308

Protocole de communication

Version 1.5

Historique des révisions

Date	Version	Description	Auteur
2023-03-14	1.0	Première version de la partie client-serveur et de l'introduction.	Thomas Thiboutot 2143877
2023-03-15	1.1	Première esquisse du travail complet	Thomas Thiboutot 2143877
2023-03-19	1.2	Révision des paquets	Thomas Thiboutot 2143877
2023-03-20	1.3	Révision de la communication serveur et la description des paquets	Huy Viet Nguyen 2136374
2023-03-20	1.4	Révision de la qualité de la langue	Benjamin Piché 2151538
2023-04-18	1.5	Correction Sprint 2 et fin sprint 3	Thomas Thiboutot 2143877

Table des matières

Protocole de communication

1. Introduction

Nous allons présenter dans ce document l'architecture du système de communication de notre projet. La première partie de chaque section sera consacrée au Sprint 2 et la deuxième partie sera réservée au Sprint 3. Tout d'abord, dans la partie « Communication client-serveur », nous situerons nos points de communication dans nos fichiers et nous expliquerons nos choix de protocoles dans les différentes parties de notre application. Dans la partie description de paquets, nous décrirons plus en détails les méthodes, interfaces, classes et services qui interagissent entre eux par HTTP ou par WebSocket.

2. Communication client-serveur

2.1 Sprint 2

D'abord, du côté client, nous avons le service *communication.service.ts* qui utilise le protocole de communication HTTP. Ce service est utilisé pour les requêtes de fiches de jeux et la suppression de jeux. Du côté serveur, ces requêtes sont faites au *data.controller*.ts qui utilisent les fonctions du *data.service.ts* pour manipuler les données des jeux. Le service *data.service.ts* est celui qui s'occupe d'accéder aux données des jeux pour les modifier, en ajouter, ou les envoyer au *data.controller.ts* lorsqu'il reçoit une requête du client. Le titre, les meilleurs temps et la difficulté d'un jeu sont situés dans une base de données MongoDB alors que les différences et les images d'un jeu sont situées sur le serveur dynamique. Nous avons choisi HTTP comme protocole, car il ne demande pas de *handshake* avec le serveur et est très bien adapté au transport de données telles que des jeux. Aussi, les WebSockets demandent une connexion constante avec le serveur et le client, alors que nous ne faisons que quelques requêtes au départ pour aller chercher les fiches dans la vue de sélection et quelques autres lors de la création d'une nouvelle fiche. Il serait donc inutile de maintenir une connexion bidirectionnelle lorsqu'on veut seulement accéder, modifier, ou supprimer des jeux.

Ensuite, du côté client, nous avons le service de *socket-client.service.ts* qui met en place les fonctions de bases pour utiliser la communication par SocketIO. Ce service est injecté dans le service de *game-socket.service.ts* qui centralise toute la logique reliée au socket du côté client, c'est-à-dire la messagerie, toute la logique du mode de jeu solo et du mode un contre un. Ce service communique avec le service *games-manager.service.ts du* côté serveur. Nous avons choisi la communication par WebSocket pour la messagerie, car ce protocole est fait spécialement pour les communications simultanées entre plusieurs utilisateurs, car elle utilise le mode « full-duplex » et permet également des envois et des réceptions en temps réel ce qui est très utile pour une application de type messagerie. Le principe de « room » est aussi une des raisons principales pour lesquels nous avons choisis les WebSockets, puisque cela nous permet de prendre deux utilisateurs et de les mettre ensemble à l'écart des autres utilisateurs. Cette caractéristique des sockets facilite la gestion de la messagerie qui doit permettre la communication entre deux clients qui jouent dans une même partie seulement.

Le protocole WebSocket est utilisé pour la logique de jeu solo et un contre un. Pour les mêmes raisons énoncées précédemment, nous avons besoin que la communication se fasse facilement entre les deux clients pour que les messages de jeu ainsi que les changements au « front-end » se produisent en temps réel et que les deux clients reçoivent les notifications simultanément. Certes, nous aurions très bien pu faire le jeu en mode solo avec la communication par HTTP parce que le nombre de requêtes reste assez restreint, mais il était plus facile et plus logique de conserver un même protocole pour les deux modes de jeu.

2.2 Sprint 3

Pour les parties en temps limité nous utilisons le protocole WebSockets car nous avons besoin du temps réel pour la mise à jour du timer, la vérification des MouseEvent, pour la gestion des salles et des abandons. Les WebSockets sont essentiels pour transformer une partie coop dont l'un des joueurs à abandonner. Nous utilisons TCP/IP pour aller chercher les informations de la prochaine partie à afficher.

Pour les constantes de jeu nous utilisons WebSocket pour changer les valeurs sur le serveur lorsque l'utilisateur valide pour permettre le changement dans l'affichage en temps réel.

Pour les changements des meilleurs temps nous utilisons les WebSockets pour l'affichage des temps ainsi que les messages de parties globaux qui s'affichent dans la messagerie car ils doivent être afficher en temps réel avec la date du meilleur temps et que la base des communications à la fin des parties était posés en WebSocket lors du Sprint2. Nous utilisons TCP/IP pour communiquer avec la base de données des meilleurs temps.

Les indices de jeu sont traités par WebSocket, car cette fonctionnalité fait partie de la logique de jeu au même titre qu'un clic sur une différence ou une erreur. Quand un indice est utilisé, il doit y avoir un message de partie, et les messages de partie sont déjà gérés par WebSocket. Par surcroît, l'indicateur d'indices restants, qui est un élément du « front-end », doit être mis à jour en temps réel.

L'historique des parties utilise le protocole HTTP, car ces parties ssont entreposées sur une base de données et une grande partie de la logique pour ajouter et supprimer des parties est déjà implémentée par HTTP. Nous utilisons également WebSocket pour notifier l'affichage de la page d'historique pour en faire une page temps réel ce qui rend cette page plus dynamique et intéressante pour l'utilisateur.

3. Description des paquets

3.1 Constantes et interfaces utilisées

Pour faciliter la compréhension de la description des paquets, les constantes (de développement et

de déploiement) et les interfaces utilisées par le protocole de communication du projet sont décrites ci-dessous.

3.1.1 Constantes

Données de développement :
baseUrl = http://localhost:3000/api

Données de déploiement :
baseUrl = http://ec2-3-99-127-230.ca-central-1.compute.amazonaws.com:3000/api

3.1.2 Interfaces

```
Game {
   title: string; //titre de la partie
   image1: string; // base64
   image2: string; // base64
   differences: Pixel[][]; //tableau de differences
   bestTimes: { solo: {playerName: string; time: number} [];
         versus: {playerName: string; time: number} []}; //objet des meilleurs temps de la partie
   isHard: boolean; // difficulté de la partie
}

Pixel {
   x: number; //position en x du pixel
   y: number; // position en y du pixel
}
PlayingUser {
   gameTitle: string; //titre de la partie
   user: string; //nom de l'utilisateur
}

Message {
   username: string; // nom de l'utilisateur
   text: string;  // texte que contient le message
}
```

PreviewWithoutImage1 {
   title: string; // titre de la fiche
    bestTimes: {solo: {playerName: string; time: number} [];
         versus: {playerName: string; time: number} []}; //objet des meilleurs temps de la partie
   isHard: boolean; //difficulté de la partie
   isInitialValueActive: boolean; // indique si la salle d'attente de la partie à été rejoint
}
PageDetails {
   games: PreviewWithoutImage1[]; // fiches à afficher
   isLastPage: boolean; // true si dernière page
   pageIndex: number; // indice de la page
}

3.1 Paquets HTTP de communication.service.ts

Format d'un paquet HTTP:

Méthodes	Corps – Requête (Optionnel)	Code de retour
Route	Paramètre	Réponse (Optionnel)

Paquets :

createGamePost	game: Game	Succès: 201 (CREATED) Échec: 400 (BAD_REQUEST) => La partie créée par l'utilisateur n'est pas valide
POST baseUrl/data/create	game	

titleExistsGet		Succès: 200 (OK) Échec: 400 (BAD_REQUEST) => Le titre de la partie créée par l'utilisateur est invalide ou n'est pas une string.
GET baseUrl/data/create/titleExists?title=${title}	title:string	boolean

getGameFile		Succès: 200 (OK) Échec: 404 (NOT_FOUND)=> Il n'existe pas de partie avec ce titre
GET baseUrl/data/game/file?title=${title}&image=${original? '1': '2'}	title: string, original: boolean	ArrayBuffer

getGames		Succès: 200 (OK) Échec: 400 (BAD_REQUEST) => l'indice de la page envoyé est soit trop grand ou trop petit ou n'est pas un nombre
GET baseUrl/data/games?pageIndex=${pageIndex}	pageIndex: number	PageDetails

getHistory		Succès: 200 (OK) Échec: 400 (BAD_REQUEST)
GET baseUrl/data/history		GameArchive[]

deleteGame		Succès: 200 (OK) Échec: 404 (NOT_FOUND) => Il n'existe pas de base de données d'historique à cet URI
DELETE baseUrl/data/game?title=${title}	title: string	

deleteAllGames		Succès: 200 (OK) Échec: 400 (BAD_REQUEST) | 404 (NOT_FOUND)
DELETE baseUrl/data/games		

deleteHistory		Succès: 200 (OK) Échec: 400 (BAD_REQUEST) => La requête n'est pas faite au bon endroit
DELETE baseUrl/data/history		

3.2 Paquets WebSocket

Format d'un paquet WebSocket :

Nom	Source	Contenu
Explication		

Paquets :

requestClue	Client (game-socket)	
Envoi une demande d'utilisation d'un indice		

sendClick	Client (game-socket)	P: Pixel
Envoi le pixel ayant été cliqué par le joueur.		

validateUsername	Client (game-socket)	username: string
Demande une vérification du nom d'utilisateur entré.		

startGameVs	Client (game-socket)	playingUser: PlayingUser,
Envoi une requête de démarrage d'une partie 1v1 avec les détails de l'utilisateur voulant démarrer une partie.		

startGameSolo	Client (game-socket)	playingUser: PlayingUser
Envoi une requête de démarrage d'une partie solo avec les détails de l'utilisateur voulant démarrer une partie.		

StartGameLimitedTime + ('Solo' || 'Coop')	Client (game-socket)	username: string
Envoi une requête de démarrage d'une partie en temps limité solo si 'Solo'est le texte du bouton cliqué ou coop si le texte est 'Coop' avec le nom d'utilisateur.		

abandonGame	Client (game-socket)	
Envoi une requête d'abandon au serveur. Les informations de joueur ne sont pas nécessaires car elles sont emmagasinées sur le serveur à la création.		

cancelWaitingRoom	Client (game-socket)	
Ferme le pop-up de la salle d'attente après l'appui sur le bouton annuler.		

acceptPlayer2	Client (game-socket)	data.card : string
Accepte le deuxième joueur voulant rejoindre la partie un contre un.		

joinSelectionRoom	Client (game-socket)	
Demande de rejoindre la salle générale de la page de sélection de jeu.		

chatMessage	Client (game-socket)	text:string
Envoie un message pour être affiché dans la messagerie.		

updateSettings	Client (game-socket)	{initialTime, penaltyTime, bonusTime} : GameSettings
Envoie les nouvelles constantes de jeux validées par l'utilisateur		

resetGameBestTimes	Client (game-socket)	GameTitle: string
Envoie le titre de la partie dont il faut mettre à jour les temps		

resetAllBestTimes	Client (game-socket)	
Envoie l'évènement de réinitialisation de tous les temps.		

kicked	Serveur(game-manager)	
Ejecte le joueur de la salle d'attente		

joinWaitingRoom${player.gameTitle} *${player.gameTitle} est le nom de la partie	Serveur(game-manager)	
Permet de créer puis rejoindre la salle d'attente d'une partie. Ce paquet est envoyé à l'entièreté de la salle principale de sélection pour permettre de mettre à jour les boutons pour le mode 1v1.		

playerRequestJoin	Serveur(game-manager)	Player : PlayingUser
Envoie la demande de duel à la personne dans la salle d'attente de la partie avec le nom du joueur demandant.		

playerRefused	Serveur(game-manager)	
Indique au deuxième joueur qu'il a été refusé.		

gameTimeUpdated	Serveur(game-manager)	Value: number
Met à jour le chrono		

gameCreationError	Serveur(game-manager)	
Émet une erreur de création de la partie.		

gameCreated	Serveur(game-manager)	Game.data: GameInstance
Lance la partie		

usernameValidity	Serveur(game-manager)	available: boolean
Envoi au client un booléen qui indique si le nom d'utilisateur est existant et valide pour débuter une partie.		

settingsChanged	Serveur(game-manager)	Settings: GameSettings
Envoi au client les nouvelles constantes de jeu		

updateBestTime	Serveur(game-manager)	Game: Partial<DatabaseGame>
Envoi au client une fraction de la partie avec le titre et les temps réinitialisés		

updateAllBestTimes	Serveur(game-manager)	BEST_TIME_PLACEHOLDER
Envoi au client les temps réinitialisés pour mettre à jour toutes les parties.		

gameHistoryAdded	Serveur(game-manager)	
Envoi au client le signal de l'historique des parties à été mis à jour		

becameHost	Serveur(game-manager)	
Envoi à un utilisateur en attente qu'il est devenu l'hôte de la salle d'attente de la partie qu'il souhaite rejoindre.		

leaveWaitingRoom${waitingRoom.gameTitle} *waitingRoom.gameTitle est le nom de la fiche sur laquelle le joueur est en attente.	Serveur(game-manager)	
Détruit une salle d'attente dont le nombre d'occupant est de 0;		

gameTimeUpdated	Serveur(game)	++this.time : number
Est envoyé à chaque seconde pour augmenter le temps de chaque côté de la connection.		

errorClick	Serveur(game)	pixel: Pixel
Envoie le signal d'erreur.		

serverMessage	Serveur(game)	Message : string
Envoie un message de jeu à l'utilisateur		

differenceFound	Serveur(game)	diffCoord: Pixel []
Envoie le signal d'une différence trouvée.		

gameFinished	Serveur (game)	(data?: { user: string; abandoned: boolean }
Envoie le signal de fin partie avec le nom d'utilisateur ayant mis fin à la partie et le booleen « abandonned » qui indique si la partie c'est terminé avec un abandon.		

chatMessageToRoom	Serveur(game)	{ sender: this.players[senderId]: string, text: message: string}, otherPlayerSocket : string
Envoie le message de chat à la salle de jeu		

cluePixelPos	Serveur(game)	pixel:Pixel, socketId:string
Envoie le pixel de l'indice		