203

Protocole de communication

Version 2.0

Historique des révisions

Date	Version	Description	Auteur
aaaa-mm-jj	x.x	<Détails précis du travail effectué>	<Nom>
2023-03-17	1.0	Communication Client/Serveur et Description des paquets complétés	Maxime Aspros
2023-04-18	2.0	Révision du document en fonction des rétroactions du Sprint 2	Maxime Aspros

Table des matières

Protocole de communication

1. Introduction

L'intention de ce document est de présenter les solutions apportées par l'équipe 203 pour la communication de la version finale du projet. L'implémentation et les raisons d'utilisation de chacun des moyens de communication sont détaillées dans la première partie et la seconde partie traite la description des paquets utilisés au sein des protocoles de communication.

2. Communication client-serveur

La communication entre le client et le serveur est réalisée avec WebSocket, un protocole à basse latence en temps réel et HTTP, un protocole sans état (chaque requête est indépendante des requêtes qui la précèdent). Pour WebSocket, on utilise plus précisément la librairie Socket.IO, permettant une communication bi-directionnelle en temps réel. Le tableau qui suit contient une justification du moyen de communication choisi pour chaque fonctionnalité.

Fonctionnalité	Moyen de communication
Clavardage et messages de partie (global)	**WebSocket, notamment l'événement 'messsageBetweenPlayers' pour les messages entre les joueurs. Les messages, peu volumineux, doivent être envoyés et reçus en temps réel, ce qui justifie l'utilisation de ce protocole.** Les messages globaux sont initiés par une partie en cours qui envoie un événement au SocketManagerService. La classe MatchManagerServer, contenant la liste des parties en cours, s'occupe de transmettre le message global à toutes les parties concernées.
Création d'un jeu	**Les deux actions qui suivent sont assez volumineuses, puisqu'il s'agit d'envoyer des fichiers .bmp, lancer un algorithme de détection de différences, et stocker des images localement, d'où l'utilisation de HTTP.** Génération de l'image des différences: le client envoie les deux images au serveur dynamique avec une requête HTTP. C'est ImageProcessingController qui s'occupe de traiter les images et retourner au client l'image en noir et blanc. La fiche de jeu n'est pas encore générée, puisqu'il faut d'abord valider le nombre de différences. Envoi du titre du jeu: On envoie une requête HTTP réceptionnée par GamesController. Ensuite, GameStorageService sauvegarde définitivement les images localement et on stocke la fiche de jeu sur MongoDB.
Création, fonctionnement, et enregistrement d'une partie (Mode Classique)	L'utilisateur doit avoir une expérience de jeu fluide et sans délai excessif. Toutes les étapes du déroulement d'une partie sont instantanées et impliquent une communication bi-directionnelle entre le client et le serveur (ex: lorsqu'un joueur clique sur une différence, c'est au serveur de valider ou non cette tentative et de

	fournir une rétroaction aux joueurs d'une partie). **Ces exigences correspondent donc au protocole WebSocket, qui utilise des salles Socket.IO pour envoyer des événements aux joueurs concernés.** Création et enregistrement d'une partie: Une salle Socket.IO est créée à partir du moment où un joueur clique sur "Jouer" ou "Créer" / "Rejoindre" depuis la page de sélection de jeu. Une fois dans la Registration Page, le premier joueur peut entrer son nom. Ensuite, il attend à ce qu'un autre joueur demande de jouer avec lui. Une fois que le *host* (l'hôte est toujours le joueur 1) accepte cette demande, le joueur 2 est définitivement enregistré dans la partie. Plus spécifiquement, c'est la méthode "setMatchPlayer" de la classe MatchManagerService qui s'occupe de cette fonctionnalité. Données du jeu: Chaque fiche de jeu contient un identifiant unique, qui est accessible depuis l'URL de la page Classique. Le client envoie une requête HTTP (contenant l'identifiant unique) afin de récupérer du serveur les données du jeu ('GameData'). Détection d'une différence: L'événement 'validateDifference' de SocketManagerService prend en paramètre un booléen indiquant le joueur responsable du clic. Cela permet d'isoler les interactions des deux joueurs tout en synchronisant les images.
Chargement de l'historique	Chargement de l'historique: **Puisqu'il s'agit d'une requête que l'on effectue une seule fois au chargement de la page, le protocole HTTP est justifiable pour cette fonctionnalité.** Le client envoie une requête HTTP au serveur dynamique. Celui-ci accède à MongoDB et renvoie l'historique au client. Mise à jour de l'historique: **Pour enregistrer le temps où la partie s'est terminée, on privilégie le protocole WebSocket pour envoyer instantanément un message peu volumineux.** Quand une partie vient de finir (et qu'il faut donc mettre à jour l'historique), un événement WebSocket est envoyé et traité par SocketManagerService. Celui-ci appelle une méthode de MatchManagerService, qui demande ensuite à la classe HistoryStorageService de mettre à jour l'historique des parties jouées depuis MongoDB.
Meilleurs temps	Chargement des meilleurs temps: Les informations des meilleurs temps sont conservées dans les fiches de jeu. **Puisque ces fiches sont assez volumineuses (elles contiennent notamment des images ainsi que la liste des différences), il faut donc utiliser une requête HTTP pour les récupérer au chargement de la page (voir GamesDisplay).**

	Mise à jour des meilleurs temps: **La justification du choix de WebSocket pour cette fonctionnalité est identique à celle de l'historique: on veut rapidement envoyer un message avec peu de données.** Les meilleurs temps sont également conservés sur MongoDB.
Constantes de jeu	Puisque les constantes sont globales (partagées par tous les utilisateurs), il faut qu'une partie en cours d'exécution conserve les mêmes constantes, peu importe si un autre joueur les modifie durant le match. **Pour cela, une communication HTTP doit être effectuée au tout début d'une partie.** Pour les modifier, on utilise également une requête HTTP. Les constantes de jeux sont conservées sur MongoDB.
Reprise Vidéo	Pour cette fonctionnalité, on n'utilise aucun des deux moyens de communication mentionnés précédemment, mais plutôt un service injecté (donc partagé par les deux joueurs d'une partie) qui enregistre continuellement les actions à refaire.

3. Description des paquets

Chaque paquet HTTP contient une description de la méthode utilisée, la route, le corps, les paramètres, le contenu dans le corps de la réponse, et le code de retour. Pour les paquets WebSocket, on indique le nom d'événement, la provenance (= source de l'événement: si l'on indique "Client", c'est que c'est le client qui émet l'événement) et le contenu.

Requêtes HTTP:

Méthode: GET

Route: '/api/games/fetchGame/:id'

Corps: Rien

Paramètres: Aucun

Contenu dans le corps de la réponse: Objet GameData ainsi que deux images

Code de retour: 200/404

Méthode: GET

Route: '/api/games/:id'

Corps: Rien

Paramètres: Aucun

Contenu dans le corps de la réponse: Jusqu'à 4 fiches de jeu, ainsi qu'une variable indiquant le nombre de jeux envoyés.

Code de retour: 200/404

Méthode: POST

Route: '/api/games/saveGame'

Corps: Objet GameData ainsi que 2 images

Paramètres:

Contenu dans le corps de la réponse: Le titre du jeu.

Code de retour: 201/404

Méthode: DELETE

Route: ‘/api/games/deleteAllGames’

Corps:

Paramètres:

Contenu dans le corps de la réponse: La taille de la liste des jeux.

Code de retour: 200/404

Méthode: DELETE

Route: ‘/api/games/:id’

Corps:

Paramètres:

Contenu dans le corps de la réponse: La taille de la liste des jeux.

Code de retour: 200/404

Méthode: POST

Route: ‘/api/image_processing/send-image’

Corps: 2 images .bmp

Paramètres:

Contenu dans le corps de la réponse: Une image .bmp en noir et blanc avec les differences, ainsi que le nombre de différences.

Code de retour: 201/404

Evénements Socket:

Nom d'événement: ‘registerGameData’

Provenance: ClassicPageComponent **(client)**

Contenu: Un objet de type GameData est envoyé par le client, et sauvegardé dans socket.data.

Nom d'événement: ‘validateDifference’

Provenance: ClassicPageComponent **(client)**

Contenu: Le client envoie la position du clic, ainsi que la liste des différences trouvées et un booléen indiquant le joueur concerné. Le serveur utilise la méthode getDifferenceIndex de MatchingDifferencesService afin de valider la différence ou non, et met à jour la liste des différences si besoin. La validation est ensuite retournée au client avec l'événement ‘validationReturned’.

Nom d'événement: ‘validationReturned’

Provenance: SocketManagerService **(serveur)**

Contenu: Le serveur renvoie la liste des différences, l’indice de la différence trouvée, et un booléen indiquant le joueur qui a cliqué sur une différence.

Nom d'événement: ‘disconnect’

Provenance: Toutes les pages qui utilisent un Socket **(client)**

Contenu: Aucun

Utilité: Permet de prévenir le MatchManagerService d’une déconnexion afin d'attribuer la victoire à un joueur.

Nom d'événement: ‘createMatch’

Provenance: La fonction ‘createGame’ de MatchmakingService **(client)**. Celle-ci est appelée quand le client appuie sur “Jouer” ou “Créer” depuis la page de sélection de jeu:

Contenu: L’identifiant unique de la fiche de jeu, ainsi que le SocketId de la salle, c’est-a-dire le SocketId du joueur 1 qui lance la partie.

Nom d'événement: 'setMatchType'

Provenance: Page de sélection de jeu, juste après l’appel à ‘createMatch’ **(client)**.

Contenu: L’identifiant de la partie, ainsi que le type de la partie (Solo, 1v1, etc…)

Nom d'événement: ‘setMatchPlayer’

Provenance: La fonction ‘setCurrentMatchPlayer’ de MatchMakingService. Celle-ci est appelée quand un joueur entre son nom dans la RegistrationPage **(client)**.

Contenu: L’identifiant de la partie, ainsi qu’un objet Player, qui contient le playerId et le nom du joueur.

Nom d'événement: ‘setWinner’

Provenance: La fonction ‘onWinGame’ de la page classique, appelee a la fin d’une partie **(client)**.

Contenu: L’identifiant de la partie, ainsi qu’un objet Player, qui contient le playerId et le nom du joueur.

Nom d'événement: ‘setLoser’

Provenance: La fonction ‘onTimerEnd’, utilisée pour signaler une défaite en Temps Limité (une fois que le temps s’est écoulé) de la page classique, appelée à la fin d’une partie **(client)**.

Contenu: L’identifiant de la partie.

Nom d'événement: ‘joinRoom’

Provenance: MatchMakingService**(client)**

Contenu: L’identifiant de la partie que l’on veut rejoindre dans la salle d’attente.

Nom d'événement: ‘requestToJoinMatch’

Provenance: RegistrationPage **(client)**

Contenu: L’identifiant de la partie, ainsi qu’un objet Player, qui contient le playerId et le nom du joueur. Envoie l'événement ‘incomingPlayerRequest’ à la salle du match concerné.

Nom d'événement: ‘incomingPlayerRequest’

Provenance: SocketManagerService **(serveur)**

Contenu: Un objet Player, qui contient le playerId et le nom du joueur. Le client ajoute le joueur à la liste des joueurs en attente.

Nom d'événement: ‘cancelJoinMatch’

Provenance: RegistrationPage **(client)**

Contenu: L’identifiant de la partie, ainsi qu’un objet Player, qui contient le playerId et le nom du joueur. Appelle la fonction ‘sendJoinMatchCancel’.

Nom d'événement: ‘sendIncomingPlayerRequestAnswer’

Provenance: RegistrationPage **(client)**

Contenu: L’identifiant de la partie, un objet Player, et un booléen indiquant si le joueur a été accepté ou non. Envoie l'événement ‘incomingPlayerRequestAnswer’.

Nom d'événement: ‘gameProgressUpdate’

Provenance: SocketManagerService **(serveur)**

Contenu: L’identifiant de la fiche de jeu (gameId) et l’identifiant du match (matchId)

Nom d'événement: ‘incomingPlayerCancel’

Provenance: SocketManagerService **(serveur)**

Contenu: L’identifiant du joueur. Sert à retirer un joueur de la liste des joueurs en attente.

Nom d'événement: ‘deleteAllGames’

Provenance: Page de configuration **(client)**

Contenu: Rien, envoie deux événements (‘allGamesDeleted’ et ‘actionOnGameReloadingThePage’) pour relancer la page avec les jeux.

Nom d'événement: ‘allGamesDeleted’

Provenance: SocketManagerService **(serveur)**

Contenu: Rien

Nom d'événement: ‘actionOnGameReloadingThePage’

Provenance: SocketManagerService **(serveur)**

Contenu: Rien

Nom d'événement: ‘deletedGame’

Provenance: Page de configuration **(client)**

Contenu: Rien, redirige les utilisateurs qui étaient dans la RegistrationPage du jeu supprimé.

Nom d'événement: ‘sendingMessage’

Provenance: ClassicPage **(client)**

Contenu: Le nom de l’utilisateur ayant envoyé le message, le contenu du message, ainsi qu’un booléen indiquant le joueur ayant envoyé le message.

Nom d'événement: ‘resetAllGames’

Provenance: GamesService **(client)**

Contenu: Rien.

Nom d'événement: ‘resetGame’

Provenance: GamesService **(client)**

Contenu: L’identifiant GameId de la fiche de jeu à réinitialiser.

Nom d'événement: ‘allGamesReset’

Provenance: SocketManagerService **(serveur)**

Contenu: Rien.

Nom d'événement: ‘gameReset’

Provenance: SocketManagerService **(serveur)**

Contenu: L’identifiant GameId de la fiche de jeu réinitialisée.

Nom d'événement: ‘gameOver’

Provenance: Page classique **(client)**

Contenu: L’identifiant GameId de la fiche de jeu ainsi qu’un booléen indiquant si le jeu est en un contre un, et enfin un classement (nom, score, et nom du jeu).

Nom d'événement: ‘requestRefreshGameMatchProgress’

Provenance: RegistrationPage **(client)**

Contenu: L’identifiant GameId de la fiche de jeu.

Nom d'événement: ‘requestGetNumberOfGamesOnServer’

Provenance: RegistrationPage **(client)**

Contenu: Rien.

Nom d'événement: ‘numberOfGamesOnServer’

Provenance: SocketManagerService **(serveur)**

Contenu: Le nombre de fiches de jeu actuellement disponibles (type number). Pour le Temps Limité, c’est ce qui permet d’afficher un message au joueur s’il souhaite jouer une partie alors qu’il n’y a pas de fiches de jeu.

Nom d'événement: ‘randomizeGameOrder’

Provenance: ClassicPage **(client)**

Contenu: Pour une partie en Temps Limité, le client demande au serveur de retourner des “seeds” (valeurs generees aléatoirement) pour chaque fiche de jeu disponible. C’est avec ces valeurs aléatoires qu’il est possible de trier la liste des jeux.

Nom d'événement: ‘randomizedOrder’

Provenance: SocketManagerService **(serveur)**

Contenu: Liste de type ‘number’ contenant les “seeds” mentionnées ci-dessus.

Nom d'événement: ‘readyPlayer’

Provenance: ClassicPage **(client)**

Contenu: Booléen indiquant la provenance de l'événement (joueur 1 ou 2).

Nom d'événement: ‘readyUpdate’

Provenance: SocketManagerService **(serveur)**

Contenu: Booléen indiquant la provenance de l'événement (joueur 1 ou 2).

Nom d'événement: ‘startTimer’

Provenance: ClassicPage **(client)**

Contenu: L’identifiant du match ainsi que le temps écoulé.

Nom d'événement: ‘playersSyncTime’

Provenance: SocketManagerService **(serveur)**

Contenu: Le temps écoulé.

Nom d'événement: ‘stopTimer’

Provenance: ClassicPage **(client)**

Contenu: L’identifiant du match.

Nom d'événement: ‘timerStopped’

Provenance: SocketManagerService **(serveur)**

Contenu: Le temps écoulé.

Nom d'événement: ‘matchUpdated’

Provenance: SocketManagerService **(serveur)**

Contenu: L’objet de type ‘Match’ correspondant à l'identifiant fourni par le paramètre de la fonction ‘sendMatchUpdate’.

Nom d'événement: ‘newBreakingScore’

Provenance: SocketManagerService **(serveur)**

Contenu: L’objet de type ‘RankingData’ (pseudo, position, nom du jeu, et type du match) si un joueur décroche un meilleur temps.