206

Protocole de communication

Version 1.6

Historique des révisions

Date	Version	Description	Auteur
2023-10-28	1.0	Rédaction Introduction et communication client-serveur	Amira Tamakloe
2023-10-28	1.1	Rédaction des interfaces utilisés	Cédrick Nicolas
2023-10-28	1.2	Rédaction des paquets websocket côté serveur	Mélissa Hammache
2023-10-29	1.3	Rédaction des paquets HTTPs client-serveur	Jacob Ducas
2023-11-02	1.4	Révision des sections	Olivier Tremblay-Noël
2023-11-05	1.5	Rédaction des paquets websocket côté client	Clémentine Chailleux
2023-11-06	1.6	Rédaction des interfaces utilisés	Cédrick Nicolas
2023-11-07	1.7	Correction des paquets websocket	Mélissa Hammache

Table des matières

Protocole de communication

1. Introduction

Le document relate des différents formats de communication entre un client et le serveur choisi dans le contexte de notre application web à l'aide de justifications appropriées pour chaque cas spécifique. Les raisons du choix de l'implémentation d'un protocole au dépend d'un autre pour différentes fonctionnalités seront détaillés ainsi que l'utilité du protocole dans ce cas. La dernière description des paquets détaille le contenu des différents types de paquets utilisés pour interagir dans notre application pour les protocoles de communication utilisés. La description du paquet permet de clarifier les différents APIs du serveur dynamique ainsi que les différentes communications réseau de notre application. Nous inclurons également toute autre information pertinente pour la compréhension des paquets échangés dans le cadre de notre application.

2. Communication client-serveur

La communication client-serveur du projet se fait à travers des requêtes HTTP et par le protocole WebSocket. Il est important que le protocole Websocket soit majoritairement utilisé afin de permettre une communication bidirectionnelle entre le client et le serveur ainsi qu'une réactivité quasi instantané pour garantir que l'exécution des parties multijoueur soit fluide. Les requêtes brèves ne nécessitant pas une communication persistante entre le serveur et le client ont été implémentés grâce au protocole HTTP. En revanche les requêtes ayant besoin de persistance, c'est-à-dire d'une communication constante entre le client et le serveur, et qui nécessitent que le serveur puisse envoyer des informations au client sans que celui-ci n'en fasse la requête ont été implémentés à l'aide de WebSocket. Trivia Go est une application qui nécessite une forte réactivité d'ou l'usage majoritaire de websocket qui est un protocole plus réactif.

Il est important de noter que pour le protocole WebSocket, nous avons choisi la librairie Socket.IO, qui est un wrapper de haut niveau pour la communication par le protocole WebSocket. En effet, les méthodes offertes par Socket.IO simplifient les processus de connexion et de communication entre les différents sockets. Il est à noter que dans le cas du serveur autant que dans le cas du client, les deux systèmes doivent utiliser la même librairie, car WebSocket et Socker.IO ne sont pas compatibles, bien que le dernier soit bâti à partir du premier.

Le clavardage, notamment, utilise le protocole Socket.IO puisqu'une communication en temps réel entre le serveur et le client est nécessaire. En utilisant Socket.IO, le client n'a pas besoin d'envoyer une requête à chaque fois qu'il souhaite communiquer avec le serveur, puisque la liaison est déjà établie. On réduit de ce fait la latence de notre application pour recevoir l'application. Le serveur renvoie des évènements directement au client comme les messages d'autres joueurs et des événements de partie ce qui est impossible avec l'utilisation de HTTP. L'envoi de l'heure auquel un événement s'est produit dans le clavardage utilise aussi Socket.IO puisqu'une liaison constante doit être établie afin que le serveur puisse déterminer quand est-ce que l'événement s'est produit depuis l'initialisation de la liaison. Pour que les participants d'une même partie puissent être liés ensemble dans leurs

communications, ils partagent ensemble une salle Socket.IO qu'ils joignent au moment de la connexion à une partie.

La gestion des salles d'attentes, rejoindre une partie, les informations venant des joueurs (pointage, réponse, abandon, et les données de l'histogramme), le compte à rebours interactif sont des actions qui nécessitent une écoute active du serveur et sa propagation à l'organisateur ou d'autres joueurs sans que ceux-ci ne réquisitionne l'information. Ainsi, Socket.IO est le protocole a priorisé pour les besoins en termes de réactivité du serveur lors de l'usage de ces fonctionnalités. De plus, toutes ces actions nécessitent une réponse en temps réel et une certaine synchronisation à travers tous les joueurs d'où le choix de Socket.IO.

Le protocole HTTP a été le choix privilégié pour toutes requêtes qui nécessite un accès aux informations stockées sur MongoDB. Les différentes fonctionnalités nécessitant un accès à la base de données sont la création d'un jeu, la suppression d'un jeu, le changement de visibilité d'un jeu par l'admin, la modification d'un jeu par l'admin, la récupération d'un jeu spécifique, l'ensemble des jeux ou uniquement les jeux ayant l'attribut de visibilité mis à true. Le client envoie une requête HTTP qui sera gérée par Node.js et Express qui récupéreront l'information souhaitée depuis la base de données et qui la renverront au client en réponse à la requête http. Le protocole HTTP est utilisé pour toutes ces requêtes car le client a immédiatement besoin d'un retour du serveur suite à sa requête. En somme, HTTP est le protocole d'usage pour envoyer au client les données que le serveur va chercher dans la base de données MongoDB, ainsi que de toutes opérations qui ne requièrent pas de communication multidirectionnelle ou qui n'attendent pas de retour immédiat.

En somme, Socket.IO est le protocole prioritaire pour les actions qui nécessitent la communication en temps réel et HTTP pour les actions qui attendent un retour. Les fonctionnalités du sprint 3 utiliseront cette même approche. La création d'un historique de partie nécessite la sauvegarde et la récupération de l'historique ne nécessite pas une persistance du serveur donc le protocole HTTP sera priorisé pour envoyer les informations au client de manière asynchrone. La création de QRL aussi nécessite une persistance dans la base de données MongoDB. Une fois la sauvegarde effectuée, le serveur n'a pas d'actions supplémentaires à faire, ainsi HTTP sera priorisé. Pour les parties de jeu en temps réel les deux protocoles seront utilisés. HTTP sera utilisé pour récupérer l'information sur la base de données. Cependant, pour la validation et l'envoi des réponses des joueurs par l'organisateur ainsi que la mise à jour de l'histogramme, la librairie Socket.IO sera utilisée pour sa réactivité et sa communication en temps réel. Pour la liste des joueurs interactifs, le protocole Socket.IO sera utilisé puisque l'organisateur a besoin d'être constamment alerté par le serveur de l'état dans lequel se trouve ces joueurs c'est à dire s'il a quitté la partie, s'il a envoyé ou non un résultat ainsi que les fonctionnalités de clavardages. Une latence faible et une écoute constante du serveur face à ces différents évènements est requise, ce qui fait de Socket.IO un meilleur choix.

3. Description des paquets

3.1 Paquets HTTP

[Figure 1 - Table des différentes requêtes http faites du client vers le serveur]

Méthode	**URI**	**Paramètres URI**	**Description**	**Corps de la Requête**	**Corps de la Réponse**	**Code(s) de retour**
GET	/game	—	Récupérer tous les jeux contenus dans la base de donnée	—	Succès: { *game[]* } Échec: string	Succès: 200 Échec: 400
GET	/game/:isGameVisible	isGameVisible : boolean	Récupérer tous les jeux marqués comme visibles dans la base de donnée	—	Succès: { *game[]* } Échec: string	Succès: 200 Échec: 400
PATCH	/createMatch/:id/toggleVisibility	id: string	Changer le paramètre de visibilité d'un jeu	—	Succès: { *game* } Échec: string	Succès: 200 Échec: 400/404
GET	/game/:id	id: string	Récupérer un jeu spécifique à partir de son identifiant unique	—	Succès: { *game* } Échec: string	Succès: 200 Échec: 400
DELETE	/createMatch/:id	id: string	Supprimer un jeu	—	Succès: { *game* } Échec: string	Succès: 200 Échec: 400
POST	/createMatch	—	Créer un nouveau jeu	{ lastModification: string, duration: number, questions: [{ type: string,	Succès: { game } Échec: string	Succès: 201 Échec: 400

				text: string, points: string, choices: [{ text: string, isCorrect: boolean}] }] }		
PATCH	/createMatch/:id	id: number	Modifier un jeu sur la base de données	{ lastModification: string, duration: number, questions: [{ type: string, text: string, points: string, choices: [{ text: string, isCorrect: boolean}] }] }	Succès: { — } Échec: string Conflit: string	Succès: 200/201 Échec: 400 Conflit: 409
POST	/createMatch	—	Créer un nouveau jeu à partir d'un fichier JSON	{ id: string, schema: string, lastModification: string, duration: number, questions: [{ type: string, text: string, points: string, choices: [{ text: string, isCorrect: boolean}] }] isJson: boolean }	Succès: { game } Échec: string	Succès: 201 Échec: 400

3.2 Paquets WebSockets en utilisant la librairie Socket.IO

[Figure 2 - Table des différentes communications faites par Socket.IO]

Événement	**Source**	**Description**	**Données**	**Événements Potentiellement déclenchés**
successfulJoin	Serveur	Est déclenché si l'utilisateur a entré un nom valide et un numéro de partie valide. Redirige le joueur vers la page de la salle d'attente du bon match et met à jour la liste de joueurs de tous les membres de la partie.	*matchId*	-
wrongMatchId	Serveur	Notifie le client que le code fourni est invalide.	-	-
wrongUserName	Serveur	Notifie le client que le nom du joueur fourni est invalide.	-	-
matchStarted	Serveur	Redirige les joueurs et l'organisateur vers la vue du jeu et notifie que le mode de jeu est de type multiplayer.	*matchId*	
questionTimer	Serveur	Fournit le temps restant pour la question.	*remainingTime*	-
matchAboutToStart	Serveur	Notifie le client que le match est sur le point de commencer.	-	-
backToHomePage	Serveur	Redirige l'utilisateur vers la page d'accueil	-	-
matchCanceled	Serveur	Redirige l'utilisateur vers la page d'accueil et l'avertit que la partie a été annulée.	-	-
playerBanned	Serveur	Redirige le joueur banni vers la page d'accueil et le notifie du ban.	-	-
matchLocked	Serveur	Notifie le client que la partie est verrouillée pour refuser l'accès aux nouveaux joueurs.	-	-
timeIsUp	Serveur	Selon le contexte 1. décompte de début de partie : passer à la page de la partie de jeu. 2. décompte des questions : passer interdire les	-	

		joueurs de répondre.		
matchOver	Serveur	Redirige les joueurs et l'organisateur vers la page d'accueil	-	-
messageText	Serveur	Reçoit le message avec les informations d'envoi et l'ajoute à la liste de message.	*author, text, color, time*	-
enablePanicOption	Serveur	Rend le mode panique disponible.	-	-
updatedWaitRoomPlayersList	Serveur	Met à jour la liste des joueurs affichée dans la salle d'attente.	*players*	-
displayFinalResults	Serveur	Envoie les résultats finaux au client.	*finalResults*	-
questionAnswered	Serveur	Averti qu'un joueur à répondu à la question.	*value*	-
questionOver	Serveur	Averti le client qu'il faut passer à la prochaine question.	*value*	-
playerAnswerIndex	Serveur	Fournit l'index des réponses du joueur.	*answerIndex*	-
updatedPlayersList	Serveur	Met à jour la liste des joueurs de la partie.	*res*	-
updatedPlayersPoints	Serveur	Met à jour les points des joueurs.	*points*	-
updatedPlayersBonus	Serveur	Fournit le résultat du bonus.	*bonus*	-
updatedPlayersOutList	Serveur	Met à jour la liste des joueurs ayant quitté la partie.	*res*	-
updatedPlayersOutPoints	Serveur	Met à jour la liste des points des joueurs ayant quitté la partie.	*points*	-
bonusNotification	Serveur	Notifie le client afin d'avertir le joueur qu'il a reçu le bonus.	-	-
scoreCorrection	Serveur	Corrige l'attribution de points du bonus en prenant compte du temps de réponse.	-	-
messageText	Client	Envoie le message tapé aux clients présents dans la room "matchId"	*messageText, matchId*	messageText
finalResultsForAll	Client	Récupère les noms, points et bonus de tous les joueurs.	-	displayFinalResults

panic	Client	Déclenche le mode panique	-	-
pauseTimer	Client	Met le chronomètre sur pause	-	-
resumeTimer	Client	Remet le chronomètre en route	-	-
waitRoomPageInitialized	Client	initialise et synchronise la page pour tous les joueurs	-	updateWaitRoom
matchAboutToStart	Client	Démarre le décompte du début de partie	-	-
exitWaitRoom	Client	Retire le joueur de la room ‘salle d’attente’, supprime la room si appelé par l’organisateur	-	reinitializeSocketListener, matchCanceled
changeLockState	Client	Verrouille ou déverrouille la salle d’attente d’une partie	*isLocked*	-
reinitializeSocketListeners	Client	Réinitialise les listeners du socket.	-	-
banPlayer	Client	Retire le joueur de la salle d’attente et ajoute le nom du joueur à la liste des nom banni pour cette partie	*bannedPlayer*	-
attemptJoinMatch	Client	Vérifie les informations entrées par le potentiel joueur pour qu’il rejoigne une partie	-	wrongUserName, matchLocked, wrongMatchId
testMatchStart	Client	Initialise les donnée du match nécessaire au timer puis démarre le timer	-	-
createMatch	Client	Crée un match en mode organisateur	*gameId, matchId, socket, sio*	-
exitMatch	Client	Retire le joueur de la partie, retire tout les joueurs et supprime la room si appelé par l’organisateur	-	reinitializeSocketListeners
questionAnswered	Client	Vérifie si tous les	-	questionAnswered

		joueurs ont répondu à la question si oui : déclenche l'affichage du bouton suivant et l'évènement questionDone. Sinon attend.		
questionOver	Client	Déclenche l'affichage de la question suivante et reset tout les joueurs comme n'ayant pas répondu	-	questionOver
stopTimer	Client	Arrête le chronomètre	-	-
questionTimer	Client	Initialise le décompte avec le temps choisis pour le jeu	*matchId*	
playerAnswerIndex	Client	Envoie l'index (le numéro) des réponses choisis par le joueur	*answerIndex*	playerAnswerIndex
getPlayersList	Client	Demande liste des noms des joueurs	-	updatedPlayerList
getPlayersPoints	Client	Demande la liste des points des joueurs	*matchId*	updatedPlayersPoints
scoreAttribution	Client	Permet la mise à jour des points des joueurs après qu'il ai répondu à une question	*isAnswerGood, questionNumber, isScoreCorrected*	bonusNotification updatedPlayersPoints
wrongAnswer	Client	Notifie que le joueur a sélectionné la mauvaise réponse	*answer*	scoreCorrection

3.3 Interfaces

[Figure 3 - Table des différentes interfaces présentes dans le projet]

Nom	Description	Structure
Check-box	Informations sur le statut des choix de réponse	{ choices: string, isBoxChecked: boolean, buttonColor: string }

Choice	formations sur un choix de réponse avec l'indication et si elle est bonne ou pas	{ text: string, isCorrect: boolean | string }
Game-interface	Informations sur l'interface de jeu	{ _id: string, title: string, description: string, duration: number, points:number, question: [{ type: string, text: string, points: number, choices: [{ text: string, isCorrect: boolean}] },], }
Game(client)	Informations sur un jeu	{ id: string, $schema: string, visible: boolean, title: string, description: string, duration: number, lastModification: string, question: Question[], }
Question(client)	Informations sur les questions	{ type: string, text: string, points: number, choices: Choice[], }
Game(serveur)	Informations sur un jeu	{ id?: string, $schema?: string, visible?: boolean, title?: string, description?: string, duration?: number, lastModification?: string, question?: Question[], }

Question(serveur)	Informations sur les questions	{ type?: string, text?: string, points?: number, choices?: Choice[], }
Result	Informations sur les résultats	{ name: string, points: number, bonus: number, }