Algorithmique répartie et distribuée

encadré par

François Pecheux

Rapport de projet

réalisé par

Jérôme Bonacchi
Laurent Dang Vu
Thomas Genin
Najwa Moursli

Mathématiques Appliquées et Informatique Numérique
2e année de cycle ingénieur
Polytech Sorbonne
2019-2020
Paris, France

Table des matières

1

Introduction

1.1 Explications du projet

Ce rapport est consacré à l'explication, ainsi qu'aux commentaires, concernant ce projet qui porte sur la réalisation du jeu Perudo en tant que jeu en réseau. Cette réalisation utilise le logiciel Godot en combinaison avec un programme de communication réseau *UDP* écrit en C.

1.2 Le Perudo

1.2.1 Les principes du Perudo

Le Perudo est un jeu de dés propriétaire qui s'inspire fortement de jeux plus anciens comme le Dudo. Pour jouer à ce jeu, chaque joueur a besoin d'un gobelet opaque et de cinq dés à six faces presque normaux. Seul le 1, qu'on appelle Paco, a une forme différente et a le rôle en quelque sorte d'un joker car il a toutes les autres valeurs à la fois). Le jeu consiste à ce que chaque joueur remue ses dés dans son gobelet, regarde le résultat obtenu en ignorant celui des autres joueurs, puis parie sur le nombre de dés au total ayant une certaine valeur. Ce jeu fait donc appel à des statistiques rudimentaires, conditionnées par ce qu'on sait sur ses propres dés, et par ce que les autres joueurs ont parié. Une bonne dose de bluff est également nécessaire pour tromper ses adversaires.

1.2.2 Le déroulement d'une partie de Perudo

On tire d'abord au sort qui va commencer. Chaque joueur prend ensuite possession d'un gobelet et de cinq dés, puis secoue le gobelet avec les dés dedans et le pose à l'envers, de manière à ce que les dés aient des valeurs aléatoires et restent sous le gobelet, donc invisibles (les gobelets sont opaques). Chaque joueur peut alors regarder sous son gobelet et uniquement le sien. Chaque joueur à tour de rôle, dans le sens des aiguilles d'une montre, va pouvoir faire des paris sur le nombre de dés d'une certaine valeur. Le premier joueur fait une enchère (celle-ci ne doit pas porter sur le nombre de Pacos). Le suivant peut :

- surenchérir
  - en pariant uniquement une plus grande valeur,
  - en pariant plus de dés avec la valeur de son choix,

  - en pariant le nombre de Pacos (en divisant par 2 le nombre de dés en arrondissant à l'entier supérieur) ;
  - pour revenir d'un pari sur les Pacos à un pari normal, il faut multiplier par deux le nombre de dés et ajouter un ;
- considérer que le pari est erroné (c'est-à-dire que le nombre de dés annoncés est supérieur à la réalité).

Dans le dernier cas, tout le monde révèle ses dés. Si le pari était juste (c'est-à-dire s'il y a un nombre de dés de la valeur choisie supérieur ou égal au pari), le joueur qui a douté perd un dé, sinon c'est celui qui a fait le pari erroné qui en perd un. Le joueur qui vient de perdre un dé est le nouveau premier joueur à annoncer son futur pari. Si celui qui vient de perdre un dé a perdu son dernier dé, il a perdu, et c'est le joueur suivant, dans le sens des aiguilles d'une montre, qui démarre. Le jeu se termine quand tous les joueurs, sauf un (le vainqueur), ont perdu tous leurs dés.

Le Palifico est un tour de jeu particulier, qui se produit quand un joueur n'a plus qu'un seul dé. Celui-ci est le premier joueur à annoncer son futur pari. Les règles sont alors modifiées pour cette manche là uniquement : les Pacos ne sont plus des jokers et la valeur de dé pariée par celui qui commence ne peut plus être changée. De plus, celui qui commence peut parier sur les Pacos puisqu'ils sont alors des valeurs normales.

1.3 Introduction au concept de jeu en réseau

Avant de commencer l'explication de notre projet, il est important de rappeler certaines notions fondamentales pour comprendre sa conception et sa réalisation.

1.3.1 L'adressage, les protocoles et la communication

Il est important de retenir que le modèle de communication *OSI* (*Open Systems Interconnection*) à été développé afin de faciliter la compréhension des utilisateurs du réseau, il s'agit de la représentation d'un ordinateur en couches communicant entre elles. La figure 1.1 présente ce modèle en détails. Pour qu'une machine puisse communiquer sur un réseau, elle doit disposer d'une adresse *IP* (*Internet Protocole*) qui est une suite de nombres codés en général sur 4 octets (pour IPv4). Chaque machine en possède une permettant de l'identifier sur le réseau. Afin d'envoyer et de recevoir des données, il faut aussi que chaque machine parle le même langage, et l'ensemble des règles qui définissent ce langage sont les protocoles.

Il sont utilisés dans les différentes couches intermédiaires du modèle *OSI*. En tant qu'utilisateur, nous ne voyons que la couche applicative, nous ne sommes jamais en contact direct avec les couches inférieures qui permettent de traiter le texte, de le traduire en langage binaire, de l'envoyer, etc.

Les protocoles utilisent des numéros de port qui sont associés à des programmes particuliers qui écoute sur ces canaux de communications. Lorsque que l'on envoie ou reçoit une information grâce au réseau, le logiciel doit être capable de dire pour quelle application est destinée l'information.

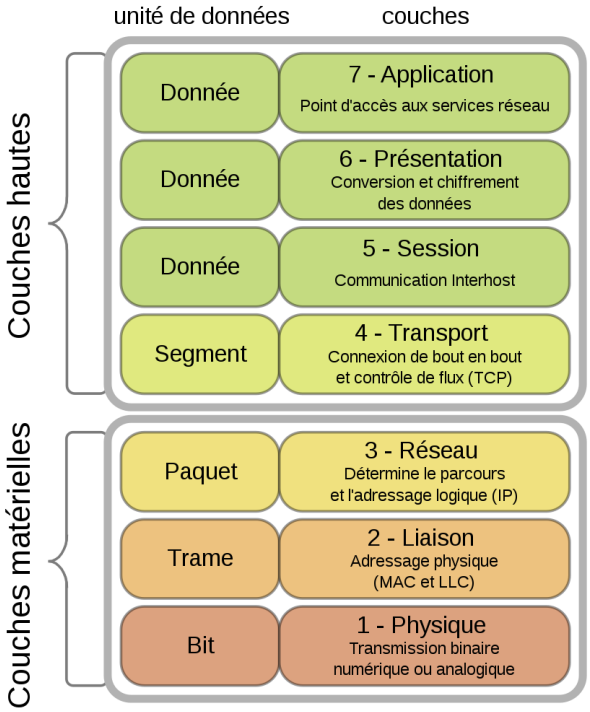

FIGURE 1.1 – Modèle *OSI* (fr.wikipedia.org/wiki/Modèle_OSI)

Lors de son passage entre chaque couche, une information supplémentaire est ajoutée à la donnée initiale : c'est l'encapsulation (cf. figure 1.2). Lorsque le message passe dans l'autre sens, dans les couches respectives de l'autre machine, ces informations sont analysées et supprimées du message. Ces informations supplémentaires sont appelées des en-têtes et permettent de transmettre des informations sur le protocole qui a été utilisé par chaque couche.

1.3.2 Le *UDP*

Le *User Datagram Protocol* est un protocole de transmission de données (sous forme de datagrammes) correspondant dans le modèle Internet à la couche transport (comme le *TCP*), intermédiaire de la couche réseau et de la couche session. Contrairement au *TCP*, ce protocole ne nécessite pas de mécanisme d'établissement de liaison et il est non-fiable car il expose le programme qui l'utilise aux problèmes éventuels de fiabilité du réseau ; ainsi, il n'existe pas de garantie de protection quant à la livraison, l'ordre d'arrivée, ou la duplication éventuelle des datagrammes. Notre choix d'utilisation s'est guidé vers *UDP* car il est utile pour transmettre rapidement de petites quantités de données et que le programme a pour but d'être utilisé sur un petit réseau locale où il y a très peu de risques de pertes ou d'altérations des données communiquées.

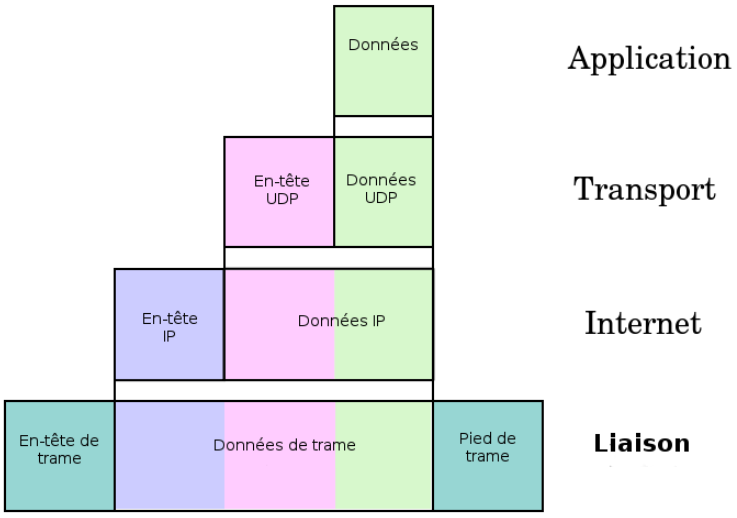

FIGURE 1.2 – Encapsulation *UDP* (fr.wikipedia.org/wiki/Encapsulation_(réseau))

1.3.3 Le serveur

Pour la réalisation d'un jeu en réseau, il faut dans un premier temps implémenter le serveur. Celui-ci permet à plusieurs utilisateurs de se connecter en réseau via la même plate-forme et à intéragir entre eux.

1.3.4 Les *sockets*

Les *sockets* sont des flux de données permettant à des machines locales ou distantes de communiquer entre elles via des protocoles tel le *UDP*.

1.3.5 Les *threads*

Les *threads* (ou fil d'exécution) permettent de faire de la programmation multitâche, c'est-à-dire de permettre au programme d'exécuter en parallèle du fil d'exécution originel et principal d'autres fonctions. Les *threads* sont similaires aux processus : ils regroupent des instructions à exécuter. Néanmoins, un *thread* est créé par un processus et partage son contexte d'exécution avec son processus père : il a donc accès à toutes variables de celui-ci. Puisqu'il n'y a pas de changement au niveau de la mémoire virtuelle lors de l'exécution d'un *thread*, le passage d'un contexte d'exécution à un autre (*context switch*) entre deux *threads* est très rapide. Ipso facto, cela permet d'exécuter en même temps plusieurs actions, sans pour autant dupliquer le processus père. Ce dernier cas correspondrait à un *fork* : le procédé consistant à dupliquer entièrement un processus. Nous n'avons pas choisi cette solution car elle est beaucoup plus lente, et donc désuète. Les *threads* peuvent être créés avant même que leur utilisation soit requise : on parle alors de

threadpool. En effet, la création de *threads* pendant l'exécution du programme pourrait ralentir celui-ci.

2

Développement du projet

2.1 Présentation de l'architecture du système

En bref, le projet est composé d'un serveur sur le *PC* central et de 4 clients sur 4 ordinateur annexes. Les informations transitent toutes par le serveur qui correspond à la racine d'un arbre avec 4 fils. Par exemple, lorsqu'un joueur se connecte, cette information est envoyée au serveur qui la transmet aux autres joueurs connectés. Le code du serveur est en C et le jeu fonctionne sur l'application Godot qui utilise le langage gdscript très inspiré de langages existant comme Python, TypeScript ou encore C++. La même application Godot sera lancée sur chacun des 4 ordinateurs que l'on peut nommer : Nord, Est, Sud et Ouest.

2.2 Le serveur

Dans cette section, nous aborderons la partie du programme concernant l'implémentation du serveur. Nous expliquerons brièvement l'ossature du processus, les différentes fonctions qui le compose ainsi que le format des messages pour la communication avec les clients.

2.2.1 L'ossature

Dans un premier temps, la structure de notre programme serveur se composent des parties suivantes. Tout d'abord, il faut créer la *socket*, par la suite il faut créer l'interface de connexion. C'est sur cette même interface que le serveur va « écouter » pour permettre d'établir la liaison entre les *sockets* des clients et l'interface sur laquelle le serveur acceptera les adresses. Ensuite, une partie concerne l'écoute et la connexion des clients afin d'établir les connexions entrantes. Finalement, il faut fermer la *socket*.

Plus précisément dans notre code, il nous faut créer une structure `client` qui va nous permettre de stocker les informations concernant les joueurs qui se connecteront au serveur. Par la suite, dans la fonction principale on créé la *socket*, puis on récupère les informations (n'importe quelle adresse) pour les connecter à la *socket* avec `sockaddr` qui est une référence générique pour les appels systèmes ; pour se faire on utilise `bind` qui lie une *socket* avec une structure `sockaddr`. Ensuite, on « écoute » le nombre de connexion avec `listen`, qui définit le nombre maximal de connexions (le serveur et les quatre joueurs), et renvoie un message d'erreur si un joueur tente de

se connecter passé cette limite. In fine, on récupère l'adresse *IP* (*Internet Protocol*), le numéro de port, le nom et la position (Nord, Est, Sud, Ouest) de chaque joueur qui se connecte. Nous avons également implémenter le fait que le premier joueur de la partie soit tiré au sort, au lieu que ce soit le premier joueur à s'être connecté. Pour finir, on boucle sur les différents cas possibles d'action des joueurs qui seront explicités par la suite.

2.2.2 Les variables

Le code contient deux structures : une structure _client et une structure _pari qui contient la valeur de dé et le nombre d'occurrences de cette valeur parié ainsi que celui a fait ce pari. Nous enregistrons également dans des variables globales le dernier pari émis, la liste des clients, le nombre de clients et l'état du jeu.

2.2.3 Les différentes fonctions

Tout d'abord, les fonctions `sendMessage` et `broadcast` servent à envoyer des messages du serveur aux joueurs, respectivement à un joueur en particulier d'une part et à tous les joueurs d'autre part. De plus la fonction `broadcast` possède une option pour envoyer aux quatre joueurs ou seulement aux joueurs qui n'ont encore pas perdu.

On retrouve également des fonctions relatives à la gestion des dés. En effet, la fonction `init_table_des` met à jour la table des dés globale (la variable globale `tableDes`), c'est-à-dire contenant les occurrences de chaque valeur parmi tous les dés des joueurs. Puis, la fonction `totalDes` qui renvoie le nombre total de tous les dés de tous les joueurs. Enfin, la fonction `clean` permet de réinitialiser à 0 l'ensemble des dés à chaque nouvelle manche.

À la suite du pari d'un joueur, `get_id_parieur` renvoie l'identifiant de l'émetteur du pari parmi *enListe[id]* (les joueurs encore en jeu). Dès que le parieur et les autres joueurs sont identifiés, `perduPasPerdu_parieur` récupère l'identifiant du parieur grâce à la fonction précédente. Ensuite, elle utilise le nombre de dés obtenus par `totalDes` pour tester le nombre de dés du parieur.

La fonction `perduPasPerdu_joueur` fait de même avec le joueur dont les dés sont l'objet du pari. Ces deux fonctions retournent 1 si un des deux a perdu sinon 0. Lorsqu'un joueur pense que le pari du joueur précèdent est erroné le serveur vérifie le pari grâce à la fonction `verification_menteur`. Celle-ci vérifie le nombre de dés du dé parié dans tout le jeu.

Si la face choisie n'est pas un Paco, alors il faut les ajouter pour le décompte au nombre de dés avec la même face que celle pariée. À la suite de ces calculs deux scénarios se présentent :

- Le parieur a raison et le menteur a tort : si le nombre de dés est inférieur ou égal, le joueur qui met en doute le dernier pari perd un dé (`resultats=0`).
- Le menteur a raison et le parieur a tort : c'est le joueur qui a fait le pari qui en perd un dé (`resultat=1`).

Finalement, la fonction `pacifico` permet de savoir si on est au tour suivant en Pacifico par la condition :

— `perdant == 0 && udpClients[idClient].nbrDes == 1` correspondant au cas où le joueur qui possède un dé et qui doute du pari est perdant ;
— `perdant == 1 && udpClients[idParieur].nbrDes == 1` correspondant au cas où le parieur possède un dé est perdant.

Dans tous les cas, il y a `broadcast` du message du commencement du tour pacifico.

2.2.4 Le traitement des différents cas

Lors de l'exécution du jeu, les joueurs (quatre directions chacune identifiée à un identifiant (North ⟵ 0, South ⟵ 2, East ⟵ 1 et West ⟵ 3) ont le droit de réaliser plusieurs actions chacun leur tour. Il faut dès lors que le serveur soit en continuelle attente de messages de la part des joueurs pour pouvoir les recevoir, les traiter et renvoyer des informations aux joueurs pour l'affichage. On recense alors les cas suivants d'après la première lettre des messages.

Le cas **C** correspondant au cas où un joueur se connecte. Juste après, le serveur va partager à tous les autres joueurs déjà connectés, via le message **J**, l'adresse *IP*, le port, le nom et la direction (point cardinal) des joueurs actuellement connectés. Si le nombre de joueurs à atteint la limite maximale (qui est de quatre joueurs), alors le serveur lance le jeu. Finalement, le joueur qui commence la manche est choisi aléatoirement par le serveur, en envoyant le message **S direction**.

Le cas **D** indiquant le lancé de dés qu'un joueur vient d'effectuer.Il *broadcast* **T direction** au voisin de gauche de celui qui a parié. C'est à son tour d'agir : soit de surenchérir, soit de mettre en doute le dernier pari. Soit il surenchérit et alors **P direction valeur nombre_de_dés** est envoyé. Soit il pense que le pari est erroné : le cas **M** est envoyé au serveur qui le *broadcast*.

Le cas **P** où le serveur reçoit le pari d'un joueur. Dans ce cas là, le serveur enregistre les informations du pari pour plus tard et le *broadcast* à tout le monde. De plus, il signale à tous les clients que le voisin de gauche du parieur est le prochain joueur qui doit parier.

Le cas **M** où un des joueur dit menteur : où un joueur doute du pari fait par le joueur précédent. Alors, le serveur va appeler la fonction de `vérification` pour savoir si le pari était correct ou non. À la fin de la vérification, il informe tous les clients du perdant et du gagnant en envoyant respectivement **W IdGagnant** et **L IdPerdant**. Si en plus le perdant perd son dernier dé, le serveur envoie **Q IdPerdant** à tout le monde pour annoncer que le perdant a quitté la partie. Le serveur actualise le nombre de dés du perdant et vérifie si celui-ci est dans le cas du Palifico. Puis, il annonce le premier joueur qui doit déclarer son pari.

2.3 Le client

Dans cette section, nous aborderons la partie du programme concernant l'implémentation du client. Nous expliquerons brièvement l'ossature du processus, les différentes fonctions qui le compose ainsi que le format des messages pour la communication avec les clients.

2.3.1 Les différentes fonctions

Pour fonctionner, le jeu a besoin que le client puisse recevoir et envoyer des informations au serveur, et cela en même temps : c'est pourquoi les *threads* sont nécessaires. Au lancement du jeu Godot, plus précisément dès que la scène `ControlMenu` est instanciée, un serveur *UDP* interne à Godot est créé par un *thread* pour écouter continuellement ce que dit le serveur.

2.3.2 Le traitement des différents cas

De la même manière que le serveur, le client reçoit des messages qu'il distingue d'après leur première lettre.

Le client envoie le message **C** suivi de son identifiant au serveur pour se connecter à une partie de jeu. Le serveur accepte, *broadcast* un message **J** afin de communiquer à tout le monde la liste des joueurs actuellement connectés. Ensuite, le serveur lance le jeu en choisissant aléatoirement le joueur qui va commencer la partie, puis *broadcast* un message **S direction** à tous les clients. Ensuite, chaque client lance les dés et transmet leur valeur au serveur en envoyant le message **D**. Lorsque le client reçoit le message **L** signifiant que le client dont l'identifiant est dans le message a douté du pari à tord, il diminue de 1 le nombre de dés du joueur concerné. Lorsque le client reçoit le message **Q** signifiant que le client dont l'identifiant est dans le message a perdu, il indique dans le tableau `endGame` que le joueur concerné a perdu. Lorsque le client reçoit le message **F**, le jeu se termine.

2.4 L'interface graphique

Selon les messages que le client reçoit du serveur, il peut activer ou faire apparaître des boutons, des fenêtres, etc. Les éléments de l'affichage sont créés statiquement à la création de la scène, mais ils sont rendus cachés ou visibles via l'appel à différentes fonctions qui dépendent du message reçu. L'interface graphique est constituée de 4 scripts et 4 scènes associées.

`ControlSplash.gd` correspond au script permettant de mettre en place l'écran de lancement en changeant de scènes telles que `controlMenuNode`, `controlGameNode` et `controlOptionsNode` et les affichant au moment voulu gre au bouton `ButtonVersMenu` qui, lorsqu'il est pressé appelle la fonction `_on_ButtonVersMenu_pressed` qui permet d'accéder au menu. L'adresse *IP* de chaque joueur se voit affectée un entier correspondant aux différentes directions possibles du client Nort ⟵ 0, East ⟵ 1, South ⟵ 2 et West ⟵ 3.

`ControlMenu.gd` permet faire communiquer le serveur central et le client Godot par la fonction `createNetworkThread` appelant la fonction`_thread_network_function`, utilisant les données du client. On prépare une *socket* (ici variable globale) permettant au client d'écouter sur un port en particulier les informations que le serveur central lui enverra. D'autres fonctions et boutons sont présents pour interagir avec le serveur comme `sendMessage`, `sendMessageToServer` et `_on_ButtonTestReseau_pressed` qui test le protocole *UDP*. Ensuite, deux boutons sont utilisés dans le menu, `ButtonJouer` envoie le message **C** au serveur pour ce connecter et permet de changer de scène vers

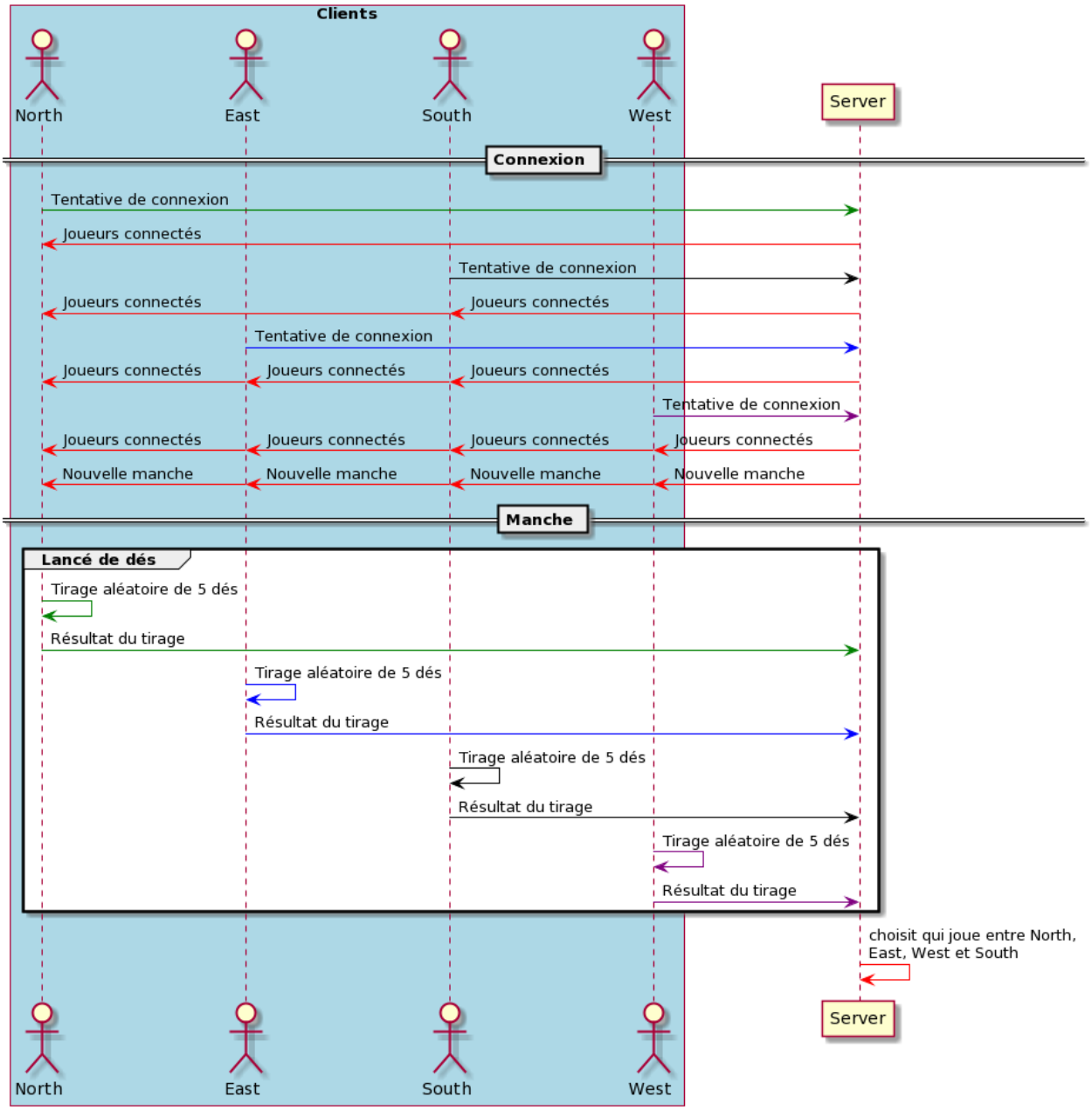

FIGURE 2.1 – Diagramme de séquence (1/2)

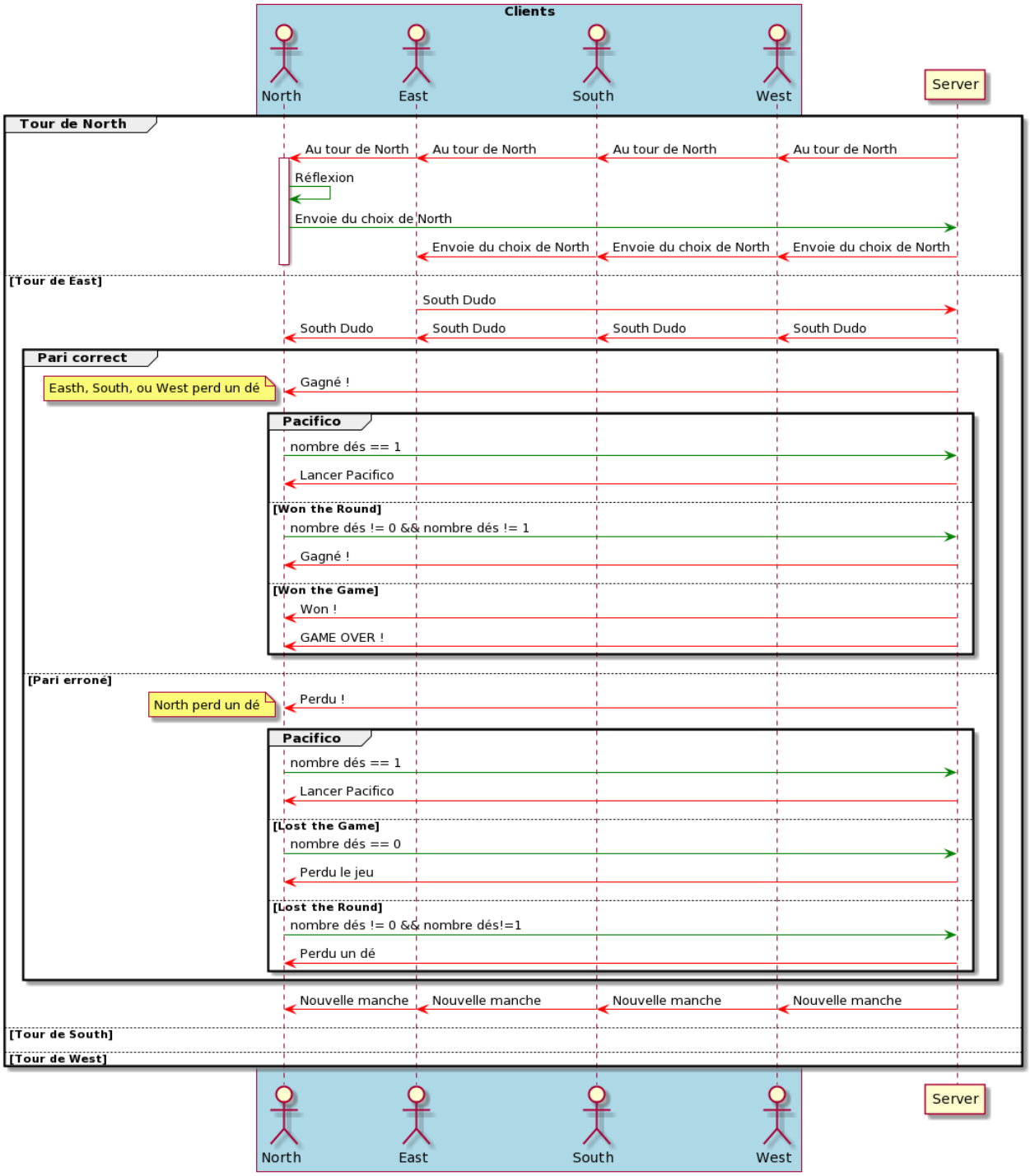

Figure 2.2 – Diagramme de séquence (2/2)

`controlGameNode`, la scène pour jouer une partie. De plus, une zone de texte permet d'entre son nom d'utilisateur. Sur la figure 2.3, nous pouvons voir le menu principal de l'application.

Dans `ControlGame.gd`, la fonction `_networkMessage` permet d'afficher l'ensemble des messages envoyés par le serveur et récupérés par la *socket* de `ControlMenu`. Quand le serveur émet le message qu'un nouveau joueur s'est connecté, `newPlayer` est appelé permettant ainsi de récupérer les pseudonymes contenus dans le message **J** et d'afficher (dans `ConnectedPopupPanel`) les personnes qui se connectent au fur et à mesure. La fonction `startGame` permet d'afficher les objets 2D à l'écran dont le joueur à besoin comme les noms des autres joueurs. `startRound`, qui se déclanche à la suite du message **S**, fait de même en affichant les boutons permettant de faire un pari ou désigner un menteur, respectivement `BetButton` et `DudoButton`. Ensuite, une manche peut commencer avec `startRound`, qui utilise `shake` et `showDices` pour faire le tirage aléatoire des dés et afficher les *sprites* des face adéquates. Les fonctions `turn` et `yourTurn` permettent de gérer l'affichage lorsque le serveur émet des messages **T**. Par ailleurs, lorsque son tour de jouer vient, le joueur peut utiliser `BetButton` et `DudoButton`. En appuyant sur `BetButton`, la fonction`makeGuess` est appelée avec la valeur souhaitée et le nombre de dés que le joueur veut parier. Si le joueur essaie d'envoyer un pari qui n'est pas en accord avec les règles du jeu, un message d'erreur s'affiche via la fonction `showWarning`. Sinon, `sendMessageToServer` permet, à partir de la scène courante, d'envoyer le message **P** contenant les informations sur son pari au serveur. Quant à `DudoButton`, le message **M** est envoyé au serveur. La figure 2.4 montre l'interface lorsque c'est au joueur de parler (ici Najwa). La figure 2.5, montre l'interface lors de la résolution de la phase de *Dudo* lancé par Jérôme contre le pari de Najwa. Par la suite, le message **R** permet la révélation des dés et et les messages **L** et **W** entraîne la résolution entre le joueur courant et le pari du joueur précédent.

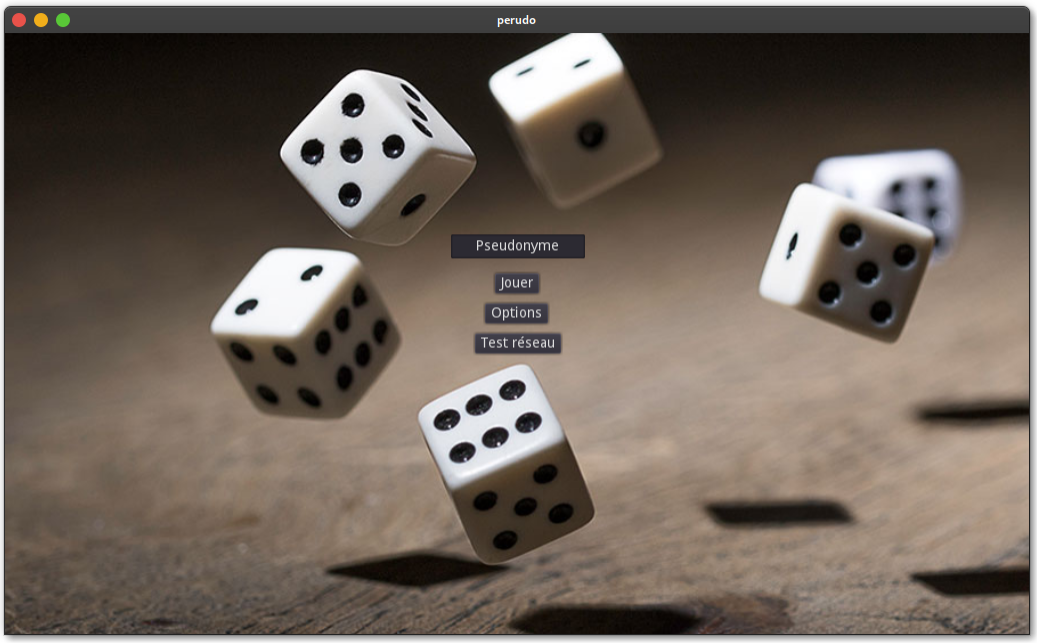

FIGURE 2.3 – Interface du menu principal

FIGURE 2.4 – Interface lorsque c'est à Najwa de parler

FIGURE 2.5 – Interface lors de la résolution de la phase de *Dudo* lancée par Jérôme contre Najwa

3
Conclusion

En ce qui concerne les tests, côté serveur, nous avons vérifié que le serveur réagissait bien face aux messages qu'il peut comprendre. C'est-à-dire que dans un premier temps nous avons commenté les *broadcasts* et remplacé ceux-ci par des *print* afin de rediriger le flux du serveur sur la console. De cette manière nous avons testé quasiment l'intégralité des cas de figure qui apparaissent au cours du jeu. Dans un deuxième temps nous avons décommenté les *broadcasts* et avons simulé quatre clients, dans le but de récupérer indépendamment leur flux pour tester les messages envoyés par le serveur, etc. Ainsi nous avons pu suivre certains cas de figure, et voir les affichages pour chaque client de façon à voir si nous donnions les bonnes sorties pour le client et vice versa.

Concernant les essais avec le client Godot, nous avons simulé l'envoie de messages du serveur central et avons observé comment le client réagissait aussi bien du point de vue de ses messages de réponses que de l'évolution de l'interface graphique au cours de la partie.

Nous avons trouvé que ce mini-projet était une application ludique du cours et une excellente introduction à la création d'un jeu en réseau. En effet, le fait d'avoir implémenté des fonctions nécessaires à la création du jeu (le protocole réseau, l'interface graphique, etc.) nous a permis d'acquérir une vision d'ensemble sur le paradigme serveur/clients, ainsi que sur certains aspects plus théoriques qui les composent, comme l'utilisation des *threads* ou encore de *UDP* par exemple. Nous aurions aimé rendre un jeu à la hauteur de nos envies (certaines sont citées dans la section précédente).

3.1 Perspectives d'évolution

Le jeu fourni n'est qu'un prototype, d'où de nombreuses améliorations sont possibles. Voici ci-dessous une liste non-exhaustive de ces améliorations.

- Avoir la possibilité de recommencer une partie.
- Créer dynamiquement des nœuds.
- Créer une plus belle interface pour les menus.
- Choisir des textures plus jolies.
- Transformer le jeu en 3D avec animation du lancé de dés (potentiellement contrôlé avec les *joysticks*) et prise en charge des collisions et de la gravité.
- Gérer les caméra et les lumières.

— Pouvoir se déplacer sur le plateau de jeu (déplacement de la caméra avec les *joysticks*).
— Prendre en charge des adversaires numériques.
— Ajouter la possibilité de jouer avec les règles optionnelles telles que Calza et Salsa.