Équipe 208

Protocole de communication

Version 1.0

Historique des révisions

Date	Version	Description	Auteur
2023-03-21	1.1	Paquet HTTP + Paquet WS	Hamza et Renel
2023-03-21	1.2	Paquet WS, description des paquets HTTP/WebSocket	Lounes
2023-03-21	1.2	Paquet WS, description des paquets HTTP/WebSocket	Melvis
2023-03-21	1.2	Paquet WS, communication client-serveur WebSocket/HTTP	Yanis
2023-03-21	1.0	Rédaction de l'introduction, et description des paquets HTTP/WebSocket	Daniel

Table des matières

1. Introduction

Dans ce rapport de protocole de communication, nous allons nous pencher sur deux aspects clés du projet. Tout d'abord, nous allons examiner les différents protocoles de communication qui sont utilisés entre les clients et le serveur pour des fonctionnalités spécifiques. Ensuite, nous allons nous intéresser au contenu des différents types de paquets qui sont utilisés dans vos protocoles de communication. Ces deux aspects sont essentiels pour comprendre comment les clients et serveurs communiquent entre eux.

2. Communication client-serveur

Dans cette section du rapport, nous présenterons brièvement les différentes raisons pour lesquelles les protocoles de communication utilisés dans le projet sont nécessaires. De plus, WebSocket a plus été utilisé que HTTP lors de l'implémentation des fonctionnalités dans le cadre du projet.

WebSocket

- WebSocket est utilisé pour la messagerie afin que les joueurs puissent s'échanger des messages et aussi pour les messages d'événements local et global (chat.gateway), il permet une communication beaucoup plus rapide que HTTP et admet une communication bidirectionnelle entre le serveur et le client, ce qui facilite le partage de données.

- WebSocket est utilisé pour le timer afin d'afficher le temps qu'une partie s'est écoulée ou bien sous forme de compte à rebours pour le mode Temps Limité.

- WebSocket est utilisé à la fois pour les parties de jeu en mode solo et un contre un, car nous exploitons les rooms de SocketIO pour placer les joueurs dans une partie de jeu donnée. Ainsi, nous pouvons aisément transmettre de l'information uniquement aux joueurs présents dans la room, si souhaité, comme les différences trouvées par l'adversaire, les messages des joueurs, etc.

- WebSocket est utilisé pour mettre en attente les joueurs qui souhaitent créer/rejoindre un jeu.

- WebSocket est utilisé pour enregistrer et reprendre les actions qui ont eu lieu durant la partie pour la reprise vidéo de la partie

- WebSocket est utilisé pour les fiches de jeu (game-card-handler.gateway)

HTTP

- HTTP est utilisé pour le chargement de l'historique des parties jouées.

- HTTP est utilisé pour la logique du jeu (game.controller), surtout dans le but de charger les jeux de la partie et dans la création de ceux-ci. Par ailleurs, ceci nous permettra d'avoir accès à la liste des différences dans les jeux créés et donc d'implémenter notre logique de détection et validation de différences à partir de WebSocket.

- HTTP est utilisé pour sauvegarder le meilleur temps des parties de jeu et réinitialiser les meilleurs temps d'un jeu (game-record.controller), avec l'aide de notre base de données MongoDB.

- HTTP est utilisé pour gérer l'obtention des indices des parties en solo pour les modes de jeux Classique et Temps Limité (game-record.controller).

- HTTP est utilisé pour configurer les constantes de jeu qui sont hébergées sur notre base de données MongoDB.

3. Description des paquets

WebSocket

- Un message provenant du chat est envoyé en utilisant des sockets. L'événement qui provoque cet envoie est un (click) = “sendMessage”. Celui-ci se trouve dans message-area. On envoie le contenu du message, le nom du joueur, la couleur du message (pour son affichage dans le chat), sa position dans le chat (à gauche ou à droite), le gameId, ainsi que l’event (qui est un booléen).

- Une différence trouvée par un joueur est envoyée en utilisant le protocole WS. L'événement qui déclenche cet envoie est un clique sur une des différences de l’image. Ceci fera appelle à la fonction ‘sendDifference(diff:Set<number>, roomName: string)’ de socketClientService. Cette fonction émettra par la suite un événement ‘feedbackDifference’ au serveur et transmettra la différence sous forme d’un tableau (Array), qui sera par la suite émise dans la room, contenant le joueur adversaire. Cette action permettra au joueur opposant d’avoir une rétroaction des différences trouvées par son adversaire.

- L’évènement de fin de partie est géré à l’aide du protocole WS. L’évènement se déclenche lorsque toutes les différences de la partie ont été trouvées par l’un des joueurs. Ainsi, l'événement ‘gameEnded’ sera envoyé au serveur par l’intermédiaire de socketClientService. Il contiendra le nom de la room (une chaine de caractère), qui sera utile pour le traitement des données du côté serveur, soit la suppression du chronomètre et de la salle et enlever les joueurs dans la salle.

- L'événement d’abandon de la partie est géré par le protocole WS. Celui-ci est déclenché lorsqu’un client décide d'abandonner la partie en cliquant sur le bouton relier à cet effet. Celui-ci lancera l’ouverture d’une modale, contenant la chaîne de caractère ‘giveUp’. Ainsi, l'événement, ‘sendGiveUp’ sera envoyé au serveur avec le nom du joueur voulant abandonner la partie et le nom de la salle dans laquelle il se trouve par l'intermédiaire de socketClientService. Le serveur quant à lui recevra cette information et émettra l’évènement ‘giveup-return’ avec le nom du joueur abandonnant la partie à l’autre joueur présent dans la salle.

- L'événement ‘findDifference’ est enclenché lorsqu’un joueur trouve une différence. Cet événement est envoyé avec le nom de la salle et le nom du joueur ayant trouvé la différence au serveur. Quant à lui, le serveur s’occupera d’émettre l’évènement ‘findDifference-return’ avec le nom du joueur ayant trouvé la différence à l’autre client

dans la même salle. Cette manière de faire permettra de mettre à jour le nombre de différences trouvées par l'adversaire sur la page du joueur présent.

- L'événement pour rejoindre une room Solo est géré par le protocole WS. Celui-ci est déclenché lorsqu' un client crée une gameSolo, c'est-à-dire lorsqu'il valide son nom qu'il a choisi. En effet, lorsque le joueur choisit de créer une partie en mode solo, le gameType dans gameService passe à 'solo'. l'événement joinRoom sera donc envoyé au serveur avec le nom du joueur également par l'intermédiaire socketClientService. On créera donc une room pour le joueur et le joueur rejoindra donc cette Room et pourra jouer.

- L'événement pour se connecter est géré par le protocole WS, il permet de faire un handshake, celui que nous avons implémenté dans chat-gateway est appelé lorsqu' un joueur veut jouer en mode solo et pour celui game-card-handler est appelé quand on veut jouer une partie en multijoueur. On appelle simplement 'this.socket.connect().

- L'événement pour quitter une room est géré par le protocole WS, celui-ci est provoqué quand une partie est terminée. On appelle leaveRoom(socket), le socket correspond au socket de la partie. Donc quand un joueur a fini une partie, on retire le joueur de la room, on retire la room du socket et déconnecte le socket, car la partie n'existe plus.

- L'événement pour envoyer le temps au serveur est géré par le protocole WS, donc dans la fonction afterViewInit qui est dans chat-gateway, on va emit la fonction serverTime qui va emit une map qui a été en array au client chaque seconde pour obtenir l'identifiant du timer et les secondes écoulés.

- L'événement pour arrêter le temps est géré par le protocole WS, donc stopTimer va supprimer le chronomètre du côté du serveur, enlever le temps et va émettre 'gameEnded' pour le joueur donné dans la salle donnée avec un boolean qui est 'true' et le nom du joueur qui a gagné la partie.

- L'événement pour envoyer le nom d'une room est géré par le protocole WS, on envoie un message au serveur avec le nom de la salle que le joueur veut rejoindre. Puis le serveur va rejoindre la room à partir de ce nom, démarrer le timer, et emit 'sendRoomName' au client qui va envoyer une chaîne de caractère qui précise que la partie que la game est de type 'double' et le nom de la salle.

- L'événement ‘Hello’ est géré par le protocole WS, celui-ci va émettre le nom de la room une fois que l'événement joinRoomSolo a été déclenché. Ceci ne prend rien en paramètre et ça émet le nom de la salle au client.
- L’événement startMultiGame est géré par le protocole WS, celui-ci est émis lorsque le créateur de la partie accepte un adversaire pour le rejoindre dans une partie. On prend en paramètre un objet ‘player’, qui contient le gameId, le nom du créateur, le nom du jeu et le nom de l’adversaire. Cet événement est provoqué dans game-card-handler (service). Le serveur va par la suite s’occuper d’associer les deux joueurs pour les mettre dans une salle ensemble afin de jouer.

HTTP

- Sauvegarder un nouveau record se fait par l’entremise du protocole HTTP. La méthode ‘saveGameRecord()’ dans notre service ‘game.service’ est suscribe à la base de données de jeu via ‘createGameRecord’ de ‘gameDataBase’ et envoie un objet gameRecord contenant le nom du jeu (gameName), le mode de jeu soit ‘solo’, soit ‘multi’ (typeGame), le nom du joueur qui a remporté la partie (playerName), la date sous le format ‘HH:MM:SS’ (dateStart) et le temps du nouveau record (time).

- Le protocole utilisé pour l'accès aux cartes de jeux, utilise HTTP, on y trouve un API ‘/games’ ce dernier renvoie comme body un JSON en forme de liste de GameInfo (une interface disponible dans @comman), contenant gameName, imageOriginale, imageModifie, liste de tous les différences, la difficulté aussi que deux listes des meilleurs records (mode solo, mode multiple), deuxièmement un API ‘/game/:gameName ici le jeu demandé retourné comme une instance de l’interface Game (une interface disponible dans @comman), la même que GameInfo sans les listes des records, troisièmement un API ‘delete/:gameName’ elle va supprimer le jeu du serveur aussi que tous ces record. Dernièrement, En cas d’un échec d’un appel précédent, le serveur retourne une réponse avec un code d’erreur (404,409), aussi qu’un message expliquant plus en détaille la cause de l’erreur