Projet d'étude – Carcassonne
Rapport Final

Sébastien AGLAE – Lucas BLANC – Mike CHIAPPE –
Loïc Le Contel – Nathan Rihet

Table des matières

1. Point de vue général du projet

Inspiré de la forteresse médiévale du sud de la France du même nom, Carcassonne est un jeu de pose de tuiles dans lequel les joueurs remplissent la campagne autour de la ville fortifiée.

Ce projet consiste à reproduire le jeu de plateau « Carcassonne » au format numérique pour un maximum de cinq joueurs en même temps.

L'objectif de ce projet est de construire un système complet qui contient tous les concepts du jeu et qui est capable d'effectuer toutes les opérations du jeu en autonomie. Il doit également enregistrer correctement l'état du jeu et renvoyer les résultats lorsque le jeu est terminé. De plus, le client a demandé que le jeu puisse lancer 500 parties simultanément et nous retourner des statistiques précises sur chacune des parties. Depuis le rendu intermédiaire, nous avons rajouté diverses fonctionnalités. Nous noterons en gras les nouvelles modifications.

a. Glossaire

Glossaire des termes techniques du jeu :

Terme	Définition
Plateau de jeu	Représente la surface sur laquelle seront placés les éléments du jeu au cours de la partie.
Plateau de score	Représente la surface sur laquelle les joueurs comptent leur score de partie.
Tuile	Éléments du terrain qui seront utilisés à chaque tour de jeu. Elles permettent de construire le paysage du jeu. Il existe différents types de tuiles • **Route** : Permet de continuer une route. • **Ville** : Permet de continuer ou fermé une ville • **Abbaye** : Permet de créer une abbaye • **Départ** : Unique, elle est composée d'une partie de ville et de route • **Tunnel** : Segment de route qui traversent le tunnel sont connectés et forment une seule et même route • **Portail magique :** Après avoir placé une tuile Portail magique, le joueur peut placer un meeple sur cette tuile ou sur toute autre tuile placée précédemment. • **Volcan :** Tuile qui peut être placée où il veut par le joueur, aucun Meeple ne peut être placé sur cette dernière.

	• **Dragon :** Lorsqu'un joueur pioche une tuile présentant le dragon, il la place comme à l'habitude et peut y placer un partisan. Ensuite, avant l'évaluation, la partie est interrompue brièvement pour que le dragon se déplace. • **Princesse :** Si un joueur pioche une tuile présentant la princesse, il la place. S'il place la tuile dans une ville où on se trouve d'autres meeple. Il remet un de ces meeple à son propriétaire. Dans ce cas, il ne peut placer de meeple sur la tuile avec la princesse qu'il vient de placer. Si le joueur place la tuile Princesse de façon à continuer une ville vide ou si la tuile début une nouvelle ville, peut à ce moment y placer un partisan en suivant les règles de pose normales Leur nombre varie en fonction de l'édition et des extensions présentes
Fée	Permet de protéger un meeple du dragon.
Meeple	Représente le joueur dans le jeu. Au début de la partie, le joueur en possède 8 : 1 est placé sur le plateau de jeu et 7 peuvent être placés sur une route ou ville.
Abbé	Unique à chaque joueur, il peut être placé sur une abbaye lorsqu'elle est terminée.

b. Représentation générale

Lors de ce projet nous avons pu réaliser le jeu Carcassonne complet avec une grande partie de l'extension Dragon et Fée. Toutes les fonctionnalités du jeu ont été implémentés, on retrouve aussi une connexion client/serveur permettant d'avoir un dialogue entre ces deux derniers. Lors de ce rapport nous vous présenterons les détails des fonctionnalités et de la connexion.

c. Fonctionnalités traitées

Durant le développement de ce projet, nous avons implémenté les fonctionnalités de base du jeu ainsi que les fonctionnalités liées à l'extension Princesse et Dragon.

Dans l'état actuel, le joueur peut :

- Lancer une partie
- Obtenir une pile de cartes comme pioche
- Mélanger la pile de cartes
- Piocher une tuile depuis la pioche
- Tourner la tuile piochée
- Placer la tuile piochée
- Placer un meeple sur une tuile
- Placer une fée
- Bouger le dragon apparu / existant
- Gagner des points en fonction des zones fermées lors de l'évaluation du tour ou des zones ouvertes à la fin de l'évaluation du tour

2. Développement du moteur de jeu

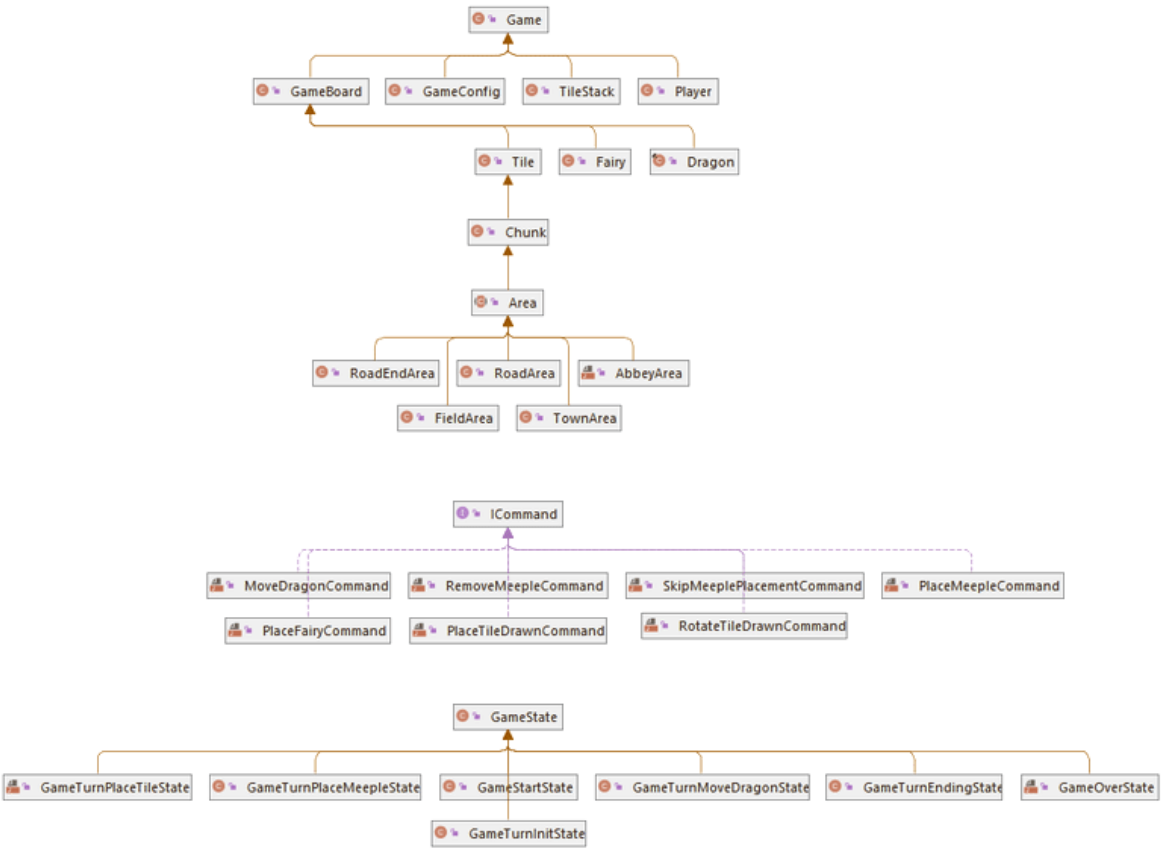
a. Analyse des besoins

Le composant « Moteur du jeu » contient toute la partie logique de Carcassonne. Elle est commune aux deux modules, « Client » et « Serveur ».

Le composant moteur doit pouvoir fournir aux différents modules les fonctionnalités suivantes :

- Créer une partie depuis une configuration de jeu
- Ajouter des joueurs à la partie
- Obtenir les joueurs dans la partie
- Démarrer la partie
- Exécuter des commandes de jeu
- S'attacher à des évènements de jeu

b. Diagramme de classes simplifié



Représentation d'un diagramme de classes simplifié du moteur de jeu. Nous expliquerons plus en détail les classes dans la prochaine partie. Le nombre de classes étant important, il est difficile de représenter le moteur sur un seul diagramme de classes.

« Game » est la classe générale du jeu. Elle représente une fois instanciée, une partie de jeu.

« Player » représente un joueur dans le jeu. Elle contient l'identifiant du joueur ainsi que son score et le nombre de meeples restants.

« GameBoard » est le plateau de jeu. Il contient et gère les tuiles sur le plateau, ainsi que les éléments « Dragon » (le dragon) et « Fairy » (la fée).

« Dragon » est le dragon présent sur le plateau. Il apparait lorsque le joueur tire une carte volcan et est bougé à chaque tour.

« Fairy » est la fée présente sur le plateau. Elle est placée par le joueur à côté de l'un de ses meeples et le protège contre le dragon.

« Tile » est une tuile de jeu. Cette tuile est composée de 13 « Chunk » (3 par côté + 1 au centre).

« Chunk » correspond aux différents éléments sur une tuile : Road (route), End Road (fin de route), Town (ville), Field (champ), Abbey (Abbaye).

« Area » correspond à la zone dans jeu. Il existe autant de type de zone que de type de « Chunk ».

« TileStack » est la pile de jeu. Elle est remplie et mélangée au début du jeu puis le joueur tire une tuile à chaque début de tour.

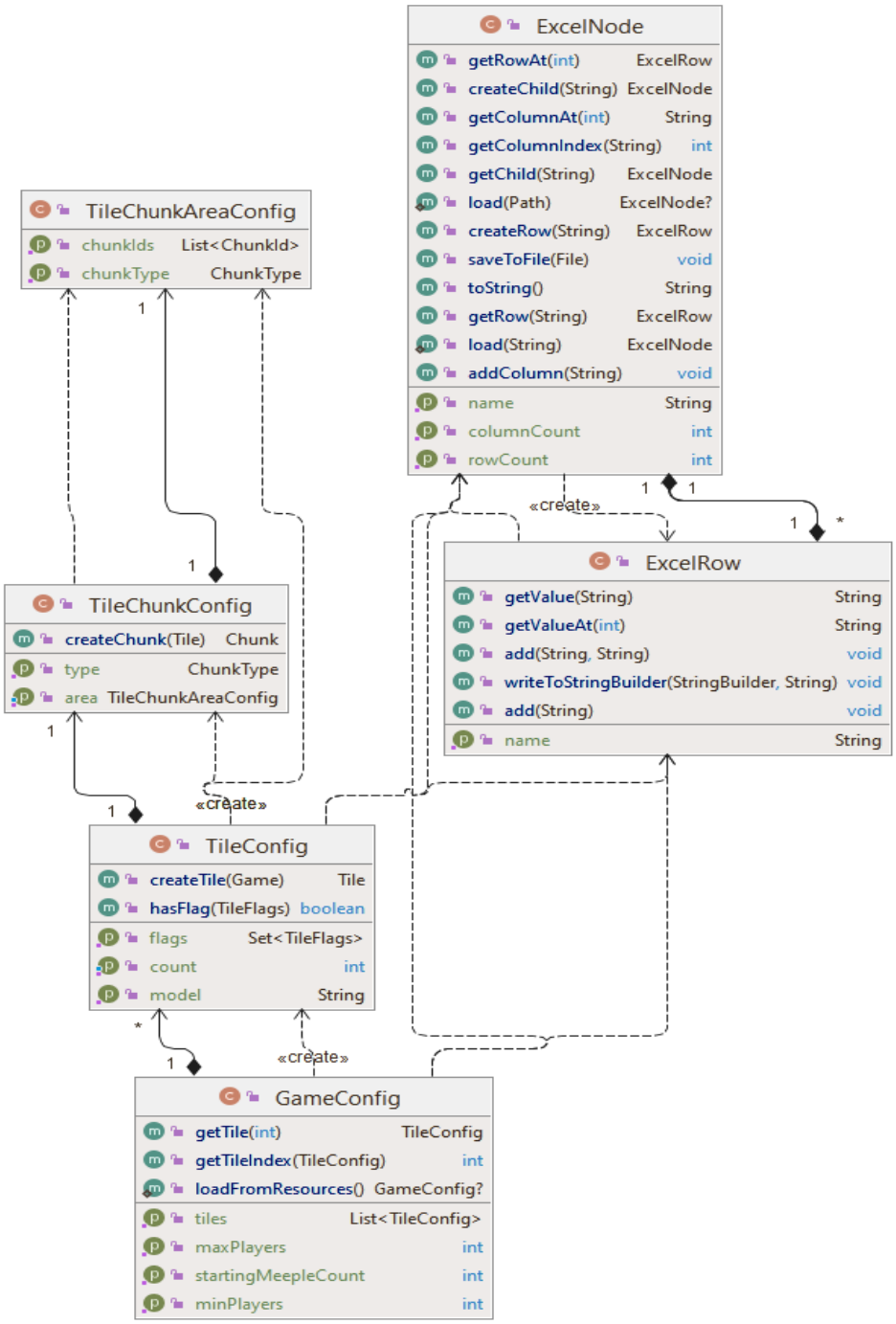
« GameConfig » est la configuration de jeu. Elle contient le nombre minimum et maximum possible dans le jeu ainsi que les tuiles jouables dans le jeu.

« ICommand » correspond aux différentes actions que le joueur peut faire dans le jeu.

« GameState » est l'état de jeu. Il évolue en fonction des actions des joueurs.

c. Diagramme de classes détaillé

i. Configuration



Voici le diagramme de classes contenant les différentes classes pour la configuration du jeu.

Notre configuration de jeu est sous forme de fichiers Excel exportés au format .txt. Chaque cellule est séparée par une tabulation (tsv). La classe « ExcelNode » se charge de charger et gérer les différents nœuds de chaque fichier (« load(Path) »). « ExcelRow » est une ligne du fichier Excel. Elle contient toutes les valeurs des cellules de la ligne.

Exemple de fichier de configuration (bleu les nœuds et orange les lignes) :

Chunks						
	Types					
		Name				
			FIELD	FIELD	FIELD	
		FIELD				FIELD
		FIELD		ABBEY		FIELD
		FIELD				FIELD
			FIELD	ROAD	FIELD	
	References					
		Name				
			A	A	A	
		A				A
		A		C		A
		A				A
			A	B	A	
Data						
	Name	Value				
	Model	A				
	Expansion					
	Flags					
	Count	2				

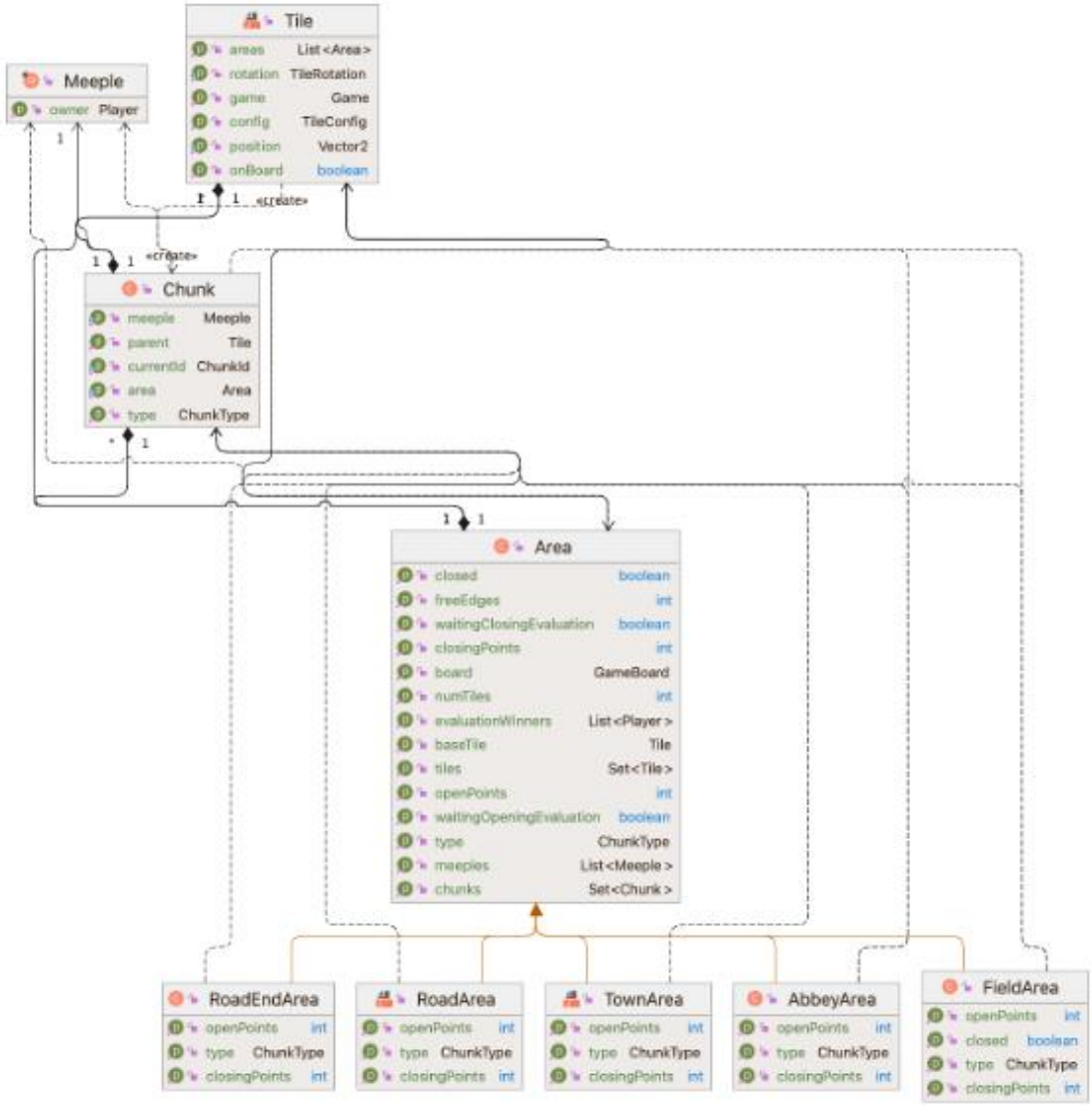
Ce fichier de configuration illustré ci-dessus correspond à la configuration d'une tuile de notre jeu. En effet, la tuile est composée de 13 chunks. Ce fichier de configuration correspond à cette tuile :

Comme on peut le voir on a principalement des chunks de « FIELD » qui correspondent aux champs. Au centre on y retrouve une « ABBEY » et en bas une « ROAD ». Ce qui est en bleu corresponds aux « nodes » et ce qui est en orange corresponds aux « rows ».

Il existe aussi une configuration pour « Game ». Elle permet de spécifier le nombre de joueurs minimum et maximum ainsi que les meeples de départ pour chaque joueur.

Name	Value
MinPlayers	2
MaxPlayers	5
StartingMeepleCount	7

ii. Tuiles et Zones



Le jeu est composé de plusieurs tuiles qui sont découpées en 13 chunks (3 par côté et 1 au centre). Chaque tuile est composée de plusieurs zones qui sont retrouvé grâce à la configuration de la tuile.

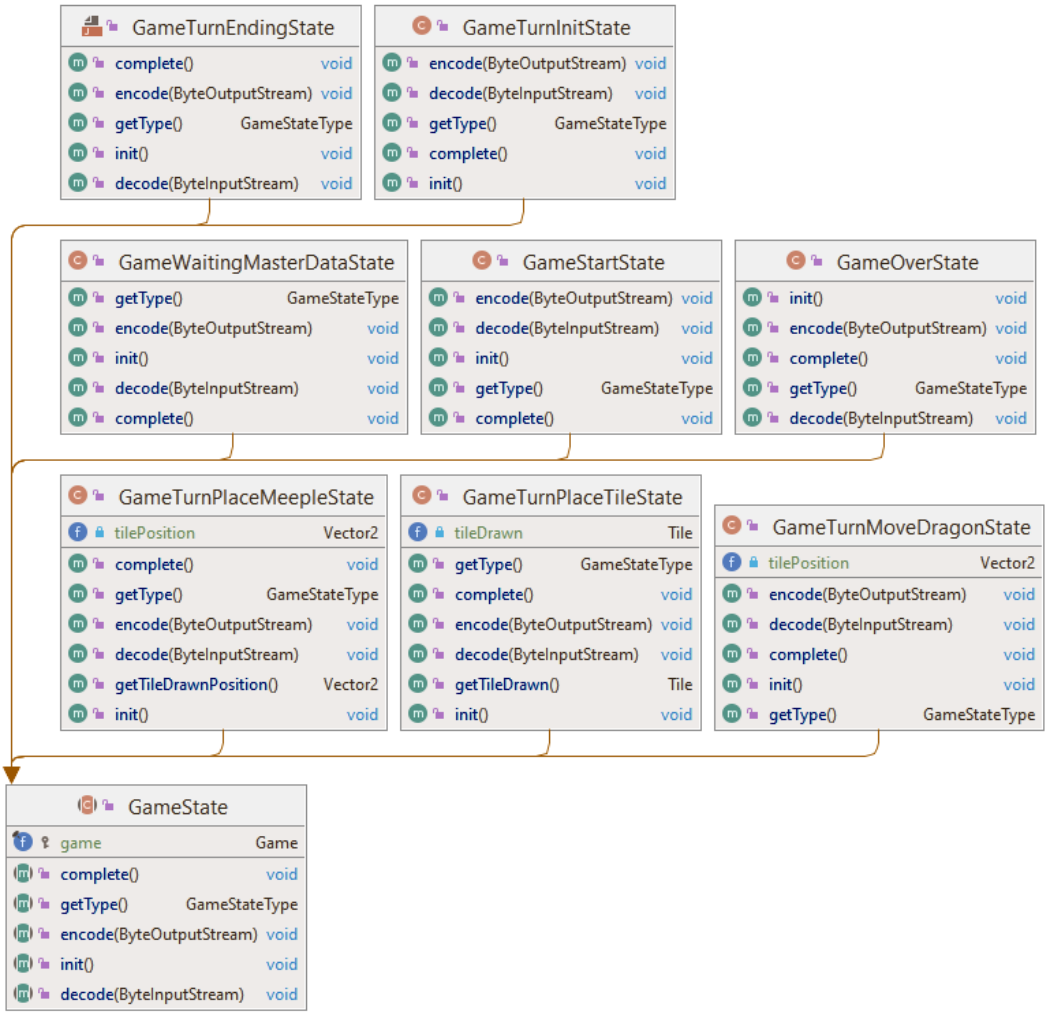
Chaque chunk est caractérisé par un type (ROAD, TOWN, ABBEY, FIELD). Ces types correspondent aux différents types de zones dans le jeu Carcassonne. Chaque chunk contient la zone à lequel il appartient ainsi que le meeple se trouvant dessus.

Les zones sont composées de plusieurs chunks. Il en existe plusieurs sortes. Chaque type de zone correspond à un type de chunk. Si un chunk est de type « ROAD », alors sa zone est dans une zone « ROAD ». Lorsque la tuile est placée, ses différentes zones sont fusionnées avec les zones aux alentours si elles sont connectées. Lors de cette fusion, le jeu va vérifier si la zone est dite fermée. Lorsqu'elle l'est, elle devra être évaluée à la fin du tour. Certaines zones comme « AbbeyArea » ne sont pas fermée de la même manière que les zones « conventionnelles ». Elles nécessitent d'être connectées à 8 tuiles, chacune autour de la tuile principale.

Les meeples sont des éléments placés par le joueur sur la tuile qui vient d'être placé (sauf exception du portail magique). Ils permettent de gagner des points lorsque la zone est fermée lors de l'évaluation

de tour ou lorsqu'elle est ouverte lors de l'évaluation finale. Le joueur ayant le plus de meeples dans la zone gagne les points de fermeture de zones et ces meeples se voient retourner aux joueurs. Dans le cas où il existe plusieurs joueurs gagnants, ils se verront tous recevoir ces points.

iii. États



Le jeu fonctionne grâce à différents états. Ces états vont évoluer en fonction des actions que le joueur exécutera durant la partie. L'état de départ est appelé « GameStartState ». Il est défini lorsque le jeu démarre.

Chaque état implémente les méthodes « getType() », « init() », « complete() ».

Explication de chaque état :

- GameStartState : Correspond à l'état du jeu lorsque le jeu démarre. Il initialise chaque composant du jeu et rempli la pioche. Lorsque c'est fait, le jeu passe à l'état GameTurnInitState si c'est un master (serveur) ou bien GameWaitingForMasterDataState si c'est un slave (client).
- GameTurnInitState : Tire une tuile de la pioche. Si aucune tuile ne peut être tiré, le jeu passe automatiquement à l'état GameOverState.
- GameWaitingMasterDataState : Correspond à l'état lorsque les données du master (serveur) sont en attente. Les données du master sont requises car nous avons besoin de connaitre la prochaine tuile dans la pioche pour démarrer le nouveau tour.

- GameOverState : Etat du jeu en fin de partie.
- GameTurnPlaceTileState : Correspond à l'état du jeu lorsque le joueur place une tuile.
- GameTurnPlaceMeepleState : Correspond à l'état du jeu lorsque le joueur doit placer un meeple. Le joueur peut compléter l'état en exécutant la commande PlaceMeepleCommand ou en passant le placement grâce à la commande SkipMeeplePlacementCommand. Il peut aussi exécuter la commande PlaceFairyCommand.
- GameTurnMoveDragonState : Correspond à l'état du jeu lorsque le joueur doit déplacer le dragon. L'état est complété s'il n'y a pas de dragon sur le plateau. Le joueur peut compléter l'état en exécutant la commande MoveDragonCommand.
- GameTurnEndingState : Correspond à l'état du jeu lorsque le tour se termine.

iv. Commandes

Le joueur peut effectuer des actions dans le jeu à l'aide de « commande ». Ces commandes peuvent être exécutées via la méthode « Game::executeCommand(Command) ».

« getType() » permet d'obtenir le type de commande.

« getRequiredState() » permet d'obtenir l'état de jeu dans lequel cette commande peut être exécuté.

Les méthodes « decode() » et « encode() » permettent de lire et d'écrire les attributs des commandes dans un flux d'octets. Cela permettra plus tard d'envoyer les commandes au serveur et vis-versa.

« canBeExecuted(Game) » permet de savoir si cette commande peut être exécuter sur l'instance de jeu spécifiée. L'erreur retournée correspond à ERROR_STATUS si la commande peut être exécutée, sinon elle correspond aux constantes « ERROR_* » déclarées dans la classe de la commande.

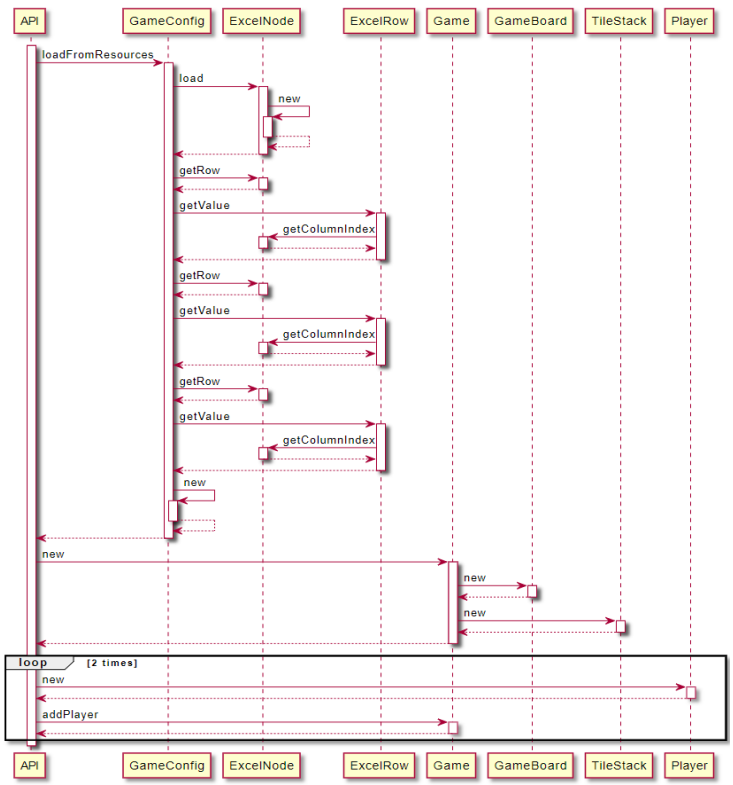
Enfin, « execute(Game) » exécute la commande sur la partie spécifiée.

Explications des différentes commandes :

- MoveDragonCommand : Permet de déplacer le dragon dans une direction spécifié
- RotateTileDrawnCommand : Permet de faire tourner une tuile dans une position spécifiée (exemple : faire tourner la tuile à droite -> TileRotation.RIGHT)
- PlaceTileDrawnCommand : Permet de placer la tuile tirée durant le début du tour à l'emplacement spécifié.
- SkipMeeplePlacementCommand : Permet de passer l'étape du placement de meeple
- PlaceMeepleCommand : Permet au joueur de placer un meeple sur une tuile grâce à la position de la tuile et à l'identifiant du chunk
- RemoveMeepleCommand : Permet au joueur de retirer un meeple placé grâce à la position de la tuile et à l'identifiant du chunk
- PlaceFairyCommand : Permet de placer une fée sur un chunk grâce à la position de la tuile et à l'identifiant du chunk

d. Diagramme de Séquence

i. Initialisation d'une partie



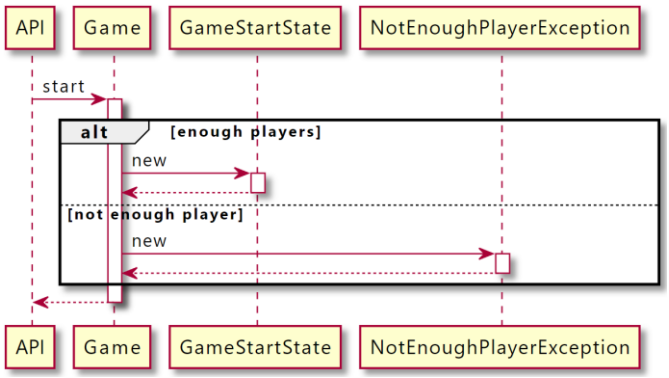
Séquence d'initialisation d'une partie avec deux joueurs.

Dans un premier temps, nous chargeons la configuration « GameConfig » en appelant la méthode statique « loadFromResources() ». Elle permet de charger automatiquement la configuration depuis les fichiers ressources.

Ensuite, nous créons une instance « Game » avec la configuration chargée.

Enfin, pour chaque joueur, nous créons une instance puis nous appelons « addPlayer(Player) » pour l'ajouter à la partie.

ii. Démarrage d'une partie

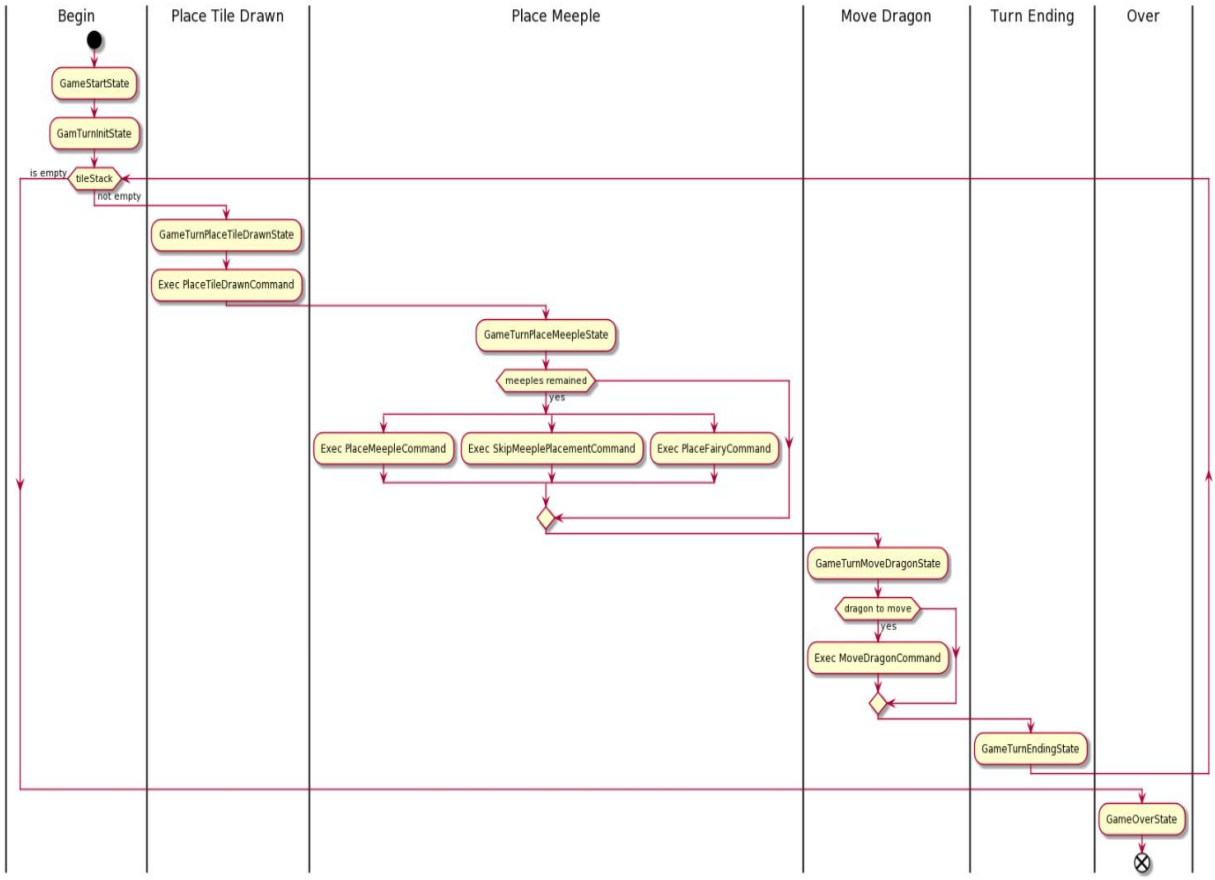


Afin de démarrer la partie, la méthode « start() » doit être appelée.

Dans le cas où le nombre de joueurs est suffisant, l'état de jeu va passer à « GameStartState ».

Sinon, le jeu lancera une exception « NotEnoughPlayerException » pour informer l'appelant que le nombre de joueurs n'est pas suffisant pour jouer.

e. Diagramme d'activité d'une partie



3. Développement du Client

a. Analyse des besoins

Le commanditaire souhaite une architecture client / serveur pour son fameux jeu Carcassonne. Dans cette architecture, le client joue le rôle de vue et d'interface entre le joueur (IA) et le serveur. Lorsque nous avons accepté de réaliser sa demande, nous avons réfléchi à deux designs différents.

	Moteur de jeu sur le serveur	Moteur de jeu sur le client et le serveur
Avantages	- Client qui ne sait que ce qu'il a à savoir - Client indépendant de la logique de jeu	- Client qui sait tout de l'état actuel du jeu - Client indépendant du serveur pour simuler une action de l'IA afin de prévoir les conséquences d'une action - Client indépendant du serveur pour jouer au jeu (mode offline)

		- Economie de bande passante (pas besoin de renvoyer un instantané du plateau de jeu pour chaque action du client)
Inconvénients	- Client qui n'a pas une vue complète sur l'état du jeu - Client fortement dépendant du serveur - Client qui a besoin du serveur pour simuler une action - Haute utilisation de la bande passante	- Client peut savoir des choses qui ne lui sont pas / peu utiles
Choix		X

La seconde solution a été choisi pour ces nombreux avantages comparés à l'autre solution. Elle nous semblait la plus adapter car elle permet un énorme gain de bande passante et de latence.

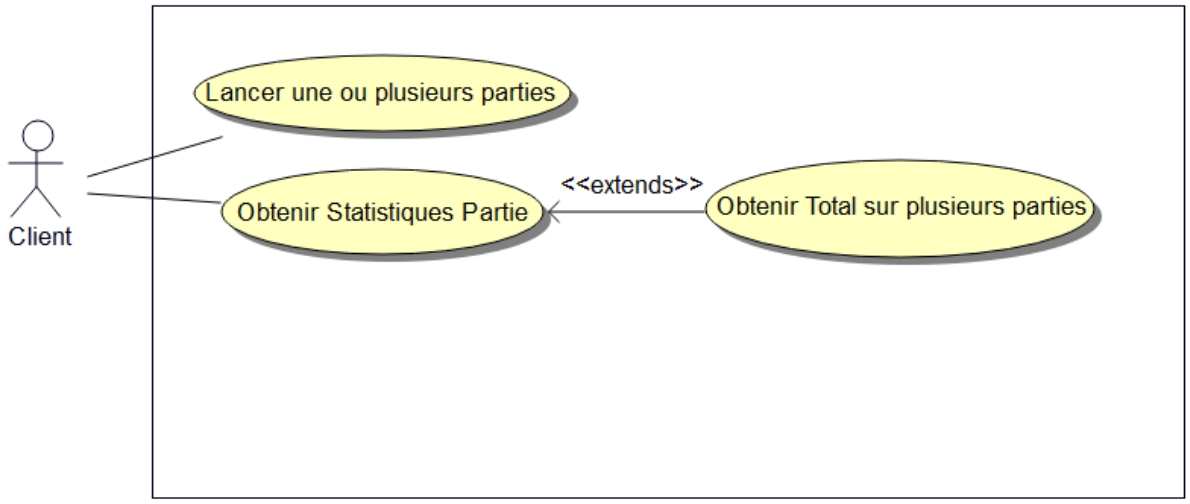
Sur 500 parties et avec 5 joueurs nous avons enregistrés un total de 10MO de données envoyées par le serveur. Avec la seconde architecture, nous avons estimés que ce chiffre peut grimper jusqu'à 5GO en raison des nombreux envoies de données à chaque tour (tout le plateau doit être envoyé aux clients). Cette économie de bande passante se retrouve aussi au niveau de l'utilisation du processeur qui se retrouve beaucoup plus faible étant donné qu'il n'a pas à traiter de grandes quantités de données.

De plus des économies de bande passante, grâce à l'utilisation des évènements pour déclencher l'IA et le système de commande avant la livraison 4, cette solution était beaucoup plus rapidement implémentable sur notre moteur.

Dans cette solution, le client devra pouvoir :

- Se connecter au serveur
- Rejoindre automatiquement le matchmaking pour rejoindre une partie multijoueur
- Démarrer une partie depuis les données de partie du serveur
- Simuler le comportement d'un joueur via une intelligence artificielle
- Transférer les commandes exécutées au « maitre de jeu » (serveur) pour les exécuter réellement sur la partie
- Recevoir et exécuter les commandes validées et exécutées par le serveur
- Enregistrer la notification de fin de partie du serveur pour générer les statistiques

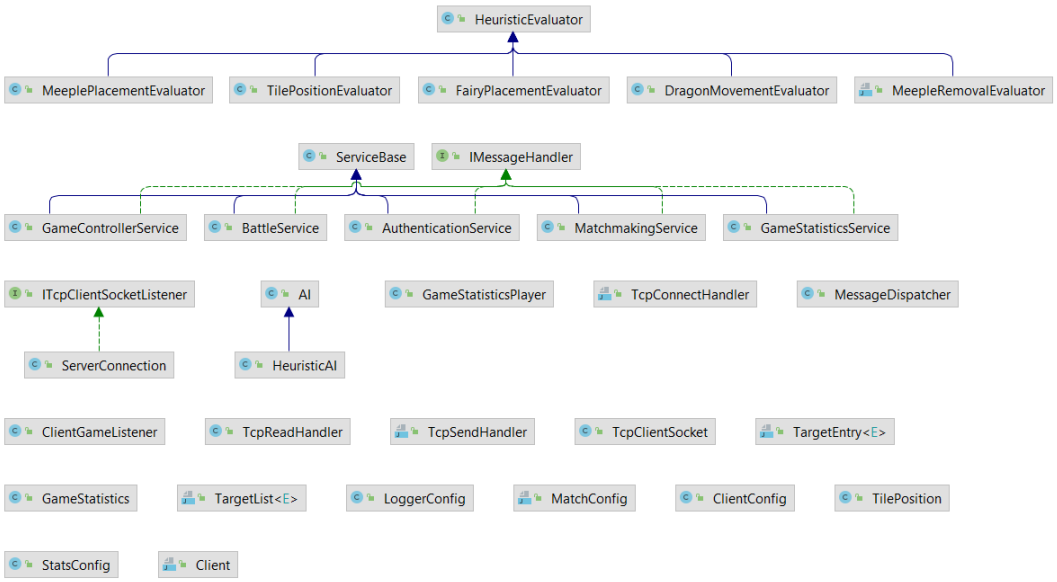
En plus des obligations pour l'architecture client / serveur, le client doit respecter les cas d'utilisations fournis par le client.



Le client doit pouvoir :

- Lancer une ou plusieurs parties
- Obtenir les statistiques : il pourra s'il le souhaite obtenir un total des statistiques sur plusieurs parties

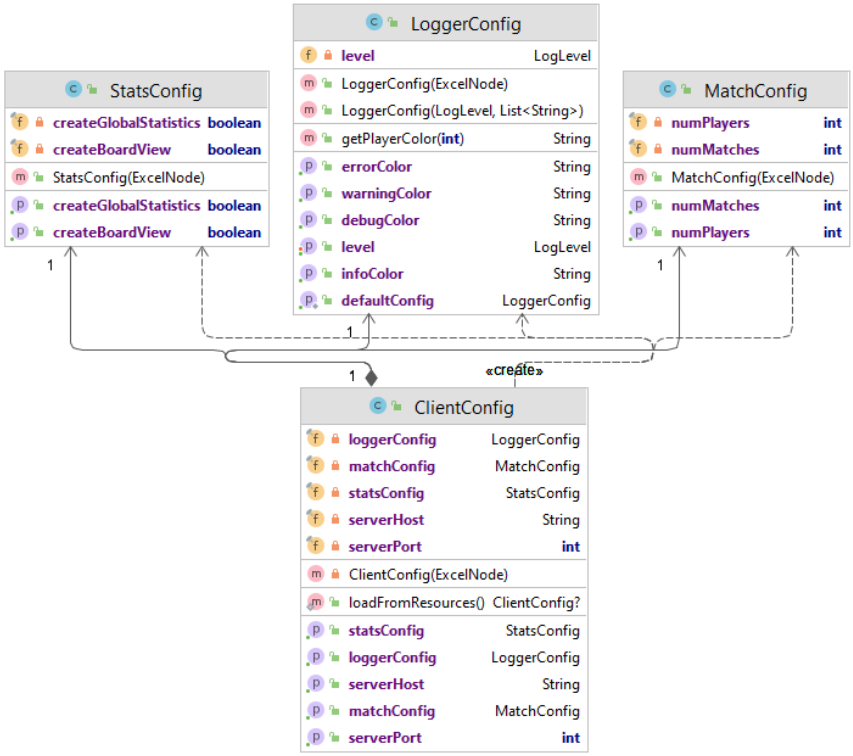
b. Diagramme de classes simplifié



Voici un diagramme simplifié du client. Nous expliquerons plus précisément chaque classe dans la prochaine partie.

c. Diagramme de classes détaillé

i. Configuration



Le client peut changer la configuration des statistiques, du logger, des parties à lancer et du réseau.

Les 4 types de configuration sont stockés dans un fichier texte. Ce fichier texte contient 4 nœuds pour chaque élément à configurer.

Server		
	Name	Value
	Host	127.0.0.1
	Port	8080
Logger		
	Name	Value
	Level	INFO
	Player_1	35m
	Player_2	34m
	Player_3	93m
	Player_4	94m
	Player_5	95m
Match		
	Name	Value
	NumMatches	10
	NumPlayers	2
Stats		
	Name	Value
	CreateBoardView	True
	CreateGlobalStatistics	True

Dans le « StatsConfig », on y trouve :

- « createGlobalStatistcs », il permet de dire si l'on veut générer des statistiques globales.
- « createBoardView », il permet de dire si l'on veut générer une image du jeu.

Dans le « LoggerConfig », on y trouve :

- « errorColor », « warningColor », « debugColor » et « infoColor » qui sont les différentes couleurs pour les logs du jeu.
- « playerColors » permet de spécifier les couleurs de logs pour chaque joueur.
- « level » indique le niveau de logging qu'on veut afficher sur la console.

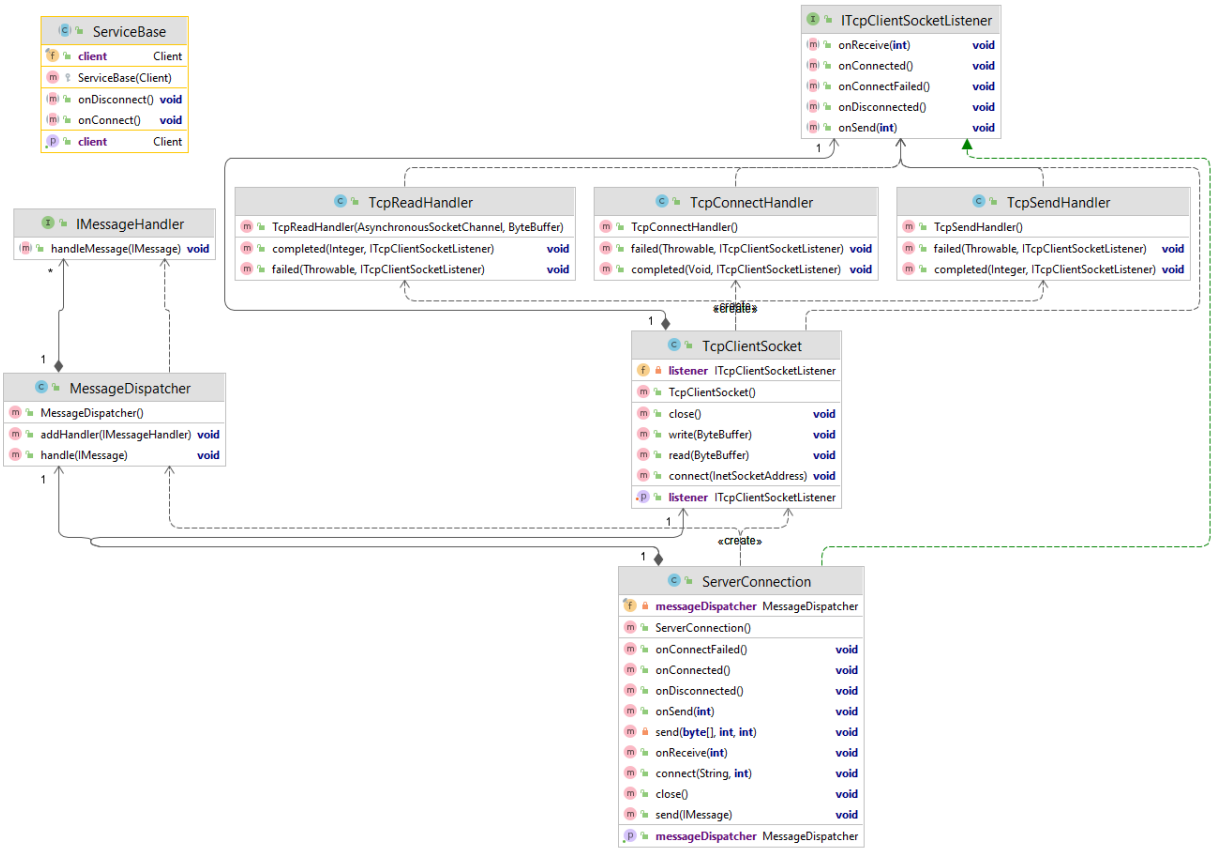
Dans « MatchConfig », on y trouve :

- « numPlayers », il permet de spécifier le nombre de joueur dans la partie.
- « numMatches », il permet de spécifier le nombre de partie à lancer.

Dans « ClientConfig », on retrouve toutes nos configurations précédentes avec en plus :

- « serverHost », il permet de spécifier l'IP du serveur Carcassonne.
- « serverPort », il permet de spécifier le port du serveur Carcassonne.
- « loadFromResources() », il permet de charger la configuration depuis des fichiers textes.
-

ii. Réseaux



La classe « TcpClientSocket » va établir la connexion avec le serveur :

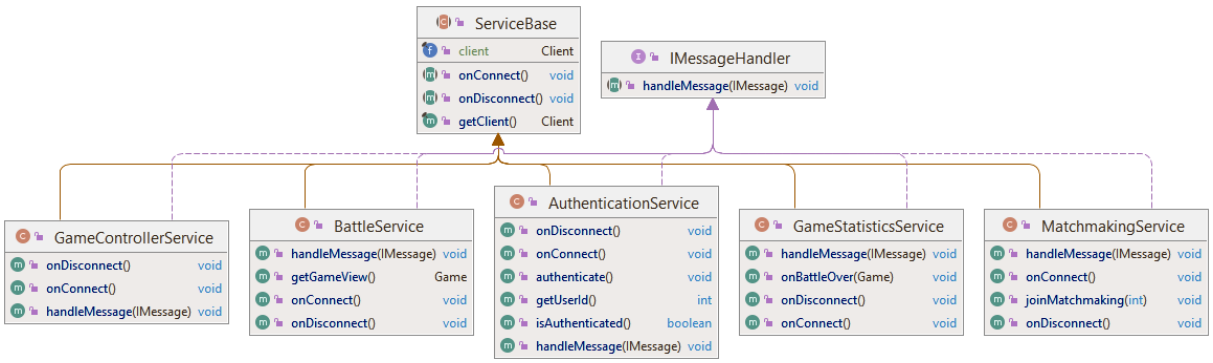
- « connect(INetSocketAddress) » permet de se connecter au serveur à l'adresse spécifiée.
- « write(ByteBuffer) » permet de démarrer l'envoi de données de manière asynchrone. Le « socket » notifiera la fin de l'envoi au listener attaché.
- « read(ByteBuffer) » permet de démarrer la réception des données. Le « socket » notifiera la fin de la reception au listener attaché.
- « close() » permet de fermer la connexion avec le serveur.

La classe « ServerConnection » gère le socket et l'écoute en implémentant l'interface « ITcpClientSocketListener ». Il est notifié par une réception via la méthode « onReceive(int) » et par une fin d'envoi par « onSend(int) ».

Les classes « TcpReadHandler », « TcpConnectHandler » et « TcpSendHandler » sont des classes qui permettent de gérer la lecture, la connexion et l'envoie des données de manière asynchrone.

Le « MessageDispatcher » permet de dispatcher les messages reçus vers les services qui écoutent la réception des messages (voir Services).

iii. Services



Le client possède plusieurs services et effectue différentes tâches.

Les services héritent tous de « ServiceBase » et peuvent implémenter l'interface « IMessageHandler » pour qu'il soit ajouté au « MessageDispatcher » et ainsi être notifié des messages envoyés par serveur.

Détails de « ServiceBase » :

« onConnect() » est appelé lorsque le client est connecté au serveur.
« onDisconnect() » est appelé lorsque le client est déconnecté du serveur (peut arriver lorsque Client::close est appelé ou si la connexion est perdu).

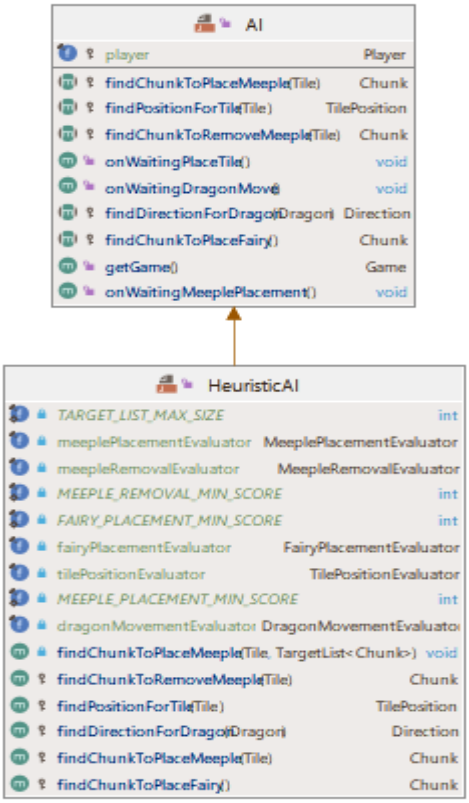
Détails de l'interface « IMessageHandler » :
« handleMessage(Message) » : Appelé lorsque le client reçoit un message.

Liste des services :

« GameControllerService » contrôle le jeu. C'est lui qui démarre automatiquement le matchmaking et arrête le client lorsque toutes les parties sont jouées.

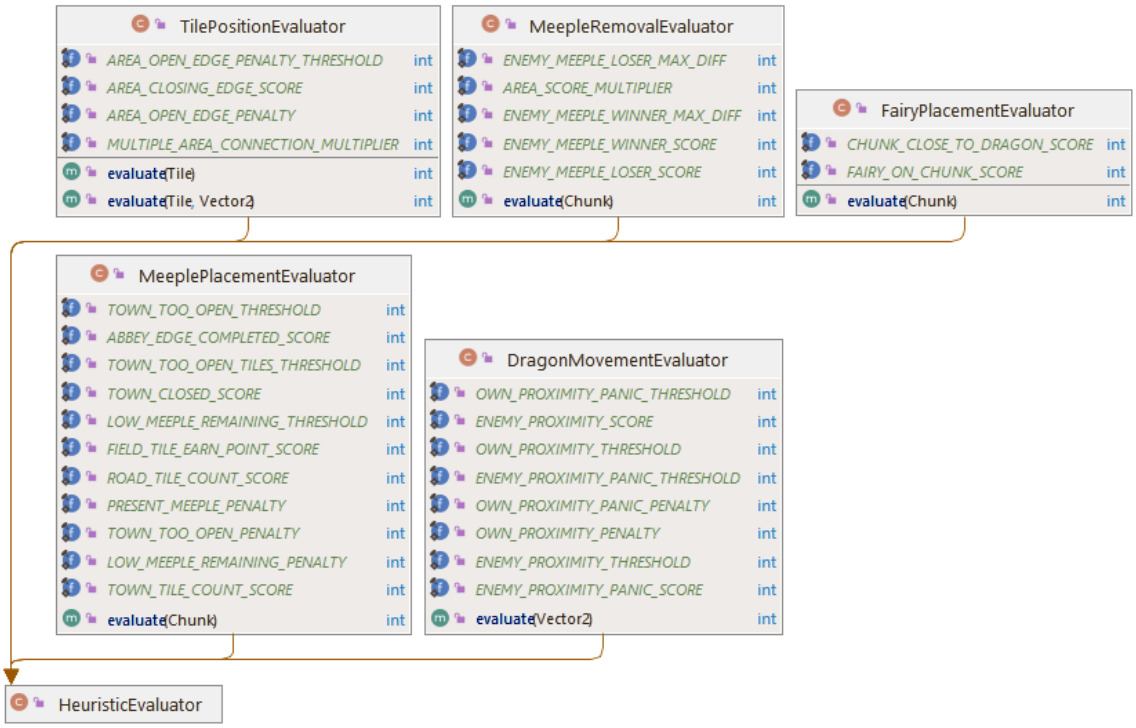
« GameStatisticsService » gère les statistiques des parties. Il enregistre chaque statistique de partie et génère les statistiques globales lorsque toutes les parties sont jouées.
« MatchmakingService » gère le matchmaking. Il demande de rejoindre un matchmaking via la méthode « joinMatchmaking() ».
« AuthenticationService » s'occupe de l'authentification du client. Il démarre automatiquement le client lorsqu'il est connecté au serveur.
« BattleService » est responsable des parties en cours. Il initialise la partie en cours lorsqu'il reçoit « GameDataMessage » et la détruit lorsqu'il reçoit « GameResultMessage ». Lorsqu'une partie est terminée, il notifie le service « GameStatisticsService » afin qu'il génère les statistiques de la partie.

iv. Intelligence Artificielle



L'intelligence artificielle « HeuristicAI » est une intelligence artificielle utilisant un algorithme heuristique. Elle utilise différents évaluateurs pour chaque cible à déterminer. Ces évaluateurs vont attribuer un score à chaque cible et l'IA prendra le score le plus haut comme cible à choisir.

Diagramme de classes des évaluateurs :

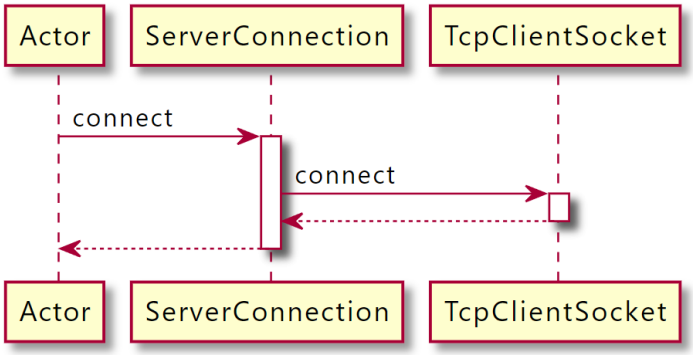


Tous les evaluators ont tous une méthode « evaluate() » qui permet d'évaluer une action.

« HeuristicEvaluator » permet de gérer le score pour chaque situation :
« resetMultiplier() » Remet à 1 le multiplicateur.
« addPenality() » ajoute une pénalité dans le score.
« finalizeScore() » permet de récupérer le score et le remet ensuite à zéro.
« multiplyScore() » multiplie le score actuel.
« addScore() » ajoute un nombre au score.
« HeuristicAI » permet d'obtenir le mouvement à faire.
« FairyPlacementEvaluator » permet d'évaluer le placement de la fée.
« TilePositionEvaluator » permet d'évaluer la position de la tuile.
« MeeplePlacementEvaluator » permet d'évaluer le placement d'une meeple.
« DragonMovementEvaluator » permet d'évaluer le déplacement du dragon.
« MeepleRemovalEvaluator » permet d'évaluer la suppression d'un meeple.

d. Diagramme de séquence

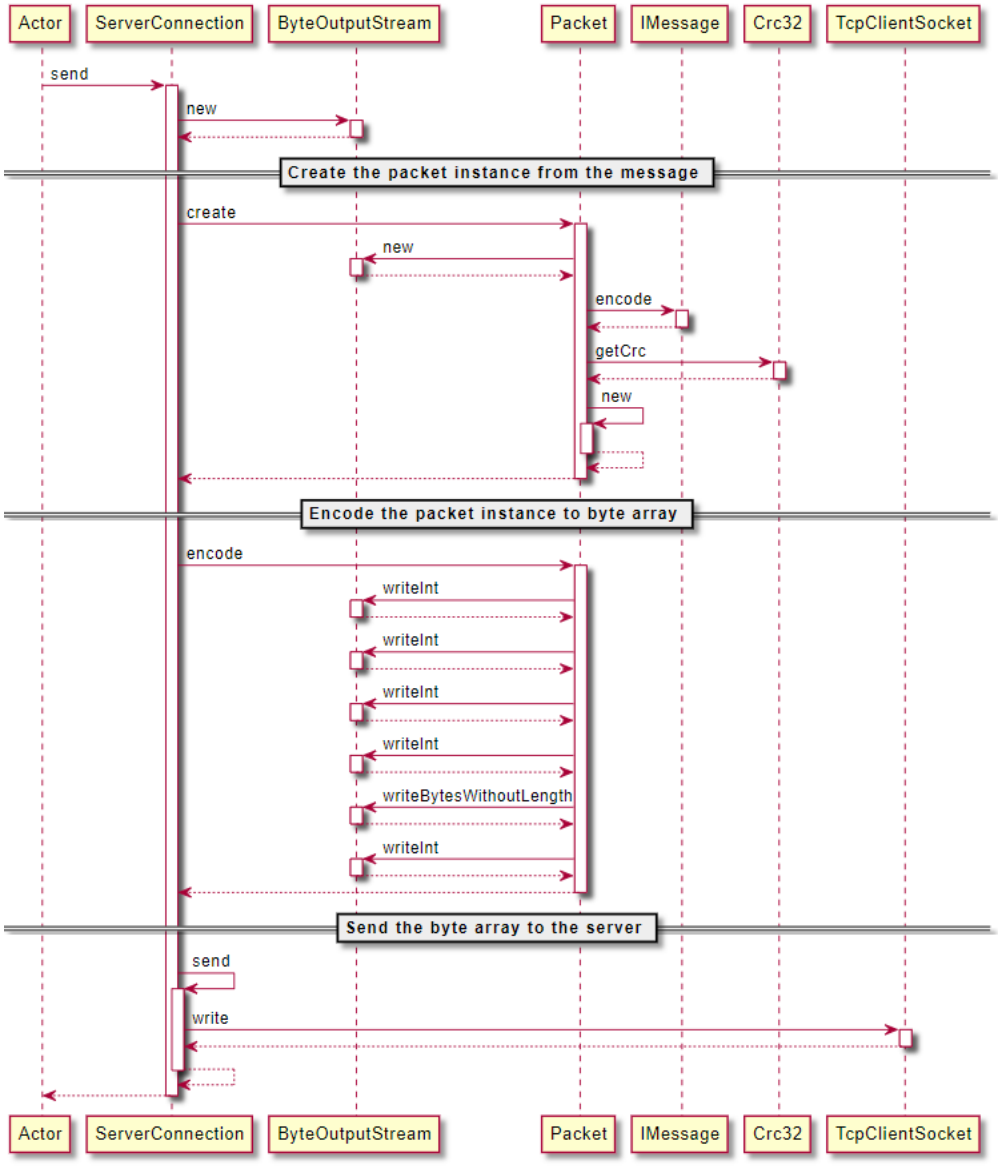
i. Connexion au serveur



Afin d'établir une connexion, la méthode « connect() » doit être appelée.

« ServerConnection::connect(String, int) » va créer un INetSocketAddress depuis l'adresse et le port du serveur. Ensuite il va appeler « TcpClientSocket::connect(INetSocketAddress) ». Le reste de la connexion se fait de manière asynchrone. Le client sera notifié du résultat via le listener.

ii. Envoyer un message au serveur



L’envoi d’un message au serveur se fait à l’aide de « ServerConnection::send(Message) ».

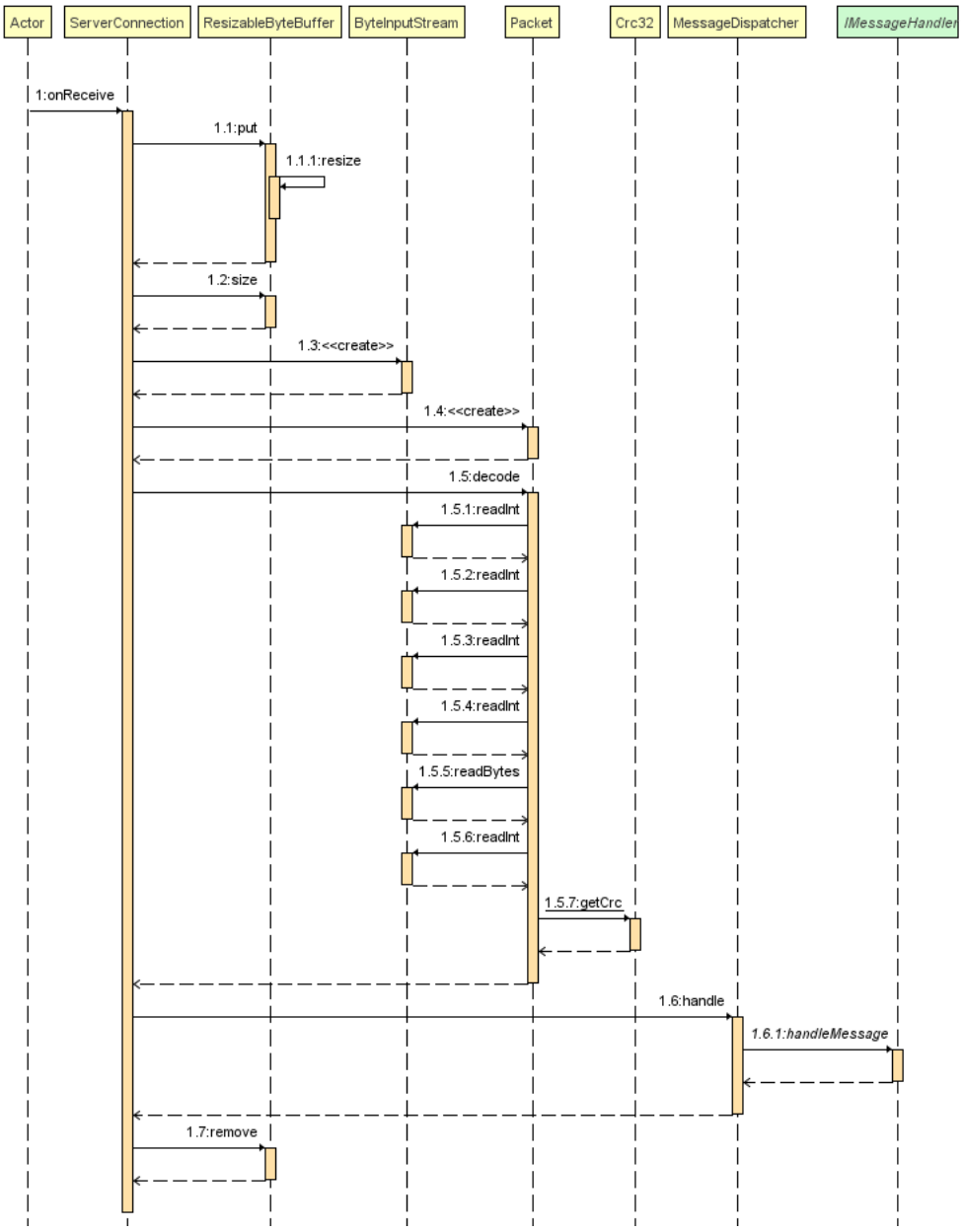
Cette méthode va créer un flux de sortie qui va servir à encoder le paquet en liste d’octets. Ce flux est un objet « ByteOutputStream » qui permet d'écrire des éléments primitifs (byte, int, String) et d'obtenir à la fin une liste d'octets.

Ensuite, nous allons créer un paquet depuis le message qu’on veut envoyer. Cette méthode va encoder le message et ensuite initialiser les attributs du paquet (type de message, longueur, somme de contrôle et contenu).

La somme de contrôle est calculée à l’aide de l’algorithme CRC32. Elle permet de vérifier que le contenu du paquet à bien été lu.

Enfin, le paquet est envoyé au serveur en appelant « TcpClientSocket::write(ByteBuffer) ».

iii. Recevoir un message du serveur



La réception d'un message démarre lorsque que « ServerConnection::onReceive(int) » est appelé par le socket. Les octets reçus vont être ajoutés à la liste d'octets à traiter.

Ensuite, nous allons décoder les paquets depuis la liste d'octets à traiter.

Par la suite, nous appelons « MessageDispatcher::handle(IMessage) » qui va se charger de dispatcher les messages aux services.

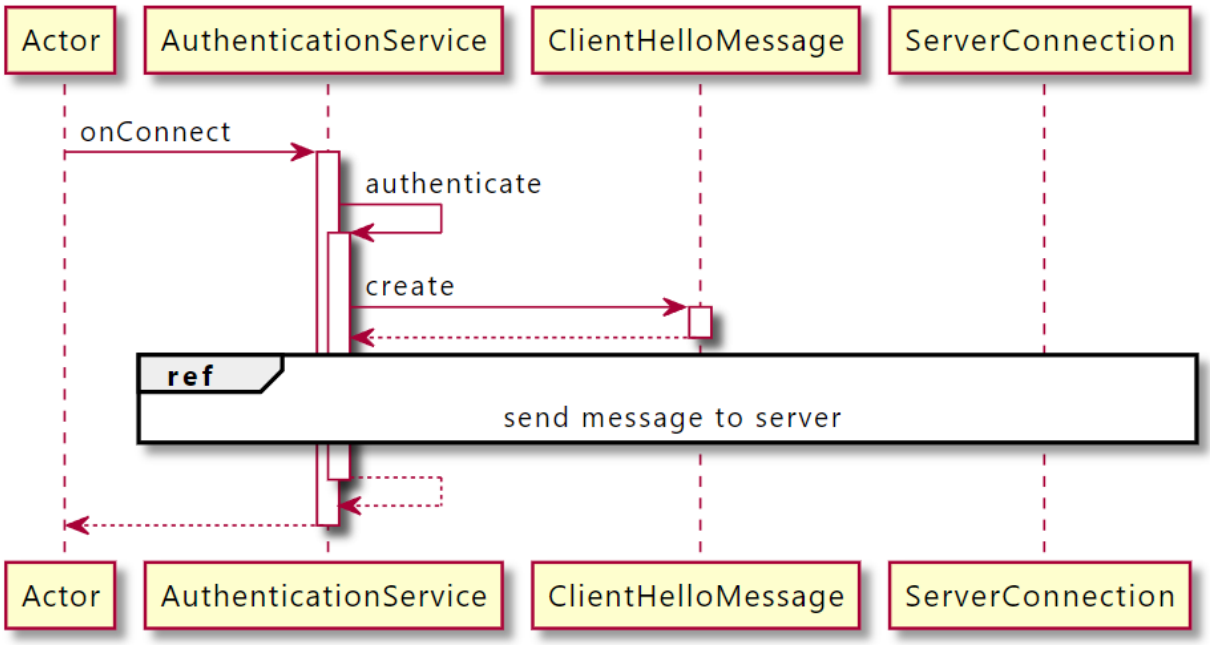
Enfin, nous supprimons les octets lus correctement de la liste des octets à traiter et nous redémarrons le processus de réception asynchrone.

iv. Authentification avec le serveur

Afin d'être authentifiée avec le serveur, le client doit envoyer au serveur le message « ClientHelloMessage ».

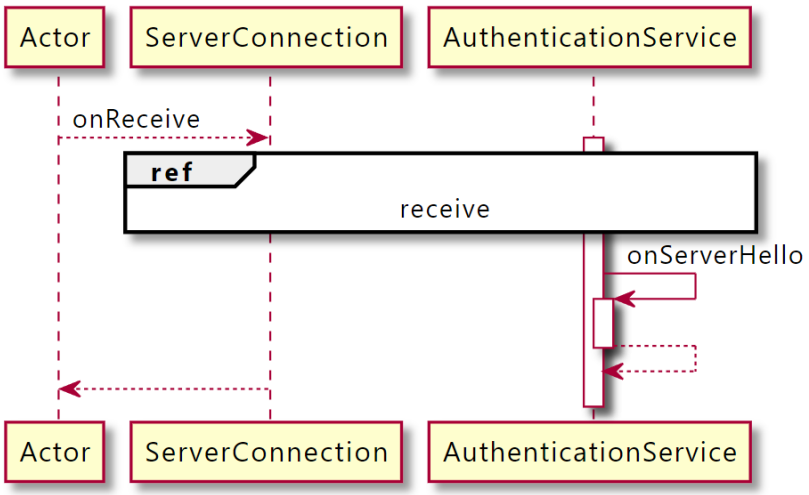
Lorsque le serveur recevra ce message, il répondra via le message « ServerHelloMessage ».

Diagramme de séquence pour l'envoi du message :



Lorsque le client est authentifié, il appelle la méthode « onConnect() » des services. Cette méthode va envoyer le message « ClientHelloMessage » au serveur.

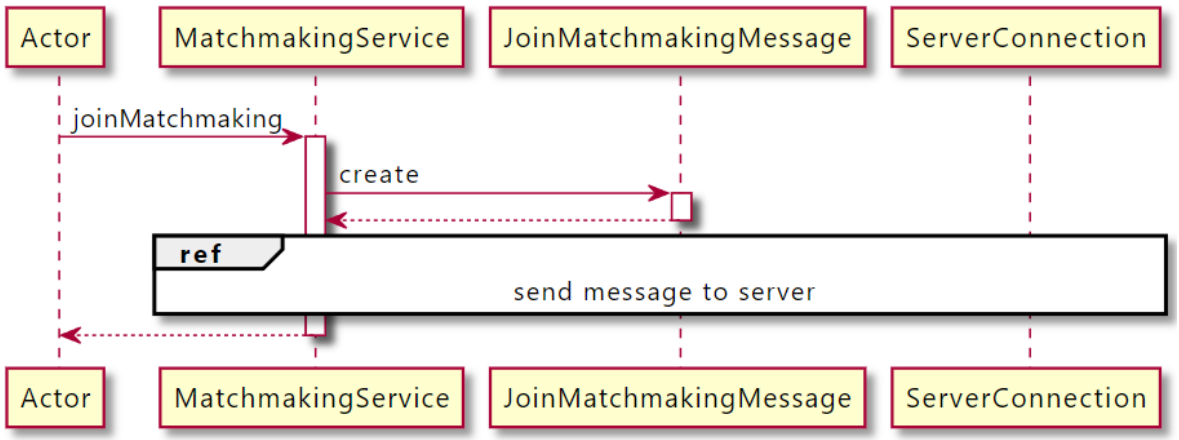
Diagramme de séquence pour la réception du message :



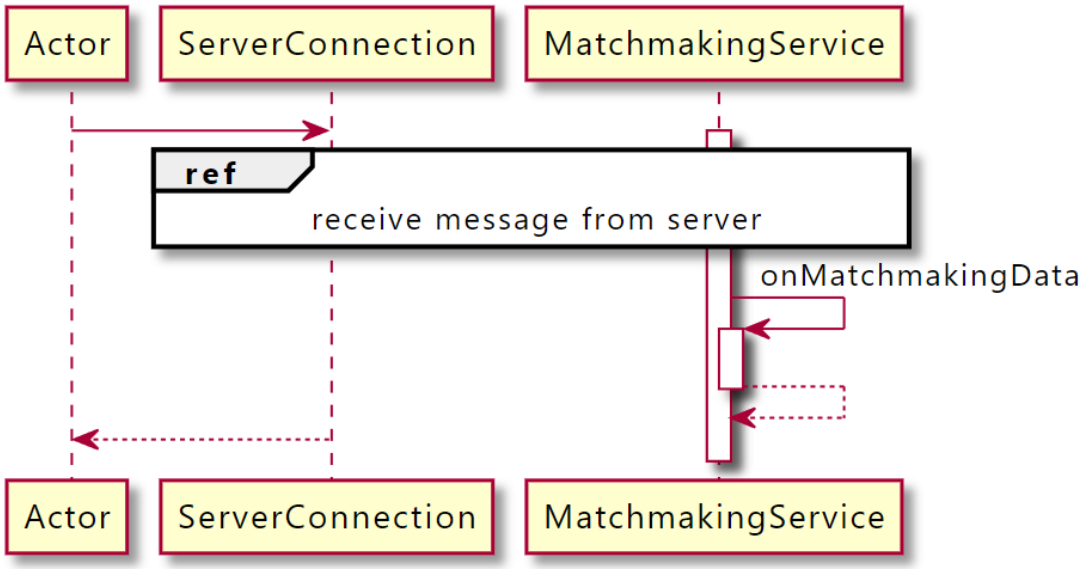
Le client reçoit la réponse « ServerHelloMessage » (cf Recevoir un message du serveur).

AuthenticationService est notifié de la réception du message par le dispatcher. Alors, il définit son userId par l'userId contenu dans le message.

v. Rejoindre le matchmaking



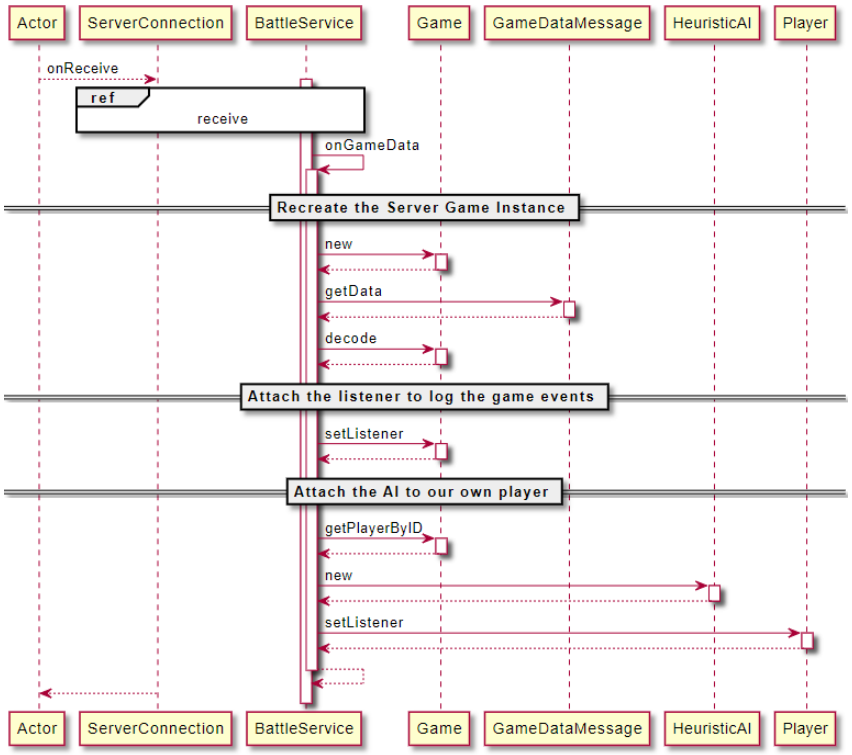
Lorsque la méthode « joinMatchmaking » est appelé, le service va envoyer une demande au serveur pour rejoindre le matchmaking. Cette demande est faite par l'intermédiaire du message « JoinMatchmakingMessage ».



Le client est notifié de la réussite pour rejoindre le matchmaking lorsqu'il reçoit le message « MatchmakingDataMessage » de la part du serveur.

vi. Démarrage de la partie

Le client sait que la partie commence lorsqu'il reçoit « GameDataMessage » de la part du serveur. Ce message contient toutes les données nécessaires pour reconstruire la partie.



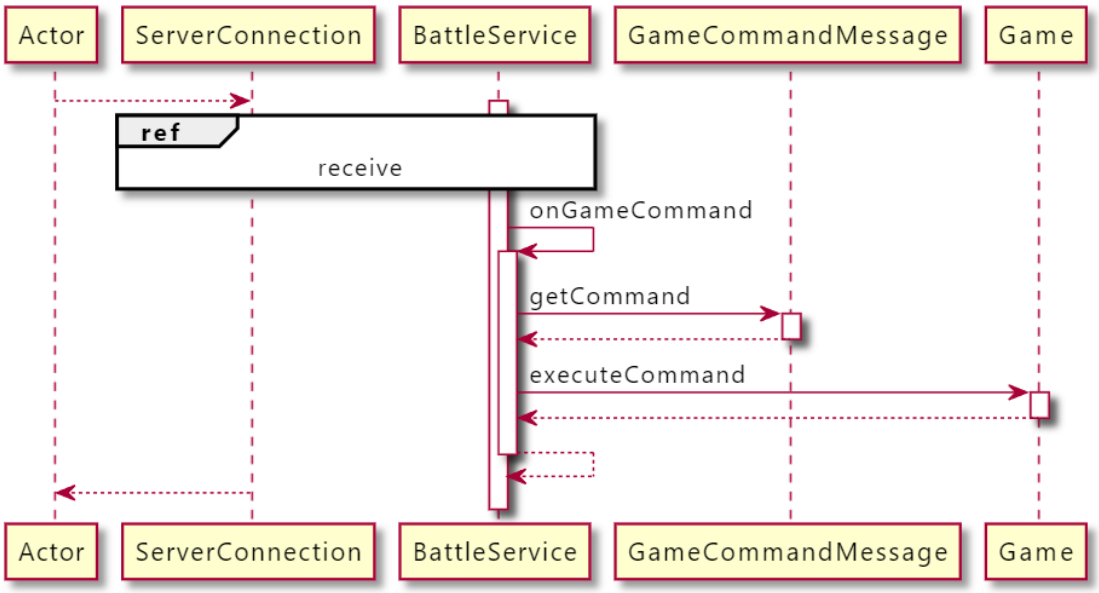
Lorsque le client reçoit le message « GameDataMessage » le client crée une instance du jeu. On récupère les données de partie puis décode l'instance du jeu.

Ensuite, on attache un « listener » pour avoir un suivi des évènements du jeu.

Enfin, on affecte une IA à notre joueur.

vii. Mise à jour de la partie

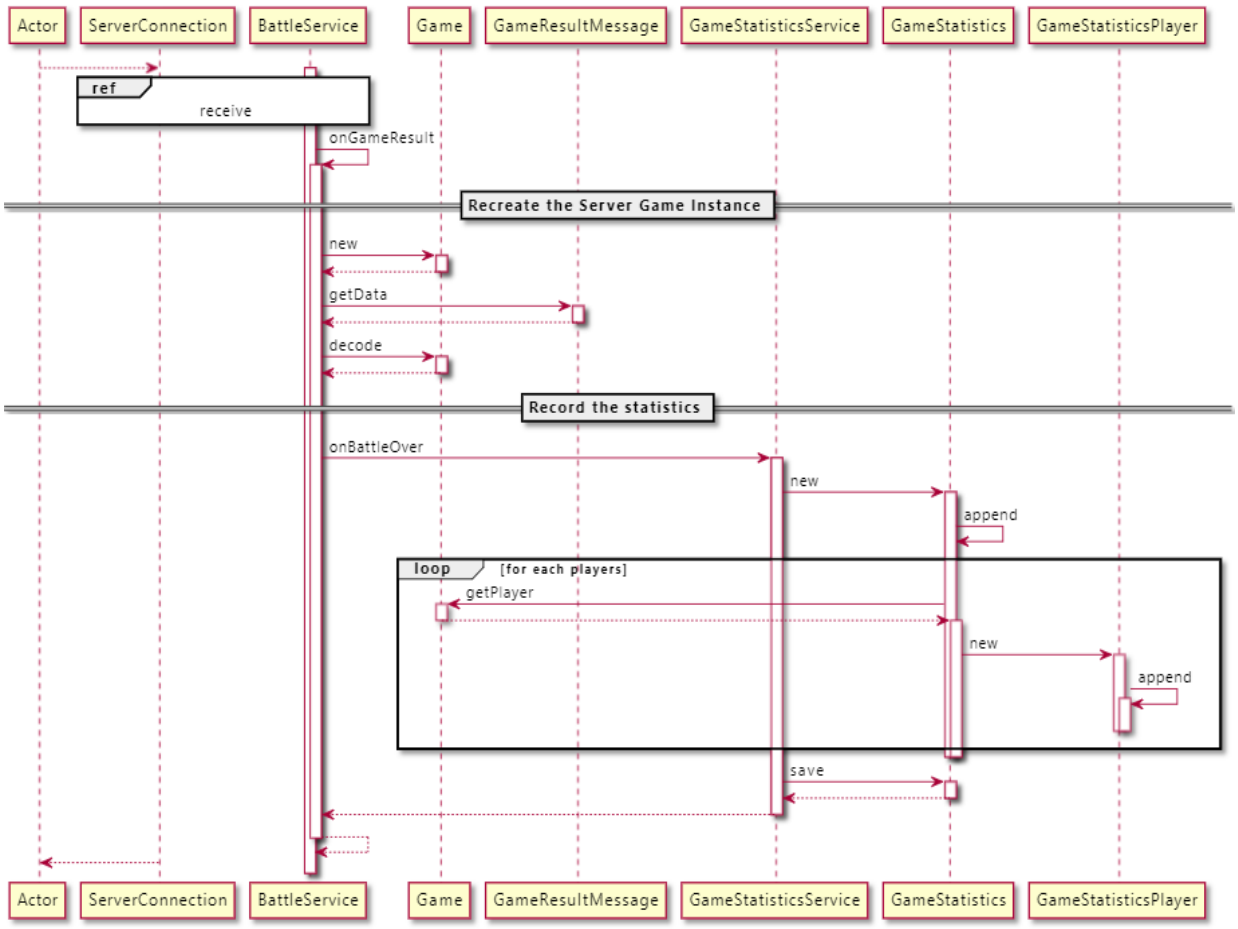
Le serveur met à jour la partie des clients en envoyant le message « GameCommandMessage ». Ce message contient une commande à exécuter sur l'instance Game du client.



Lorsque le client reçoit un message « GameCommandMessage », on récupère la commande puis on l'exécute ce qui met à jour l'état du jeu.

viii. Fin de la partie

La partie est terminée lorsque le client reçoit du serveur le message « GameResultMessage ».



Lorsque le client reçoit le message "GameResultMessage" le client recréé une instance du jeu. On récupère les informations de partie puis on décode l'instance du jeu.

Ensuite, pour chaque joueur on calcule le score.

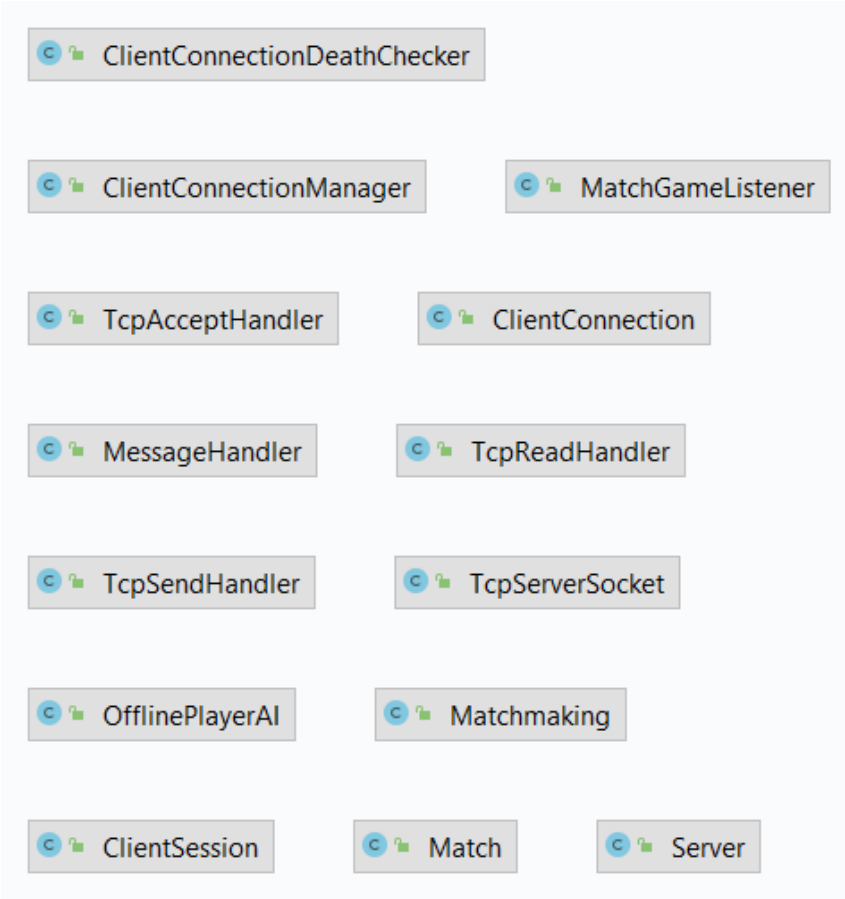
Enfin, on sauvegarde les statistiques dans un fichier Excel.

4. Développement du Serveur

a. Analyse des besoins

L'architecture client / serveur nécessite le développement d'un serveur de jeu. Comme vu précédemment, le serveur est un serveur TCP et qui joue le rôle de maitre de jeu. Le serveur doit supporter plusieurs connexions et fonctionner lors de trafics élevés. Un système de matchmaking devra être présent pour matcher les joueurs entre eux.

b. Diagramme de classes simplifié

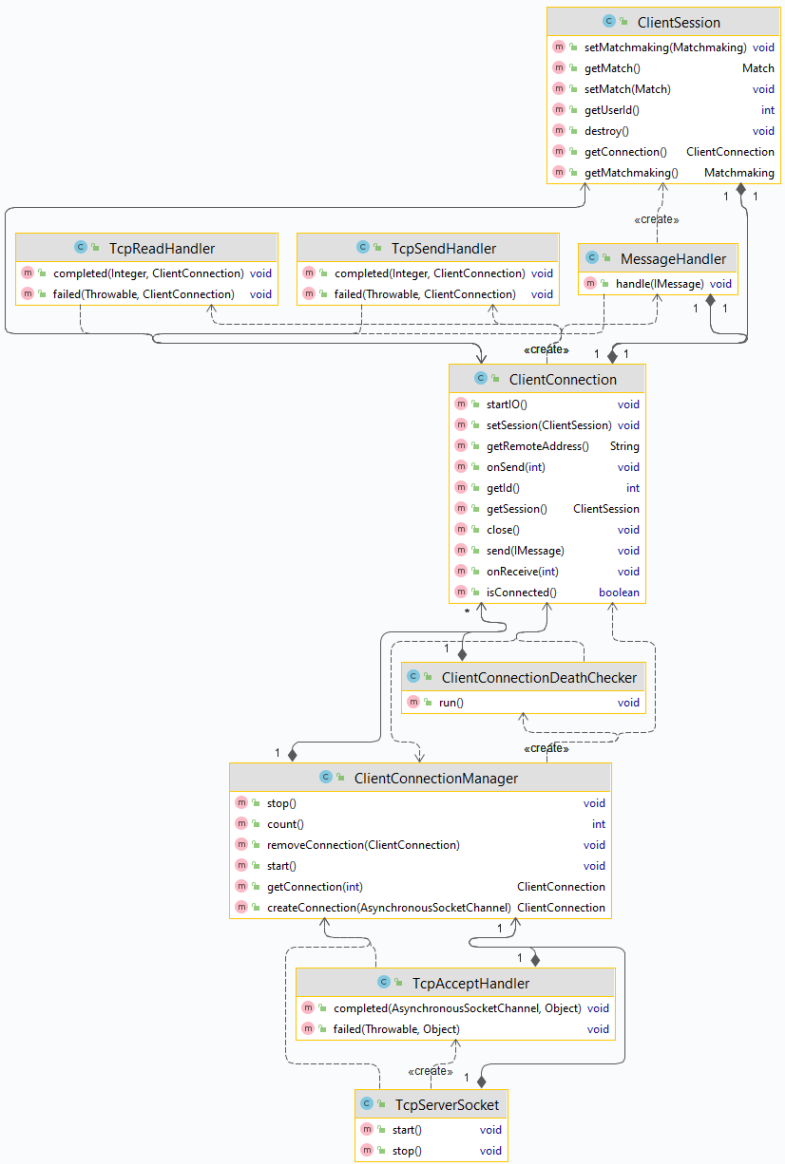


Comme pour les autres projets, le serveur est composé de plusieurs classes que nous allons détaillées plus tard.

« Server » est la classe générale du projet. Elle gère les différents composants du serveur.
« TcpServerSocket » est le socket qui gère l'acceptation des clients.
« ClientConnection » représente une connexion cliente.
« ClientConnectionManager » gère les connexions actuelles.
« ClientConnectionDeathChecker » détruit les connexions mortes.
« TcpAcceptHandler », « TcpReadHandler », « TcpSendHandler » sont les opérations asynchrones pour accepter les connexions, recevoir les données et envoyées données.
« MessageHandler » est la classe responsable du traitement des messages.
« ClientSession » est la session du client créé après l'authentification.
« Matchmaking » permet de créer des matchs entre des joueurs.
« Match » est un match entre plusieurs joueurs.
« MatchGameListener » permet d'être averti des différents évènements de jeu.
« OfflinePlayerAI » permet de simuler les actions d'un joueur lorsqu'il est hors ligne.

c. Diagramme de classes détaillé

i. Réseaux



« TcpServerSocket » est le socket qui gère l'acceptation des clients. Il peut être démarré via « start() » et arrêté par « stop() ».

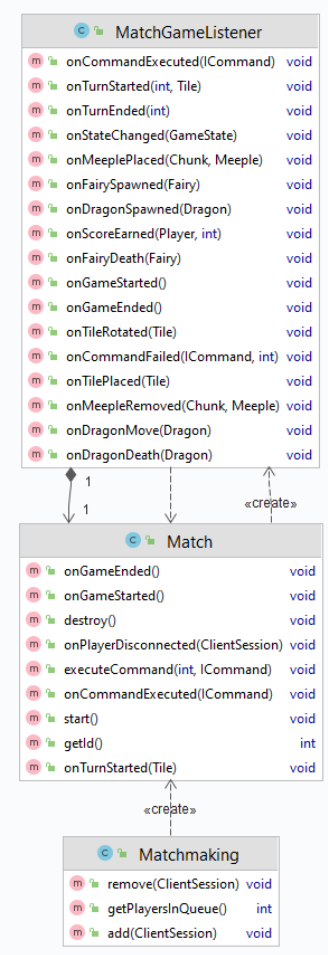
« ClientConnection » représente une connexion cliente. L'objet est créé par « ClientConnectionManager::createConnection(AsynchronousSocketChannel) ».

« ClientConnectionManager » gère les connexions actuelles. Il crée les connexions et détruit les connexions inactive via « ClientConnectionDeathChecker ».

« ClientConnectionDeathChecker » détruit les connexions mortes. Une connexion est dite morte lorsqu'elle est inactive depuis longtemps.

« TcpAcceptHandler », « TcpReadHandler », « TcpSendHandler » sont les opérations asynchrones pour accepter les connexions, recevoir les données et envoyées données.

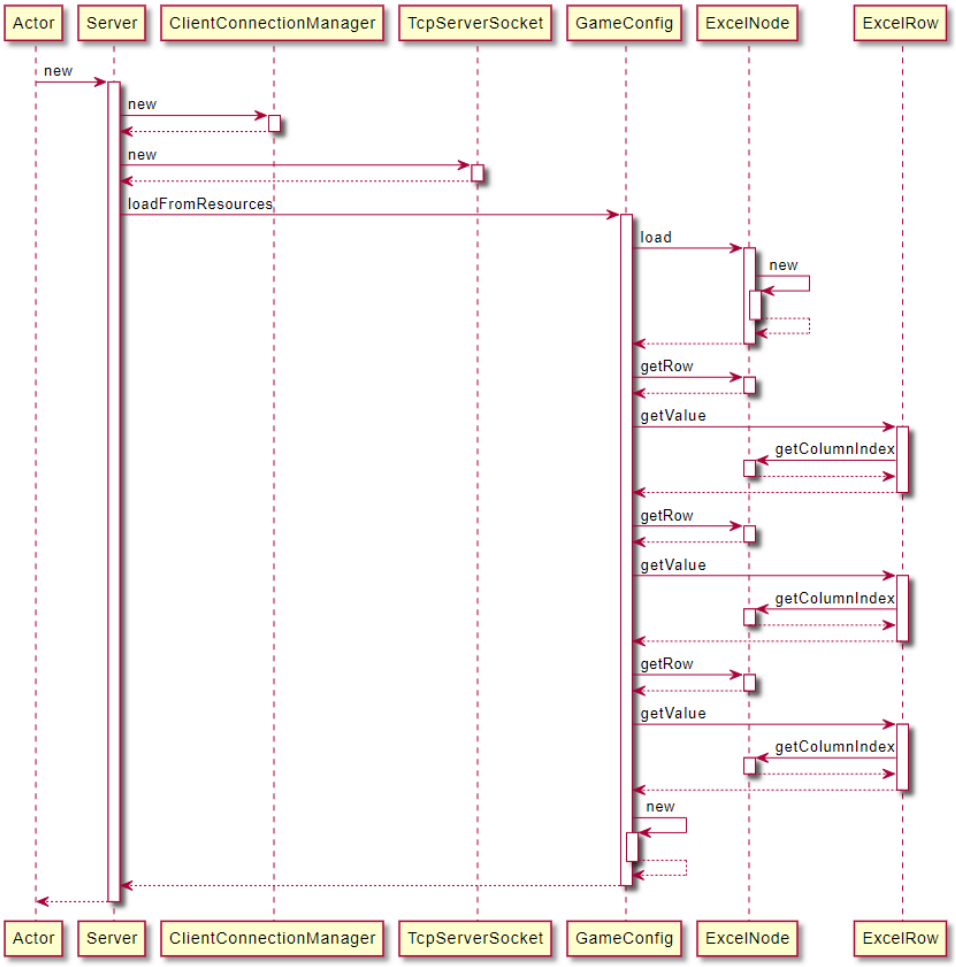
ii. Matches



Les matchs sont créés par le matchmaking lorsqu'il y a assez de joueurs pour qu'ils jouent ensemble. Il peut exister plusieurs instances de matchmaking en fonction du nombre de joueurs qui sont requis pour créer un match.

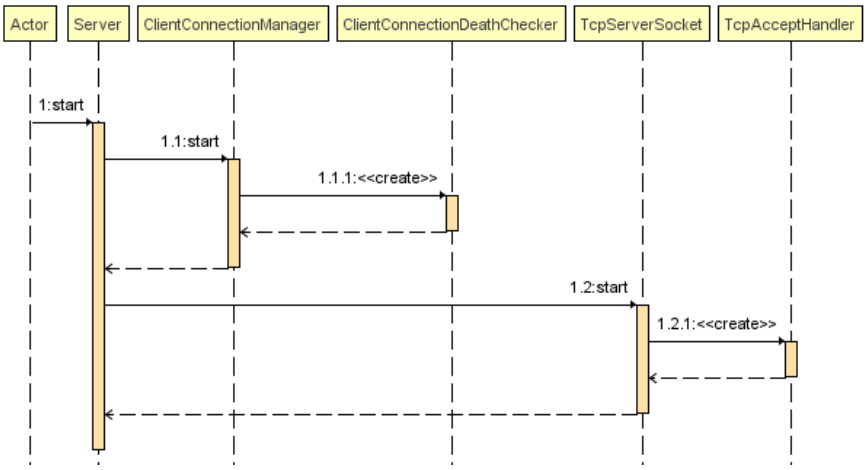
d. Diagramme de séquence

i. Création du serveur



Lorsque l'on créé un serveur, nous créons un « ClientConnectionManager » qui va gérer les connexions. Puis, pour accepter les clients il faut un « TcpServerSocket ». Ensuite, la configuration du jeu est chargée grâce au fichier ExcelNode et ExcelRow.

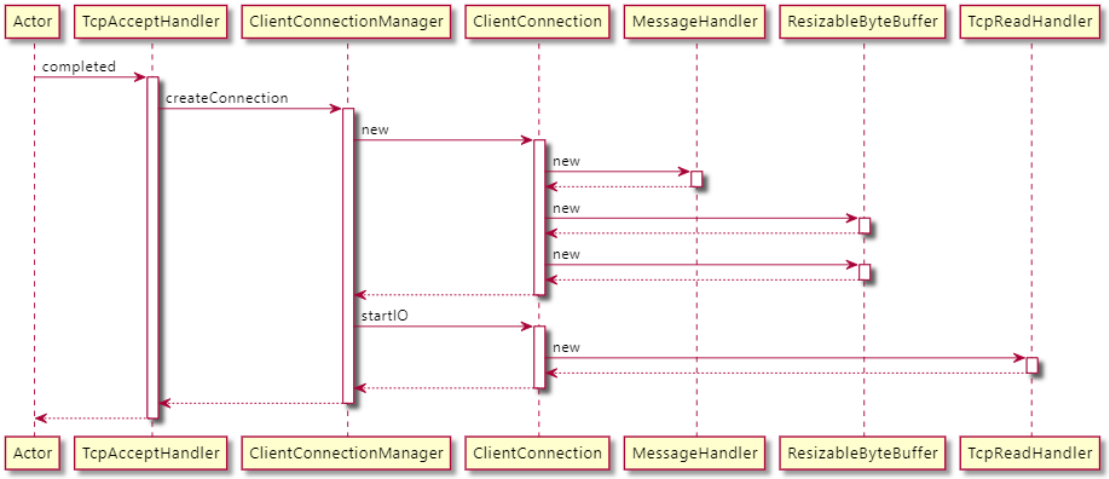
ii. Démarrage du serveur



Lorsque le serveur démarre, on démarre un « ClientConnectionManager » qui va gérer les connexions. Puis nous créons un « ClientConnectionDeathChecker » qui va vérifier si la connexion est fermée alors que nous n'avons pas reçu de notification.

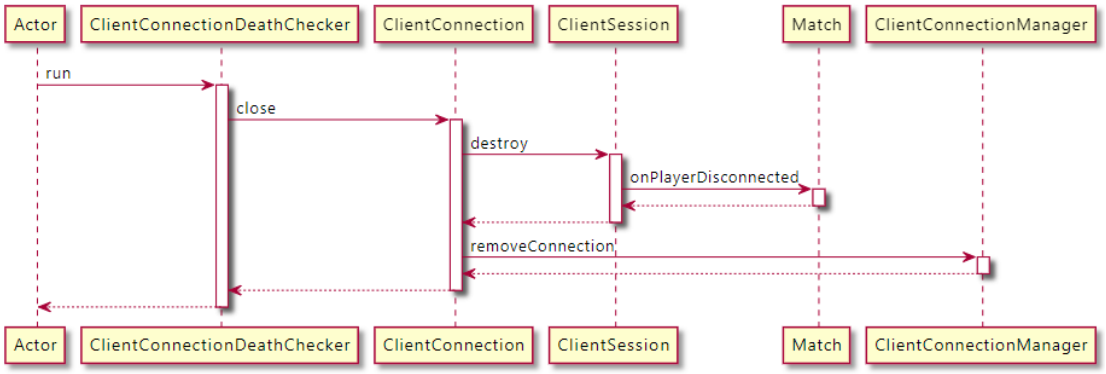
Enfin, on démarre un « TcpServerSocket » qui va créer un « TcpAcceptHandler » pour accepter les connexions.

iii. Acceptation d'une connexion



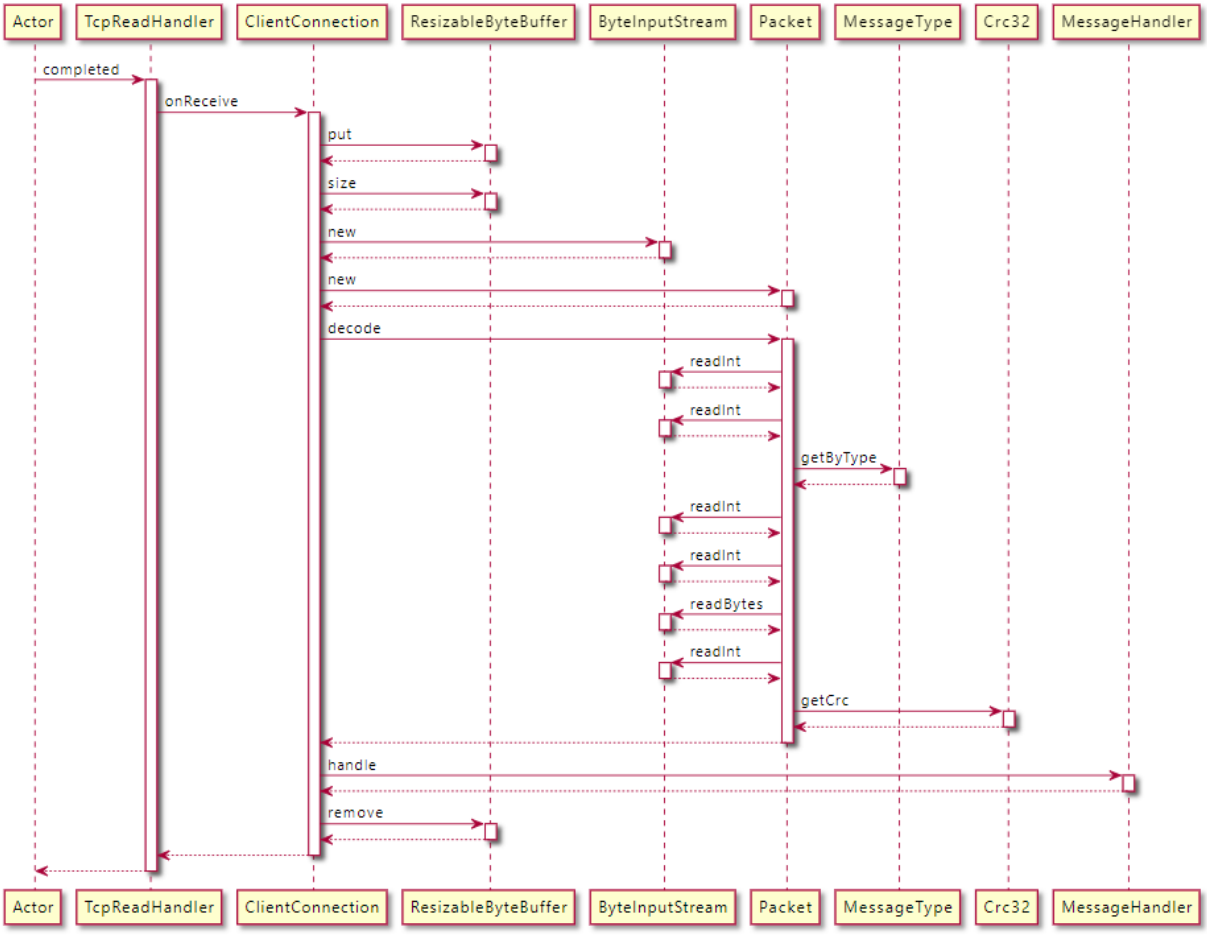
Lorsque le serveur reçoie une demande de connexion, « TcpAcceptHandler » va créer une connexion. Une fois la connexion créée, on initialise les différents composants puis on démarre la réception des données.

iv. Destruction d'une connexion morte



Lorsque le serveur trouve une connexion morte il ferme la connexion client. Il détruit la session du client. Cette destruction va supprimer le client de la partie. Enfin, la connexion est supprimée.

v. Réception d'un message d'une connexion



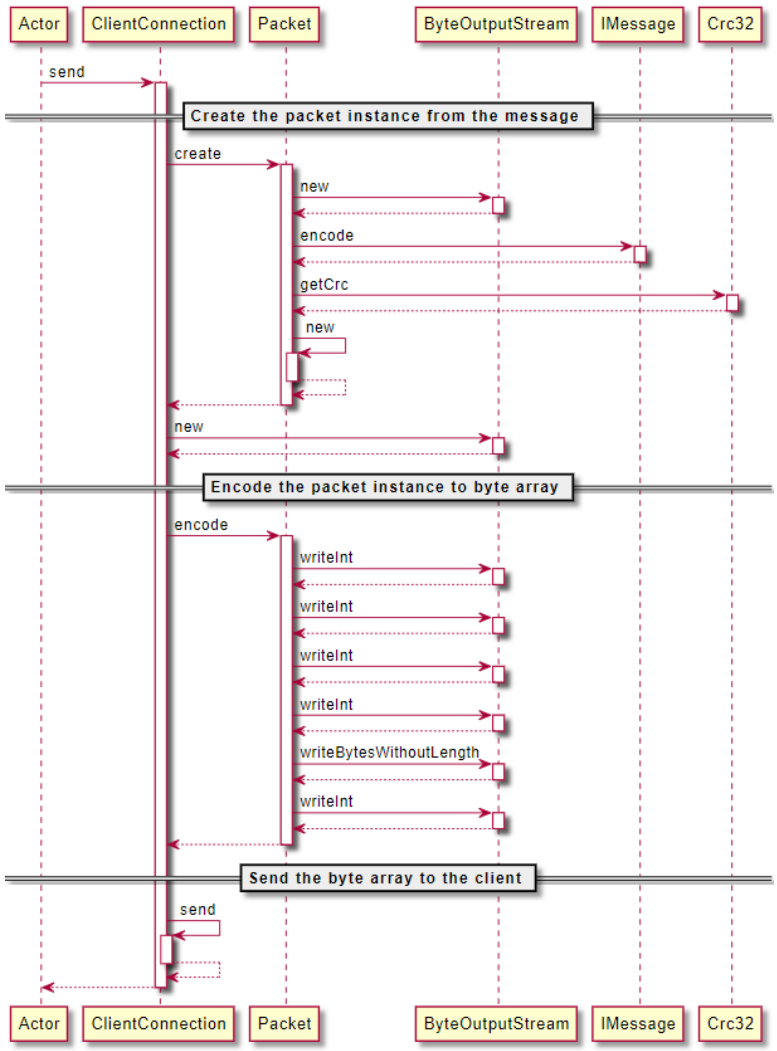
La réception d'un message démarre lorsque que « ClientConnection::onReceive(int) » est appelé par « TcpReadHandler ». Les octets reçus vont être ajoutés à la liste d'octets à traiter.

Ensuite, nous allons décoder les paquets depuis la liste d'octets à traiter.

Par la suite, nous appelons « MessageHandler::handle(IMessage) » qui va se charger de gérer les messages reçus.

Enfin, nous supprimons les octets lus correctement de la liste des octets à traiter et nous redémarrons le processus de réception asynchrone.

vi. Envoi d'un message à une connexion



L'envoi d'un message au client se fait à l'aide de « ClientConnection::send(Message) ».

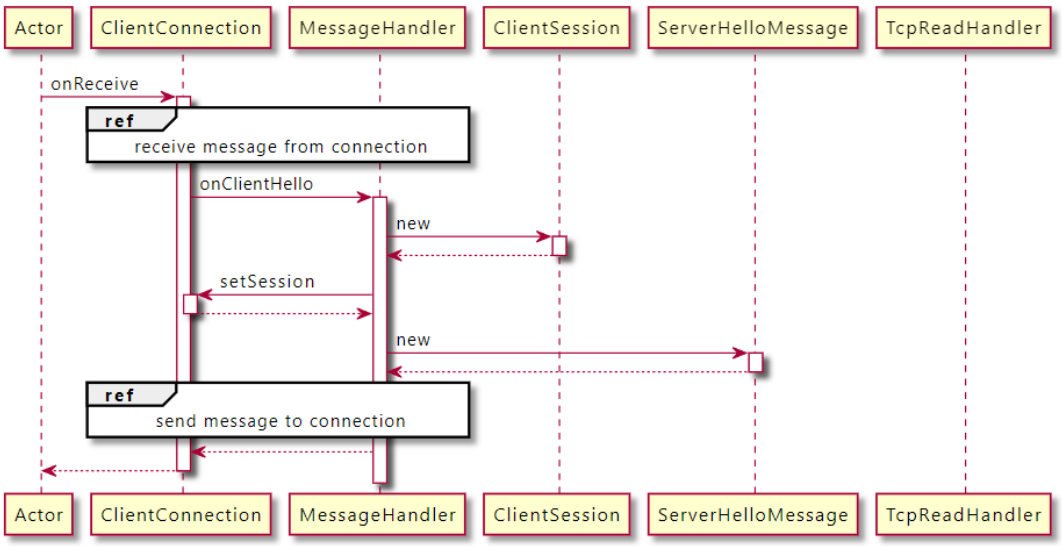
Cette méthode va créer un flux de sortie qui va servir à encoder le paquet en liste d'octets. Ce flux est un objet « ByteOutputStream » qui permet d'écrire des éléments primitifs (byte, int, String) et d'obtenir à la fin une liste d'octets.

Ensuite, nous allons créer un paquet depuis le message qu'on veut envoyer. Cette méthode va encoder le message et ensuite initialiser les attributs du paquet (type de message, longueur, somme de contrôle et contenu).

La somme de contrôle est calculée à l'aide de l'algorithme CRC32. Elle permet de vérifier que le contenu du paquet à bien été lu.

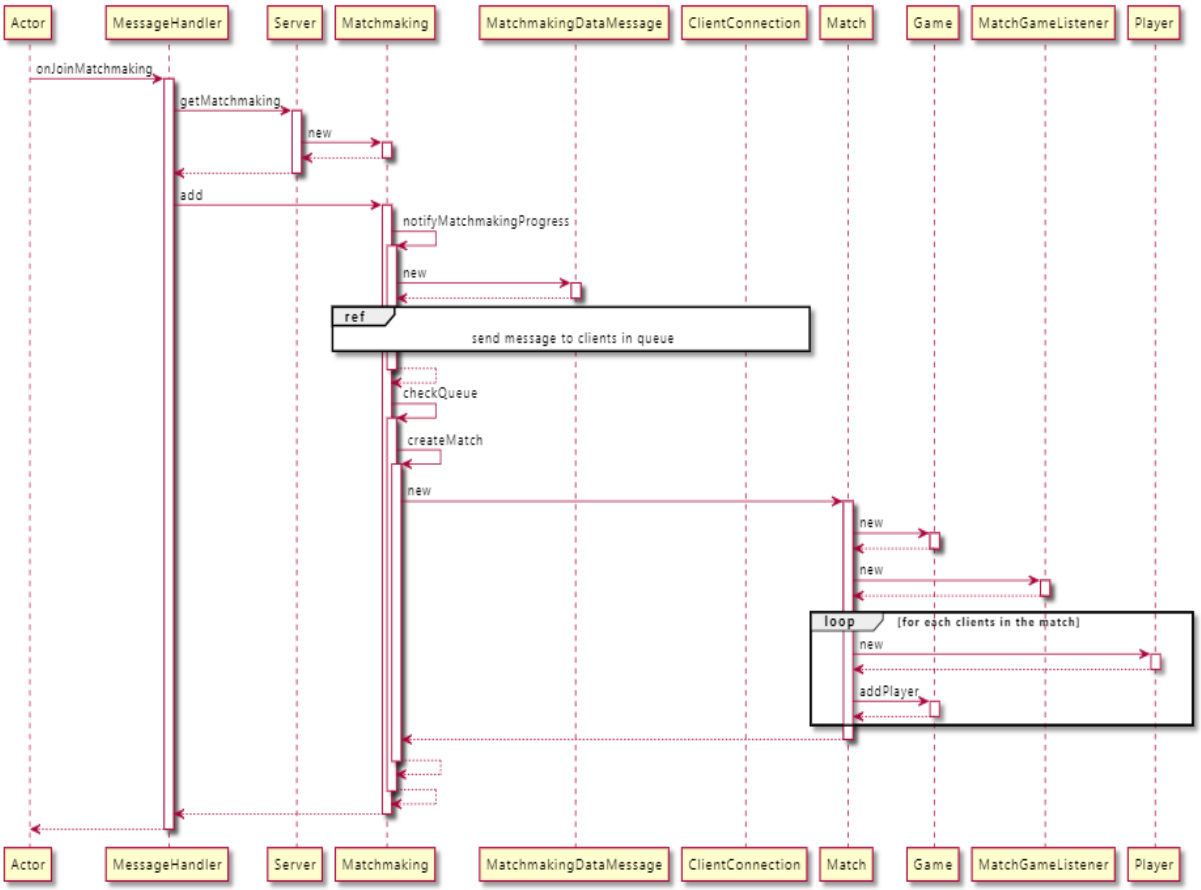
Enfin, le paquet est envoyé au client.

vii. Authentification de la connexion



Lorsque le serveur reçoie un message de connexion du client, une nouvelle session client est créé. Un id d'utilisateur est attribué à la connexion. Enfin, le serveur envoi « ServerHelloMessage » au client.

viii. Rejoindre un match



Le client rejoint un match en envoyant « JoinMatchmakingMessage » au serveur. Lorsque le serveur reçoit ce message, il va obtenir l'instance du matchmaking souhaité par le client.

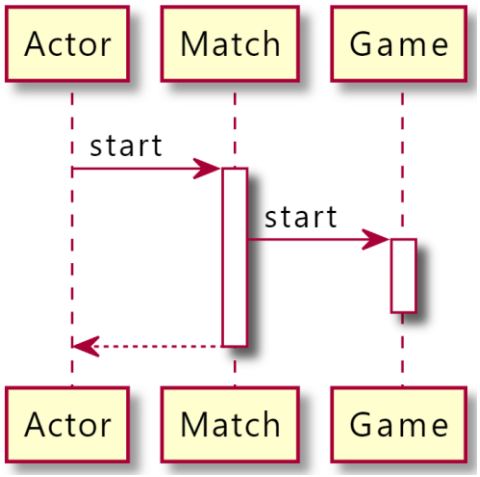
Le joueur va être ajouté à la file d'attente du matchmaking.

Le matchmaking va notifier les joueurs en cours d'attente qu'un nouveau joueur est présente dans le matchmaking.

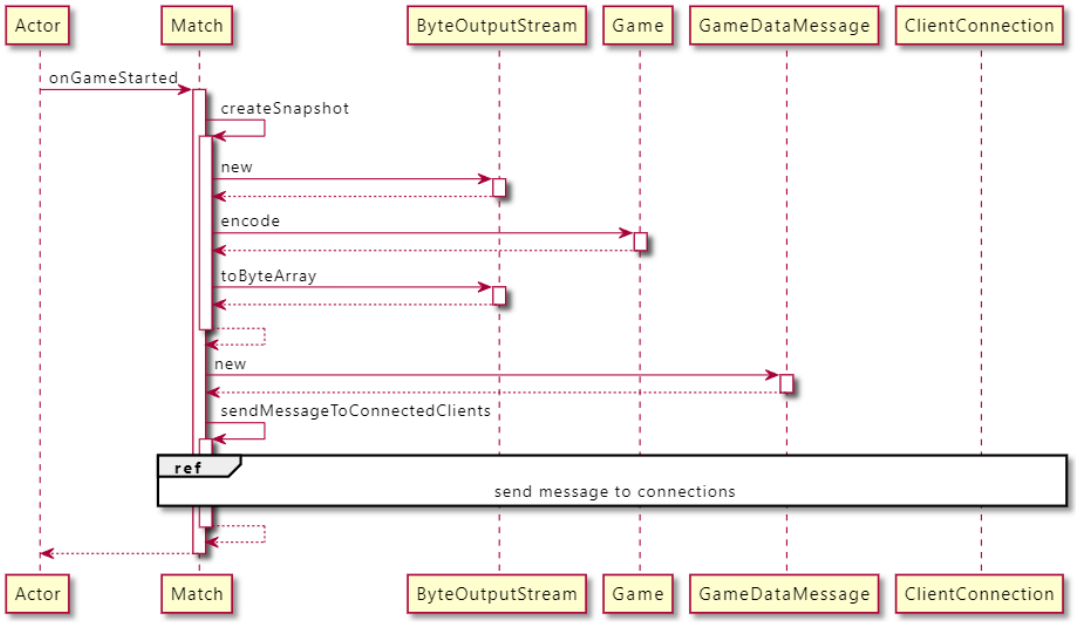
Par la suite, il vérifie s'il y a assez de joueurs pour créer un match. Si oui, il crée un match et supprime les joueurs de la file d'attente.

Enfin, le match va initialiser une instance « Game » est va créer une instance « Player » pour chaque joueur.

ix. Démarrage d'un match

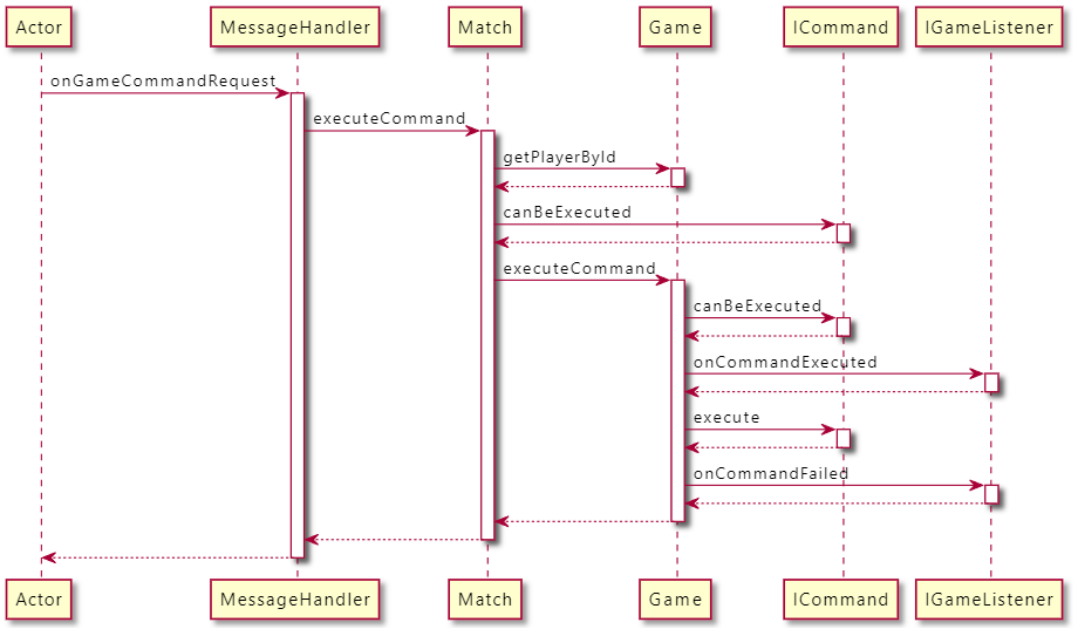


Le match est démarré via la méthode « start() ». Cette méthode va démarrer « Game ».

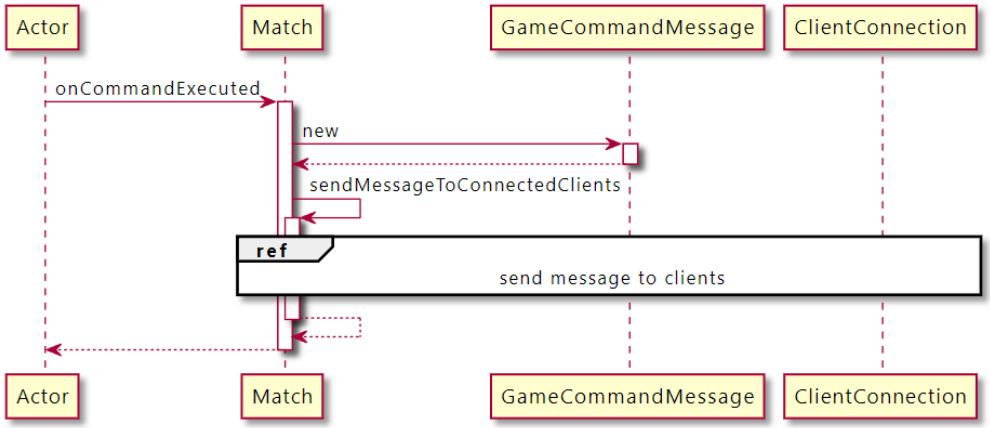


Lorsque le jeu est démarré, un instantané est créé puis envoyé aux joueurs via le message « GameDataMessage » pour qu'ils puissent initialiser à leur tour la partie.

x. Exécution d'une commande sur le match

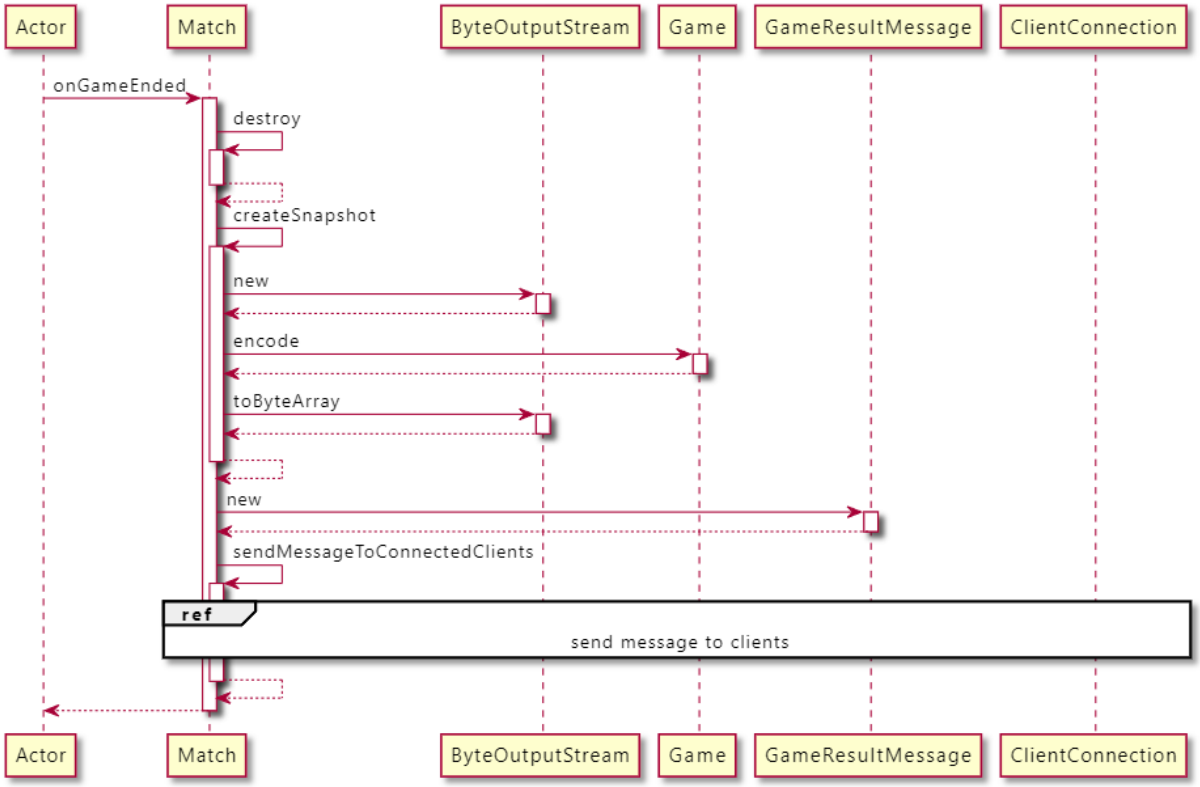


Lorsque « GameCommandRequestMessage » est reçu, le serveur va vérifier si la commande est autorisée durant l'état actuel du jeu. Si elle l'est, alors le serveur va l'exécuter.



Enfin, lorsque la commande est exécutée, les joueurs sont notifiés de l'exécution via l'envoi du message « GameCommandMessage ».

xi. Fin du match



Lorsque que la partie est terminée, nous détruisons les références vers le match. Enfin, nous faisons un instantané de la partie puis l'envoyons aux clients via le message « GameResultMessage ».

5. Analyse du protocole de communication

Dans cette partie, nous allons expliquer le protocole de communication entre le(s) client(s) et le serveur.

Le client et le serveur communique via le protocole TCP. Il permet à l'instar du UDP de garantir la livraison des données et de manière ordonnée. Dans notre conception, il est important que tous les paquets arrivent bien aux clients et au serveur. En effet, si une commande a exécuté ou n'arrive pas dans le bon ordre, alors le jeu peut rester dans un état « figé » (le client n'a pas reçu les commandes pour finir le tour ET/OU le serveur n'a pas les commandes nécessaires pour terminer le tour).

De plus, nous avons besoin de structurer la façon dont les données sont envoyées. Le protocole TCP étant un stream, les données reçus d'un paquet peuvent ne pas être encore totalement reçu ou être fusionnée entre eux (length > buf.length entrainant un fractionnement ou encore algorithme Nagle pour réduire le nombre d'envoi en les groupant). De ce fait, nous avons eu besoin de concevoir une surcouche au protocole TCP, notre protocole de communication : le **CarcaStream** !

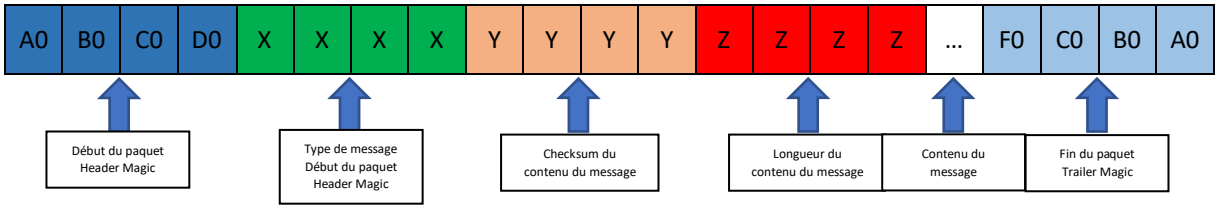
Pour le réalisé, nous avons voulu respecter :

- Le fait qu'il soit peu gourmand en ressources
- Le fait d'être simple de structure

- Le fait d'être facilement debuggable par les ingénieurs du client

Nous en avons conclu une structure « Packet ».

Structure d'un paquet :



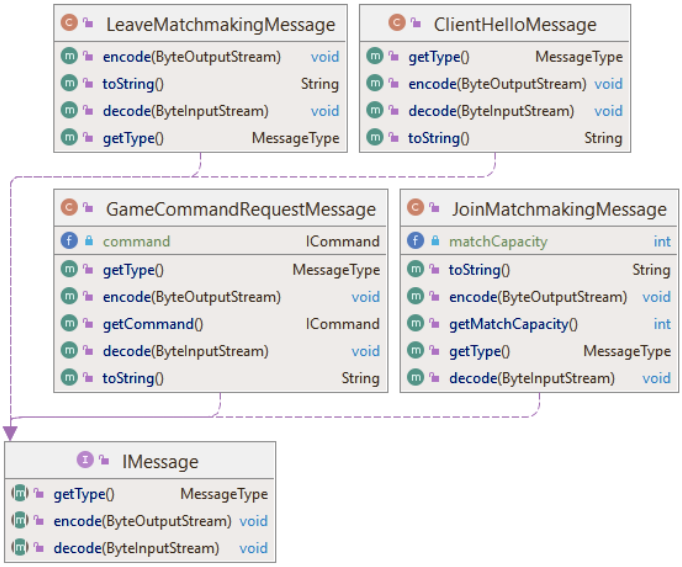
Header magic : Numéro magique permettant de vérifier que les données reçues correspondent bien à un début de paquet
Type de message : Valeur du type de message contenu dans le paquet
Checksum du contenu du message : Somme de contrôle utilisant l'algorithme CRC32 permettant de vérifier que le contenu du message est correct (ou pratiquement, la collision de somme est possible)
Contenu du message : Contenu du message sous forme d'octets obtenu lors de l'encodage
Trailer magic : Numéro magique permettant de vérifier que le paquet a été correctement lu

Exemple de paquets :

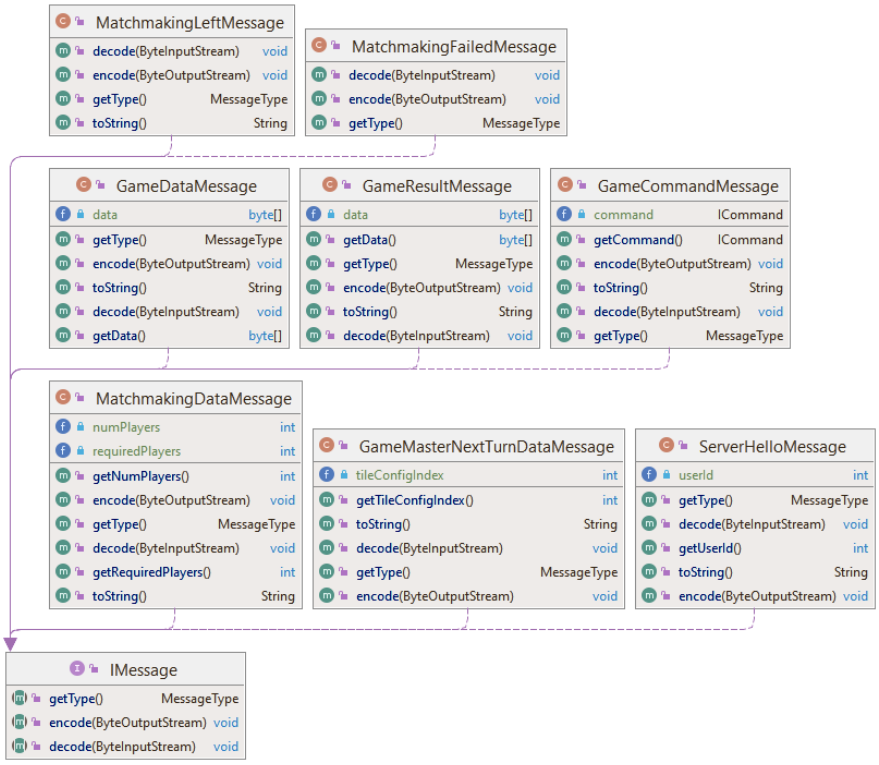
Message dans le paquet	Contenu du paquet
ServerHelloMessage	A0 B0 C0 D0 00 00 00 C8 56 43 EF 8A 00 00 00 04 00 00 00 01 F0 C0 B0 A0
MatchmakingDataMessage	A0 B0 C0 D0 00 00 00 D2 B6 4C 97 F5 00 00 00 08 00 00 00 01 00 00 00 02 F0 C0 B0 A0

Il existe différents types de messages pouvant être envoyé au client et serveur. Nous avons déjà vu la gestion des messages dans la partie « Diagramme de Séquence » du client et serveur mais désormais nous allons voir les échanges au niveau réseau.

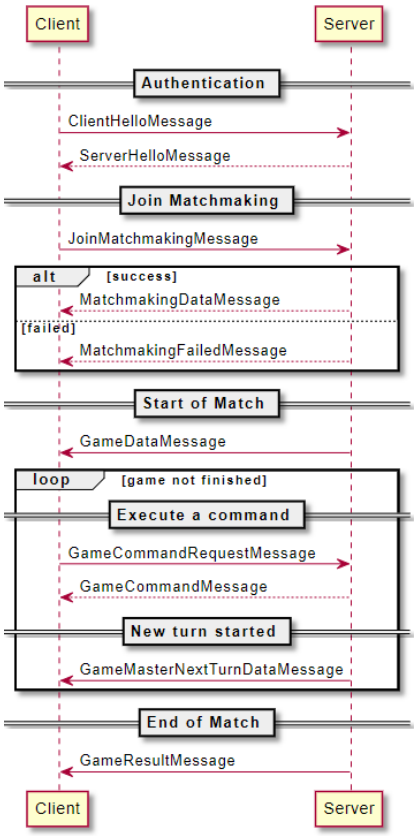
Messages du client



Messages du Serveur



Représentation d'un échange entre un client et un serveur pour une partie de jeu :



La première étape est l'authentification. Le client envoi « ClientHelloMessage » au serveur, puis le serveur réplique par la réponse « ServerHelloMessage ».

Par la suite, le client rejoint un match. Il envoi « JoinMatchmakingMessage » au serveur. Si le joueur a réussi à rejoindre le matchmaking, le serveur va renvoyer « MatchmakingDataMessage » contenant l'évolution du matchmaking. Si la demande a échoué, le serveur va envoyer « MatchmakingFailedMessage ».

Lorsque le matchmaking a réussi, le serveur va envoyer les données de la partie au client via le message « GameDataMessage ».

Le client envoi des commandes au serveur pour mettre à jour l'état de la partie via « GameCommandRequestMessage ». Si la commande est autorisée, le serveur va l'exécuter puis notifier tous les clients de l'exécution via « GameCommandMessage ».

Lorsqu'un nouveau tour démarre, le serveur envoi « GameMasterNextTurnDataMessage » aux clients. Ce message contient la tuile à tirer pour le nouveau tour.

Enfin, lorsque la partie est fini, le message « GameResultMessage » est envoyé aux clients.

Les clients pourront par la suite refaire un match en reprenant la séquence à partir de l'étape « Join Matchmaking ».

6. Conclusion

a. Analyse de la solution

Avant de commencer la phase de développement, nous avons choisi de jouer au jeu afin de bien comprendre chacune des règles et ses spécificités. Bien comprendre le sujet est une étape clé dans le déroulement d'un projet.

Après avoir compris les règles, nous avons tout d'abord donné la priorité sur le développement des éléments qui sont essentiels au fonctionnement du jeu :

- Plateau : Représente le plateau de jeu (cf glossaire). Il contient une liste des tuiles sur le terrain, ainsi que le dragon et la fée.
- Tuile : Représentation des tuiles dans le jeu. Elle contient des chunks qui représente les différents côtés de la tuile.
- Joueur : Représente un joueur de jeu.
- Configuration : Comme Carcassonne existe en plusieurs éditions, nous avons choisi de permettre au client de paramétrer les règles du jeu via la lecture d'un fichier texte (.txt). Cette configuration permet de spécifier le nombre de joueurs minimums et maximums, les différentes tuiles jouables et le gain de score pour chaque interaction.

Pour le développement du jeu, nous avons choisi de nommer nos méthodes et classes en anglais. En effet, un projet en anglais propose des avantages comme l'universalité du code (il peut être compris par tout le monde) et permet de gagner du temps, la langue anglaise étant plus courte syntaxiquement que la langue française.

Nous avons pris l'initiative de répartir notre code dans un nombre conséquent de package et de classe pour faciliter l'ajout de fonctions et la lisibilité des méthodes. Développer de cette façon nous permet de répartir plus facilement le travail et de mieux analyser les problèmes. Cependant, nous avions eu tendance à trop découper en sous-classe (overdesign) ce qui nous a poussé à revoir notre structuration

durant l'itération 3. Aussi, notre approche sur les tuiles n'était pas correcte. Nous avons revu plusieurs fois la conception de ces dernières avant d'arriver à la solution proposée dans le rendu final. On avait initialement réalisé un fichier JSON pour la configuration des tuiles, mais les tuiles demandaient trop d'informations et cela en devenait illisible. On s'est donc orienté vers une solution qui répondaient au mieux à notre besoin : L'utilisation de fichiers Excel.

Chunks						
	Types					
		Name				
			FIELD	FIELD	FIELD	
		FIELD				FIELD
		FIELD		ABBEY		FIELD
		FIELD				FIELD
			FIELD	ROAD	FIELD	
	References					
		Name				
			A	A	A	
		A				A
		A		C		A
		A				A
			A	B	A	
Data						
	Name	Value				
	Model	A				
	Expansion					
	Flags					
	Count		2			

Chaque tuile dispose donc de son fichier au format .txt qui lui est propre avec ses 13 chunks, ses relations entre les chunks, le nombre de tuile ayant cette configuration ainsi que la présence d'élément spéciale (flags) afin de reconnaitre la tuile de départ et les tuiles disposant de blason.

On a pris l'initiative à partir de la livraison 5 de développer une interface graphique complète ce qui nous permet d'avoir une meilleure visibilité sur le développement des tuiles, des placements des Meeple, de la délimitation des zones et facilite grandement le débogage.

Lorsque le client a demandé de réaliser des évolutions, nous avons décidé de toutes les réaliser.

Evolutions à implémenter :

- Implémentation de couleurs pour le texte
- Ajouter l'extension complète "Princesse et Dragon"
- Revoir la structure pour réaliser un client et un serveur

Dès l'itération 5, nous avons mis en place l'architecture client et serveur. Implémenter cette fonctionnalité nous a poussé à revoir la structure complète du code. Nous sommes passé d'un module

à 3 modules distinct (Client, common et server). Common, comme son nom l'indique, dispose du code commun au client et au serveur. C'est ici qu'on a intégré la logique du jeu (moteur) ainsi que les classes Dto (message).

De plus, nous avons ajouté à l'itération 5 les couleurs pour les logs.

Ensuite dès l'itération 6, nous nous sommes penchés sur les règles lié à cette nouvelle extension et nous avons réfléchi à comment implémenter ces nouveaux éléments au sein de notre jeu. Cette itération nous a également permis de finaliser le client et le serveur ainsi que la réalisation d'une nouvelle IA plus performante en lui donnant la possibilité d'effectuer une rotation de la tuile dans le sens des aiguilles d'une montre.

Sur l'itération 7 nous avons développé une nouvelle version de l'Intelligence Artificielle qui évalue via des scores heuristiques chacune des actions avant de les effectuer. Ces actions sont le placement des tuiles en fonctions des tuiles adjacentes ou bien des Meeple afin de marquer le plus de points.

L'itération finale nous a permis d'importer les dernières tuiles comme Tunnel et Portail Magique et de corriger les derniers nos bugs. Enfin, nous en avons profiter pour nettoyer le code.

La version finale du jeu est complète, fonctionne et répond aux attentes du client.

b. Suite du projet

Nous avons implémenté toutes les fonctionnalités demandées ainsi que la fonctionnalité optionnelle a l'exception de l'abbaye dans la ville qui n'a pas encore été réalisé. Le projet pourrait évoluer en donnant la possibilité au joueur de jouer contre l'IA que nous avons développée. Pour cela il faudrait réaliser une interface utilisateur.