109

Protocole de communication

Version 1.0

Historique des révisions

2023-03-18	1.0	Description du protocole de communication HTTP et WS	Asimina Koutsos
2023-03-21	1.0	Ajout de détails pour WS dans la partie 2	Asimina Koutsos
Date	**Version**	**Description**	**Auteur**

Table des matières

Protocole de communication

1. Introduction

La communication se sépare en deux sous-groupes: la communication par protocole HTTP et la communication par les sockets avec le protocole WebSocket (WS). Ci-bas seront détaillés les choix d'utilisation de chaque moyen de communication ainsi que la description des différents types de paquets utilisés au sein de ces protocoles.

2. Communication client-serveur

Le protocole HTTP nécessite une requête du client pour permettre au serveur au serveur de retourner des données. Ce protocole est utilisé pour communiquer avec les jeux sur le serveur. Les jeux sont stockés dans un fichier JSON games.json sur le serveur et on y accède à ces jeux à travers des méthodes HTTP.

Le protocole HTTP est utilisé pour gérer l'affichage pour les jeux stockés. Le client fait une requête pour obtenir tous les jeux et on peut ensuite les afficher sur la page de sélection de jeu et de configuration.

Elle a aussi été utilisée pour gérer les clics des joueurs sur les canevas de jeu. Quand un joueur clique sur un canvas, la position en X et en Y du clic, ainsi que l'id du jeu sont envoyés au serveur, le serveur vérifie si ce clic identifie une différence du jeu et renvoie la réponse au client.

Les jeux sont stockés sous la structure suivante:

```
export interface GameData {
    id: string;
    name: string;
    originalImage: string;
    modifiedImage: string;
    nbDifferences: number;
    differences: Vec2[][];
    isDifficult: boolean;
}
```

Le protocole WebSocket est utilisé pour pour gérer le mode 1v1 incluant le chat. Une grande utilité du protocole WS est le fait que ce protocole est bidirectionnel, ce qui permet donc au serveur de communiquer avec le client sans une requête du client. Ceci permet donc la communication en temps réel. Des fonctionnalités spécifiques que cela permet sont:

- la mise en place de salles pour gérer la communication entre clients dans une partie et dans le chat
- la mise en place d'une salle d'attente pour les joueurs qui veulent jouer en 1v1; puisque le protocole WS permet de garder une trace des clients et de leur envoyer des données, on pourra gérer la queue des joueurs qui attendent d'être acceptés dans un jeu et le serveur pourra leur émettre des événements (comme dans le cas où un 2ème joueur est accepté dans une partie créée.
- la gestion des compteurs de différences des 2 joueurs en mode 1v1. Le serveur émet des événements au client pour gérer l'affichage des 2 compteurs. Aussi, lorsqu'un des joueurs atteint le nombre de différences trouvées nécessaire pour gagner la partie, le serveur peut émettre un événement au client pour lui indiquer qui a gagné.

Il sera aussi utilisé dans les fonctionnalités du sprint 3 comme pour émettre un événement qui indique que le temps s'est écoulé dans le mode Temps Limité.

3. Description des paquets

Description des requêtes HTTP utilisées:

La requête **GET** sur la route /games va retourner la liste de tous les jeux stockés. Dans la réponse de retour sont envoyés le code de retour 200 (OK) et la liste de jeux. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

On obtient les jeux à partir de la méthode getAllGames() qui va lire le fichier games.json contenant les jeux et retourne une promesse avec l'array de jeux (Promise<GameData[]>).

```
this.router.get('/', async (req: Request, res: Response) => {
    try {
        const games: GameData[] = await this.gameService.getAllGames();
        res.status(StatusCodes.OK).send(games);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **GET** sur la route /games/:id va retourner un jeu en fonction de son id. Si le jeu est trouvé, la réponse de retour contient le code de retour 200 (OK) et le jeu. Sinon, la réponse de retour contient le code 404 (NOT FOUND) et un message pour indiquer que le jeu voulu n'existe pas. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est obtenu à partir de la méthode getGameById(id: string); cette méthode obtient la liste de jeux à partir de la méthode getAllGames() et cette liste sera trié avec "find" pour retourner le jeu avec l'id voulu.

```
this.router.get('/:id', async (req: Request, res: Response) => {
    try {
        const game: GameData = await this.gameService.getGamebyId(req.params.id);
        if (game === undefined) {
            res.status(StatusCodes.NOT_FOUND).json('The requested game does not exist');
        } else {
            res.status(StatusCodes.OK).json(game);
        }
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **GET** sur la route /difference/:id est utilisée pour vérifier si un pixel appartient à une différence du jeu lorsque le joueur clique sur le canvas. Dans la requête, req.params.ClickX contient la position en X du clic effectué, req.params.ClickY contient la position en Y du clic effectué et req.params.id contient l'id du jeu auquel le joueur joue. La réponse de retour contient le code de retour 200 (OK) et la réponse sous la forme { result: boolean, index: number}. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

La réponse {result: boolean, index: number} est obtenu à partir de la méthode verificationInPicture(positionX: number, positionY: number, id : string); cette méthode accède à la liste de différences du jeu (champ “differences” dans l’interface GameData) à partir de la méthode getGameById(id). Si les coordonnées du pixel se trouvent dans la liste de différences du jeu, alors la réponse de retour sera { result: true, index: indexDePosition} où indexDePosition sera l’index des coordonnées dans la liste de différences. Si les coordonnées du pixel ne se trouvent pas dans la liste de différences du jeu, alors la réponse de retour sera { result: false, index: -1}.

```
this.router.get('/:id', async (req: Request, res: Response) => {
    try {
        const positionX = Number(req.query.ClickX);
        const positionY = Number(req.query.ClickY);
        const id = req.params.id;
        const response = await this.gameManager.verificationInPicture(positionX,
positionY, id);
        res.status(StatusCodes.OK).send(response);
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **POST** sur la route /games/send va ajouter un nouveau jeu à la liste de jeux. Le corps de la requête contient le jeu à ajouter. Quand le jeu est créé, la réponse de retour contient le code de retour 201 (CREATED) et un message indiquant que le jeu a été ajouté. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est ajouté à l’aide de la méthode addGame(newGame: Gamedata); cette méthode obtient la liste de jeux à partir de la méthode getAllGames() et on ajoute le nouveau jeu à cette liste en effectuant un “push” sur la liste. On écrit ensuite dans le fichier json pour modifier le contenu avec la liste modifiée de jeux.

```
this.router.post('/send', async (req: Request, res: Response) => {
    try {
        const game: GameData = req.body;
        await this.gameService.addGame(game);
        res.status(StatusCodes.CREATED).json('The game was added');
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

La requête **DELETE** sur la route /games/:id va supprimer un jeu de la liste de jeux en fonction de son id. Le params.id dans la requête contient l’id du jeu à supprimer. Si le jeu existe et est supprimé, la réponse de retour contient le code de retour 200 (OK) et un message indiquant que le jeu a été supprimé. Sinon, la réponse de retour contient le code 404 (NOT FOUND) et un message pour indiquer que le jeu voulu n’existe pas et ne peut donc pas être supprimé. Dans le cas où il y a une erreur dans le serveur, on renvoie le code de retour 500 (INTERNAL SERVER ERROR) et le message de l'erreur.

Le jeu est supprimé à l'aide de la méthode deleteGame(id: string); cette méthode obtient la liste de jeux à partir de la méthode getAllGames() et on trie cette liste pour trouver le jeu à supprimer. Si le jeu existe, le jeu est enlevé de la liste de jeux et on écrit ensuite dans le fichier json pour modifier le contenu avec la liste modifiée de jeux. Dans le cas où le jeu existe et on peut la supprimer, la fonction deleteGame(id) retourne true, sinon elle retourne false;

```
this.router.delete('/:id', async (req: Request, res: Response) => {
    try {
        const gameToDelete = await this.gameService.deleteGame(req.params.id);
        if (gameToDelete === false) {
            res.status(StatusCodes.NOT_FOUND).json('The requested game cannot be
deleted as it does not exist');
        } else {
            res.status(StatusCodes.OK).json('The game with the requested id has been
deleted');
        }
    } catch (error) {
        res.status(StatusCodes.INTERNAL_SERVER_ERROR).send(error.message);
    }
});
```

Description des évènements du protocole WebSocket:

Plusieurs événements sont transmis entre le client et le serveur. Ci-bas se trouve la description de la gestion des différents événements par le serveur:

- **'startTimer':** le serveur capte cet événement qui est émis par le client lorsqu'une partie de jeu est commencée. Le serveur lance alors le chronomètre de la partie.

- **'getRealTime':** le serveur émet cet événement au client pour lui indiquer le temps actuel du timer qui est en cours. Il contient l'information du timer à afficher soit les minutes et secondes écoulés (affichés dans la vue sous cette forme XX:XX)

- **'createLobby':** lorsqu'un client joint un jeu pour jouer dans le mode 1v1 en premier et est mis dans la salle d'attente, il émet cet événement au serveur avec comme contenu l'id du jeu qu'il attend et son nom de joueur. Le serveur va capter l'événement et crée une salle; le socket qui a émis l'événement est alors ajouté à la salle comme host. Ça permet donc par la suite de regrouper les 2 sockets qui jouent dans le mode 1v1 ensemble dans une salle.

- **'joinQueue':** lorsqu'un deuxième client joint un jeu pour jouer dans le mode 1v1, il émet cet événement avec comme contenu l'id du jeu à joindre et son nom de joueur pour pouvoir joindre la queue de joueurs qui attendent d'être acceptés par un host dans un jeu 1v1. Le serveur va capter cet événement, ajouter le nom du joueur dans la queue et émet a son tour l'événement **'updateQueue'** au client qui est host de la partie de jeu avec comme contenu la queue qui a été mise à jour. Le client qui hoste la partie capte l'événement 'updateQueue' et met donc à jour sa queue de joueurs en attente pour joindre sa partie.

- **'checkPlayersInGame':** le client émet cet événement avec l'id du jeu voulu en contenu pour savoir quels joueurs se trouvent dans un jeu particulier. Le serveur capte l'événement et vérifie s'il y a des salles qui existent pour ce jeu. Il émet ensuite l'événement **'playersInGame'** qui contient le nombre de joueurs dans la salle de jeu s'il y a une salle existante ou 0 comme nombre de joueurs s'il n'y a pas de salle. Le client va ensuite capter cet événement et stocke le nombre de joueurs dans les informations de chaque jeu.

- **'disconnect':** lorsqu'un client quitte la salle d'attente pour retourner dans la page de sélection de jeux, cet événement est envoyé au serveur qui le capte et enlève ce socket de la liste de salles.

- **'removeFromQueue':** le client émet cet événement dans 2 situations:
    - Lorsque le host d'un jeu reçoit l'invitation d'un 2ème joueur qui veut rejoindre la partie que le host a créé, le host peut refuser ce joueur et attendre quelqu'un d'autre. Lorsque le host le refuse, il envoie l'événement **'removeFromQueue'** au serveur.
    - Lorsqu'un joueur est dans la salle d'attente et attend d'être accepté par un host d'une partie de jeu existante, il peut décider de quitter la salle d'attente pour retourner dans la page de sélécetion de jeux. Si c'est le cas, il envoie l'événement **'removeFromQueue'** quand il quitte la salle d'attente.
    - Dans les 2 cas, l'événement est émis avec comme contenu l'id du socket à enlever de la queue et l'id du jeu

  Lorsque le serveur capte cet événement et enlève le socket de la queue de joueurs en attente. Le serveur émet donc l'évènement **'refused'** qui sera capté par le client qui a été enlevé de la queue et ce dernier est redirigé vers la page de sélection de jeux. Le serveur émet ensuite l'événement **'updateQueue'** à l'host qui a créé la partie de ce jeu avec comme contenu la liste mise à jour des joueurs en attente pour joindre sa partie/salle.

- **'addToRoom':** Lorsque le host d'un jeu reçoit l'invitation d'un 2ème joueur qui veut rejoindre la partie que le host a créé, le host peut accepter ou refuser ce joueur. Lorsque le host accepte, il envoie l'événement **'addToRoom'** au serveur avec comme contenu l'id du socket du 2ème joueur à ajouter dans la salle de la partie de jeu.

  Le serveur capte cet événement et ajoute l'id du socket dans la salle du jeu. Il émet ensuite l'événement **'refused'** à tous les autres sockets dans la queue d'attente pour joindre la partie. Les clients captent l'événement **'refused'** qui les redirige vers la page de sélection de jeux.

  Le serveur émet ensuite l'événement **'goToGame'** avec comme contenu l'id de la salle dans laquelle se trouvent les 2 joueurs qui vont jouer 1v1. Les 2 clients captent cet événement qui les dirige vers la page de jeu 1v1.

  En dernier, le serveur émet l'événement **'username'** avec comme contenu les noms des deux joueurs, soit le host et l'invité. L'événement est capté par le client dans la page de jeu 1v1 qui va attribuer les noms des 2 joueurs à des attributs user1 et user2 de la page. On va donc pouvoir utiliser ces 2 attributs dans le fichier html de la page 1v1 pour afficher les noms des 2 joueurs.

- **'differenceFound'**: lorsqu'un joueur trouve une différence dans le jeu, il émet cet événement avec comme contenu l'id de la salle et l'id de la différence identifiée. Le serveur capte ensuite l'événement et incrémente un compteur du nombre de différences trouvées par le joueur; si le host a émis l'événement, c'est son compteur qui est incrémenté, sinon c'est celui de l'invité. Le serveur va émettre l'événement **'differenceUpdate'** à la salle où se trouvent les 2 joueurs avec comme contenu les valeurs mise à jour des 2 compteurs de différences de l'host et de l'invité ainsi que l'id de la différence. Cet événement est capté du côté client et les valeurs du nombre de différences trouvées par chaque joueur sont mises à jour. Le client vérifie ensuite si un des 2 joueurs a gagné la partie.

- **'giveUp':** lorsqu'un des 2 joueurs veut abandonner la partie en cours, il émet cet événement avec l'id de la salle comme contenu. Le serveur capte l'événement et vérifie l'id de quel joueur a émis l'événement. Le serveur émet ensuite à l'id du socket de l'autre joueur l'événement **'win'**. Le joueur qui a gagné la partie capte cet événement du côté client et un message de félicitations apparaît pour celui-ci.

Lors du sprint 3, il y aura plusieurs autres événements qui devront être ajoutés. Par exemple:

- Un événement qui sera émis pour indiquer que le timer s'est écoulé dans le mode Temps Limité
- Un événement pour envoyer les nouveau meilleurs temps de chaque jeu
- Un événement pour ajouter ou retirer du temps du timer dans le mode Temps Limité (par exemple quand quelqu'un demande un indice et une pénalité de temps s'ajoute