segment de mémoire une variable représentant l'état d'une partie du segment de mémoire. C'est-à-dire, Si un processus souhaite mettre à jour une des deux parties du segment, il positionne la variable d'état du segment à « 1 » et dès qu'il a finit la remet à « 0 ». Pour récupérer une information du segment il n'est pas nécessaire de mettre à jour cette variable d'état.



IX. Compilation et exécution

A. Compilation

➔ make

B. Exécution

1. Serveur

➔ ./serveur n° de port d'écoute

2. Client

➔ ./client adresse du serveur n° de port d'écoute

3. Remarques

- Exécution du code serveur avec 1 terminal
- Exécution du code client avec 1 terminal
- L'exécution du code client peut s'être exécuter plusieurs
- Dans cette version, dès 4 clients démarre une partie et se connecte en TCP sur la socket de connexion, le fils s'arrête.
- Pour tester l'application on peut alors lancer 2 clients et démarrer 2 parties différentes, et lancer 3ème client pour vérifier que 2 parties sont bien démarrer. Toutefois, pour tester une partie, il est par conséquent nécessaire que 4 quatre clients démarrent la même partie
- Il est nécessaire d'exécuter le code serveur avant les codes clients.

4. Arrêt du serveur

➔ CTRL + C (SIGINT)

5. Suppression des fichiers « .o », des exécutables et du segment de mémoire

➔ make clean