Équipe 203

PolyDraw
Protocole de communication

Version 1.3

Historique des révisions

Date	Version	Description	Auteur
2020-02-05	1.0	Version initiale du plan de communication	Pascal Alexandre-Morel Cédric Tessier Martin Pouliot
2020-02-06	1.1	Correction de fautes et ajustement du numéro des figures	Cédric Tessier Pascal Alexandre-Morel
2020-02-09	1.2	Mise à jour du protocole suite à la complétion du projet.	Philippe Côté-Morneault

Table des matières

Protocole de communication

1. Introduction

Dans ce document, il sera question des différents protocoles de communication utilisés dans le projet PolyDraw. La section communication client-serveur comportera un résumé des solutions établies pour la communication bidirectionnelle des clients lourd et léger avec le serveur. Par la suite, une description détaillée des paquets sera présentée et guidera le développement des composantes réseau du projet. Chacune des sections présentées précédemment sera divisée en deux parties: les communications REST et les communications temps réels faites sur un socket.

2. Communication client-serveur

Le protocole de communication est un protocole qui est hybride. Ceci dit, deux technologies différentes sont utilisées. Le but derrière le protocole hybride est de permettre de tirer des avantages des deux technologies. La première technologie utilisée dans le protocole est un api REST.

Cette technologie permet de faire des requêtes qui sont dites sans états. Par exemple, on peut aller chercher la liste des parties, la liste des joueurs, consulter un profil, etc. On remarque que dans toutes ces requêtes, le client s'occupe d'aller chercher de l'information. Le serveur a juste besoin de répondre. Cette technologie possède un désavantage. Elle ne permet pas au serveur d'envoyer de l'information directement au client. Pour ce faire, il faut combiner une autre technologie. De plus, elle ne permet pas de garder efficacement une session, il faut passer un jeton à chaque requête. L'api REST est hébergé par défaut sur le port 5000 et est offert sur le protocole HTTP. Il est possible de rajouter un certificat TLS à l'aide d'un "Reverse Proxy" tel que Nginx ou Apache.

Afin de permettre au serveur d'envoyer directement de l'information au client, il faut utiliser une autre technologie. Il a été retenu d'utiliser les sockets TCP directement. Ceci permet de garder une session ouverte et d'envoyer des messages du serveur au client. Le protocole TCP est fait pour que le récepteur envoie un accusé de réception à chaque fois qu'une donnée est envoyée. Ceci permet de s'assurer que les données sont bien transmises. Ceci implique que tous les délais doivent être multipliés par deux. Cependant, des applications existantes permettent de vérifier que ce double délai ne cause pas de limitations trop importantes. Par exemple, le site skribbl.io utilise les sockets TCP pour fournir une application quasi instantanée, sans temps de latence. Ce protocole a donc été retenu en raison de l'intégrité des données et de la facilité de l'utilisation. Le port 5001 est utilisé pour ce protocole. La communication entre les différents sockets n'est pas cryptée.

3. Description des paquets (REST)

Comme mentionné plus haut, l'application utilise le protocole REST pour certaines communications entre le client et le serveur. Ce protocole est privilégié pour les communications ne nécessitant pas une réponse en temps réel du serveur. De même, ce protocole est utilisé pour les communications ne nécessitant pas une connexion continue entre le client et le serveur. En effet, les requêtes HTTP sont sans états ("stateless") et la connexion établie entre le client et le serveur est fermée une fois la réponse reçue. C'est alors à l'aide du protocole HTTP que des actions ne nécessitant pas une connexion continue, comme aller chercher les statistiques d'un utilisateur et s'authentifier, seront effectuées.

Dans les sections qui suivront, les routes et données transmises seront décrites. Pour chaque requête et réponse, le type de média (*Media type*) utilisé sera application/json. De plus, les requêtes du client contiendront une clé *Language* en en-tête. Cette clé permettra de spécifier la langue utilisée par le client. En fonction de la langue, le serveur pourra répondre au client en français ou en anglais.

Pour éviter la répétition, on considérera que chaque requête qui sera présentée peut retourner un code d'erreur 400 (*Bad request*) si la langue en en-tête ne correspond pas aux deux options disponibles. Ces options sont FR pour la langue française et EN pour la langue anglaise. De plus, le temps est toujours donné en millisecondes, à l'exception des dates Unix Timestamp qui sont données en secondes.

3.1 Authentification

Les requêtes en lien avec l'authentification utilisent une route qui commence par */auth*.

Route /auth (POST)

D'abord, quand l'utilisateur tente de se connecter en envoyant son nom d'utilisateur et son mot de passe, les données suivantes sont transmises:

Tableau 1.1: Présentation des informations transmises

HTTP 200	
Route	/auth
Méthode	POST
En-tête	- Language: FR ou EN
Contenu de la requête (JSON)	- Username: string - Password: string
Contenu de la réponse (JSON)	- SessionToken: string - Bearer: string - UserID: string

Le tableau ci-dessus est un exemple de requête ayant un code HTTP de 200. Ici, tout s'est bien passé. L'utilisateur a entré un nom d'utilisateur et un mot de passe valide et présents dans la base de données. Comme réponse, le serveur envoie au client un jeton de session (*SessionToken*) et un jeton bearer. Le jeton de session permet au client de s'identifier aux requêtes subséquentes qu'il pourra envoyer. Il est aussi passé au service de socket. C'est à partir de ce jeton que le socket s'identifie au serveur. Ce jeton est valide jusqu'à ce que le test de santé (*health check*) détecte que la connexion est fermée. Après ce délai, le jeton expirera et il faudra déconnecter la session. Le jeton bearer est, quant à lui, un jeton envoyé au client dans le cas où il désire qu'on se souvienne de lui (*remember me?*). Lors de lancements subséquents de l'application, le nom d'utilisateur et le mot de passe seront déjà rentrés, facilitant ainsi la connexion.

Si l'utilisateur entre un nom d'utilisateur et un mot de passe erroné, les données du tableau suivant seront transmises.

Tableau 1.2: Présentation des informations transmises

HTTP 401	
Route	/auth
Méthode	POST
En-tête	- Language: FR ou EN
Contenu de la requête (JSON)	- Username: string - Password: string
Contenu de la réponse (JSON)	- Error: string

Une réponse avec un code 401 (accès non autorisé) est retournée. Le contenu de la réponse comporte un texte d'erreur que le client doit traiter. Il peut, par exemple, afficher un message expliquant l'erreur qui s'est produite. Les codes

d'erreur possibles pour la route */auth* sont présentés dans le tableau ci-dessous.

Tableau 1.3: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (*Bad request*)	La syntaxe de la requête est mauvaise (p. ex.: le champ password est manquant).
401 (*Unauthorized*)	L'utilisateur ne peut pas accéder au compte puisqu'il a rentré des informations erronées (p. ex.: mauvais mot de passe entré).
409 (*Conflict*)	L'utilisateur ne peut pas accéder au compte puisqu'il est déjà en utilisation (p. ex.: un compte déjà ouvert sur l'application ordinateur ne peut pas être ouvert en même temps sur l'application mobile).

Route /auth/bearer (POST)

Cette route est utilisée si l'utilisateur a précédemment décidé qu'on se souvient de lui (*remember me?*). Lorsqu'on tente de se connecter, on passe par cette route au lieu de */auth*. On envoie alors le jeton bearer au serveur. Il se charge alors de confirmer le jeton et d'ouvrir une session en retournant un jeton de session à l'utilisateur.

Tableau 2.1: Présentation des informations transmises

HTTP 200	
Route	/auth/bearer
Méthode	POST
En-tête	- Language: FR ou EN
Contenu de la requête (JSON)	- Bearer: string
Contenu de la réponse (JSON)	- SessionToken: string - Bearer: string (optionnel) - UserID: string

Pour cette route, les mêmes cas d'erreur présentés dans le tableau 1.3 sont possibles.

Route /auth/register (POST)

Comme le nom l'indique, cette route permet à l'utilisateur de se créer un compte. Le client envoie alors son nom d'utilisateur, son mot de passe, son adresse courriel, son prénom et son nom. Si les champs à remplir sont valides, l'utilisateur est connecté à l'application. C'est pourquoi il reçoit les jetons déjà mentionnés.

Tableau 3.1: Présentation des informations transmises

HTTP 200	
Route	/auth/register
Méthode	POST
En-tête	- Language: FR ou EN
Contenu de la requête (JSON)	- Username: string - Password: string - Email: string - FirstName: string - LastName: string - PictureID: int
Contenu de la réponse (JSON)	- SessionToken: string - Bearer: string (optionnel)

Les codes d'erreur pouvant être retournés par le serveur sont les suivants:

Tableau 3.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (*Bad request*)	La syntaxe de la requête est mauvaise (p. ex.: le champ password est manquant).
409 (*Conflict*)	L'utilisateur ne peut pas créer le compte puisque le nom d'utilisateur est déjà pris.

3.2 Clavardage

Pour les routes qui suivent, le jeton de session (*SessionToken*) doit être passé en en-tête afin d'identifier la session avec le serveur. Ceci veut dire que l'utilisateur doit être authentifié avant de pouvoir accéder aux différentes ressources disponibles.

Pour les requêtes en lien avec le clavardage, les routes commencent par */chat*. Ces requêtes permettent, entre autres, d'aller chercher les messages d'un canal de communication, d'aller chercher les différents canaux de communication disponibles, ou encore les informations d'un canal de communication.

Route /chat/messages/:channelid:/?start=0&end=100 (GET)

Cette route permet d'aller chercher les messages d'un canal spécifié par *channelID*. Puisqu'on ne veut pas aller chercher tous les messages d'un coup, *start* et *end* auront des valeurs de 0 et 100 respectivement. Ici, 0 signifie le dernier message envoyé sur le canal alors que 100 représente le centième message envoyé à partir du dernier message envoyé. De cette manière, le client peut chercher les 100 derniers messages du canal au lieu d'aller chercher l'intégralité des messages. La performance se verra alors améliorée. Si jamais l'utilisateur déroule les 100 messages et qu'il veut visionner des messages plus anciens, la même requête sera appelée avec une valeur de *start* et de *end* de 100 et de 200. Pour connaître la valeur de *end* pour des requêtes subséquentes à la même route, le serveur renvoie aussi le nombre de messages total dans le canal de discussion. Le client est alors capable de calculer combien de messages aller chercher.

Tableau 4.1: Présentation des informations transmises

HTTP 200	
Route	/chat/messages/:channelid:/?start=0&end=100
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- Messages: Message[] - MessagesTotal: int

Il est à noter qu'un objet de type *Message* est un objet comportant les propriétés suivantes: *ChannelID* (string), *Timestamp* (int), *UserID* (string), *Username* (string), *Message*(string). Les erreurs possibles pour cette route sont les suivantes.

Tableau 4.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (*Bad request*)	La syntaxe de la requête est mauvaise (p. ex.: *start* ou *end* ne sont pas des valeurs valides).
401 (*Unauthorized*)	Le jeton de session est invalide.
404 (*Not found*)	L'identifiant du canal de communication n'existe pas. Il n'existe pas de données de statistique dans l'intervalle *start* et *end* spécifié.

Route /chat/channels (GET)

Cette route retourne l'information sur les différents canaux de communication disponibles. Par information sur les canaux, on parle de leurs noms, de leurs identifiants et des utilisateurs abonnés au canal de communication.

Tableau 5.1: Présentation des informations transmises

HTTP 200	
Route	/chat/channels
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- Channels: Channel[]

Un objet de type *Channel* comporte les propriétés suivantes: *ID* (string), *Name* (string), *Users* (ChannelUser[]), IsGame(boolean). Ici, ChannelUser contient les IDs et nom d'utilisateurs des utilisateurs qui se sont joints au canal. Pour cette route, uniquement une réponse de code 401 peut survenir si le jeton de session est expiré.

Route /chat/channels/:channelID: (GET)

Les requêtes à cette route agissent comme les requêtes aux routes */chat/channels*. Par contre, le serveur renvoie uniquement l'information par rapport au canal spécifié par l'identifiant *channelID*.

Tableau 6.1: Présentation des informations transmises

HTTP 200	
Route	/chat/channels/:channelID:
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- Channel: Channel

Tableau 6.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
401 (*Unauthorized*)	Le jeton de la session est invalide.

404 (*Not found*)	L'identifiant channelID est invalide.

3.3 Utilisateurs

Route /users (PUT)

À cette route, une requête utilisant la méthode *PUT* peut être appelée. Dans le cas échéant, l'utilisateur peut modifier ses propres informations.

Tableau 7.1: Présentation des informations transmises

HTTP 200	
Route	/users
Méthode	PUT
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la requête (JSON)	- ModifiedUser: ModifiedUser

Un *ModifiedUser* contient les champs modifiables d'un utilisateur, soit *Username* (string), *FirstName* (string), *LastName* (string), *Password* (string) et *Email* (string), *PictureID* (string).

Tableau 7.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (*Bad request*)	La requête ne contient pas d'objet *ModifiedUser*.
401 (*Unauthorized*)	Le jeton de la session est invalide.
404 (*Not found*)	L'identifiant *UserID* lié au jeton de session est invalide .
409 (*Conflict*)	Le nom d'utilisateur modifié est déjà pris par un autre utilisateur.

3. 4 Statistiques

Route /stats (GET)

À partir de cette route, un utilisateur peut récupérer ses statistiques. Les statistiques récupérables sont les suivantes: *GamesPlayed* (int), *WinRatio* (double), *AvgGameDuration* (int), *TimePlayed* (int), *ConnectionHistory* (Connection[]), *GamesPlayedHistory*, (GamePlayed[]). Les objets *Connection* et *GamesPlayed* sont des structures comportant respectivement les dates et heures de connexion/déconnexion et les historiques des parties jouées (nom du gagnant, date, heure et résultat des parties). De même, la route retournera les différents trophées (A*chievements*) remportés par l'utilisateur. Il est composé du nom du trophée, la description ainsi que la date d'obtention.

Tableau 8.1: Présentation des informations transmises

HTTP 200	
Route	/stats/
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- GamesPlayed: int - WinRatio: double - AvgGameDuration: int - TimePlayed: int

Tableau 8.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (*Bad request*)	La langue est invalide.
401 (*Unauthorized*)	Le jeton de la session est invalide.
404 (*Not found*)	L'identifiant *UserID* n'existe pas.

Route /stats//history/?start=0&end=100 (GET)

Cette route agit de la même manière que la route mentionnée plus haut. Par contre, elle permet de spécifier un intervalle

pour les connexions et les parties jouées retournées. Par exemple, un intervalle avec start=0 et end=100 retourne les 100 dernières connexions et déconnexions enregistrées et les 100 dernières parties jouées.

Tableau 9.1: Présentation des informations transmises

HTTP 200	
Route	/stats/history/?start=0&end=100
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- ConnectionHistory: Connection[] - GamesPlayedHistory: GamePlayed[] - Achievements: Achievements[]

Tableau 9.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (*Bad request*)	Les valeurs pour *start* et *end* sont invalides.
401 (*Unauthorized*)	Le jeton de la session est invalide.
404 (*Not found*)	L'identifiant *UserID* n'existe pas. Il n'existe pas de données de statistique dans l'intervalle *start* et *end* spécifié.

3.7 Création d'un jeu

Route /games (POST)

Pour cette route, l'utilisateur entre les indices et le mot à deviner. Puisqu'une image représente des données binaires et non du JSON, elle sera envoyée par une autre route, soit la route /games/:gameid:/image (POST).

Tableau 17.1: Présentation des informations transmises

HTTP 200	
Route	/games
Méthode	POST
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la requête (JSON)	- Hints: string[] - Word: string - Difficulty: int
Contenu de la réponse (JSON)	- GameID: string

Le serveur retourne l'identifiant du jeu généré pour que le client puisse ensuite envoyer l'image.

Tableau 17.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (Bad request)	L'utilisateur entre un champ invalide.
401 (*Unauthorized*)	Le jeton de la session est invalide.

Route /games/:gameid:/image (POST)

Cette route permet d'envoyer l'image associée à un jeu. Il est important de prendre en compte que pour les requêtes de cette route, le champ *Media Type* ne correspond pas à application/json. En effet, des données binaires sont envoyées.

Tableau 18.1: Présentation des informations transmises à la route

HTTP 200	
Route	/games/:gameid:/image
Méthode	POST
En-tête	- Language: FR ou EN - SessionToken: string
Media Type	- image/svg+xml - image/png - image/jpg
Contenu de la requête	- file: byte[] - mode: int

Les octets de l'image seront envoyés au serveur. Ici, il peut y avoir 3 types de médias, dépendamment du format de l'image. Dans le cas où l'image n'est pas un SVG, potrace est utilisé pour la convertir. Le *mode* est la façon de tracer l'image. Dans le cas d'un SVG, cette information est déjà incluse à l'intérieur de l'image.

Tableau 18.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (Bad request)	Les octets sont invalides ou le format svg ne contient pas les commentaires nécessaires.
401 (*Unauthorized*)	Le jeton de la session est invalide.

Route /games/:gameid: (DELETE)

Cette route permet d'effacer un jeu. Elle est utile dans le cas où l'utilisateur désire annuler le dessin. Ceci permet de ne pas laisser des dessins incomplets dans la base de données.

Tableau 18.1: Présentation des informations transmises à la route

HTTP 200	
Route	/games/:gameid:
Méthode	DELETE
En-tête	- Language: FR ou EN - SessionToken: string

Route /games/:gameid: (GET)

Cette route permet de récupérer les informations d'une partie, autre que l'image. Les informations pouvant être récupérées sont les indices, le mot à deviner et la difficulté.

Tableau 20.1: Présentation des informations transmises à la route

HTTP 200	
Route	/games/:gameid:
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la requête (JSON)	- Hints: string[] - Word: string - Difficulty: int

Tableau 20.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (Bad request)	Le champ d'une des requêtes est invalide.
401 (*Unauthorized*)	Le jeton de la session est invalide.
404 (*Not found*)	L'identifiant de la partie est invalide.

3.8 Création de groupes

Route /groups (GET)

Cette route permet d'aller chercher les informations de tous les groupes existants. Les informations sont retournées par un tableau de type *Group*. Chaque *Group* a comme propriétés *GroupID* (string), *GroupName* (string), *GameType* (int), *PlayersInGroup* (int), *PlayersMax* (int) et *VirtualPlayers* (int).

Tableau 21.1: Présentation des informations transmises

HTTP 200	
Route	/groups
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- Groups: Group[]

Tableau 21.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
401 (*Unauthorized*)	Le jeton de la session est invalide.

Route /groups (POST)

Cette route permet de créer un groupe de joueur (lobby). Le client passe au serveur le type du jeu, le nom du groupe et le nombre de joueurs maximal pouvant joindre le groupe. Il doit également envoyer la difficulté du jeu pour le mode de jeu sprint solo et le mode de jeu sprint coopératif. Si le client ne veut pas fournir de nom de groupe, le serveur générera lui-même un nom.

Tableau 22.1: Présentation des informations transmises

HTTP 200	
Route	/groups
Méthode	POST
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la requête (JSON)	- PlayersMax: int - VirtualPlayers: int - GameType: int - Difficulty: int (optionnel)
Contenu de la réponse (JSON)	- GroupName: string - GroupID: string

Tableau 22.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
400 (Bad request)	Les paramètres de la requête sont invalides. Par exemple, un *GameType* qui requiert une difficulté n'a pas d'attribut *Difficulty* dans sa requête.
401 (*Unauthorized*)	Le jeton de la session est invalide.

Route /groups/:groupid: (GET)

Cette route permet d'aller chercher les informations d'un groupe spécifique. Les informations seront retournées dans un objet de type *Group*. Comme mentionné plus haut, chaque *Group* a comme propriétés *GroupID* (string), *GroupName* (string), *GameType* (string), *PlayersInGroup* (int), *PlayersMax* (int), *VirtualPlayers* (int) et *Players* (Players[]). La propriété players a été rajouté pour lister les utilisateurs qui sont présentement dans le groupe. Cette liste contient également les joueurs virtuels.

Tableau 23.1: Présentation des informations transmises

HTTP 200	
Route	/groups/:groupid:
Méthode	GET
En-tête	- Language: FR ou EN - SessionToken: string
Contenu de la réponse (JSON)	- group: Group

Tableau 23.2: Présentation des codes d'erreur possibles

Code d'erreur	Description
401 (*Unauthorized*)	Le jeton de la session est invalide.
404 (Not found)	L'identifiant de groupe n'est pas valide.

4. Communication en temps réel (socket)

Cette section du protocole est dite temps réel. Ce protocole n'est pas sans état et permet au serveur de communiquer directement avec le client. Il est nécessaire d'avoir cette partie pour avoir un clavardage et un canevas en temps réel. Pour ce faire, on utilise le protocole TCP. Ce protocole est assez rapide, même si les deux parties doivent envoyer un ACK (Acknowledge) pour vérifier que la réception du message soit bien conforme. Afin d'offrir une bonne expérience à l'utilisateur, le délai d'expiration de la session est de 10 secondes. Le port utilisé pour ce protocole est le 5001 par défaut sur le serveur.

Afin de diminuer la sérialisation au minimum, le protocole a été conçu pour utiliser les octets directement. Ceci assure une excellente performance. Dans le cas, où les messages n'ont pas besoin d'avoir un traitement court, on utilise de la sérialisation. La sérialisation utilisée est semblable à JSON. Cependant, il s'agit de MessagePack. Ce protocole est plus rapide que JSON pour la sérialisation et occupe moins d'espace. Il n'est pas censé être lisible contrairement à JSON. Dans la description des messages, on retrouvera la notation "{}" pour indiquer qu'il s'agit de sérialisation. Sinon les octets seront séparés dans des tableaux.

La réponse de certains messages n'est pas garantie d'être séquentielle. Il est donc possible qu'un autre message arrive avant la réponse pour une requête. Il est donc recommandé que l'application serveur ou cliente utilise une couche applicative pour faire le routage des messages vers la bonne partie du code. De cette façon si d'autres messages ne sont pas une réponse, ils vont être routés dans leur partie respective.

4.1 Authentification

Afin de valider l'utilisateur et satisfaire les exigences de sécurité, chaque client doit s'authentifier sur le serveur. On s'authentifie avec un jeton. Ce jeton peut être obtenu à partir de la partie REST du serveur (voir la section 4.3 message de type 0). Le message d'authentification est défini plus bas.

4.2 Format de paquet

Afin de faciliter l'encodage et le décodage des divers messages, un format de paquet a été retenu. Le format TLV a été retenu pour la forme des paquets. TLV est une abréviation pour "type", "length" et "value". Le champ "type" représente le type du message, le champ "length" représente la longueur de l'attribut "value"et le champ "value" représente le contenu du message. La valeur (contenu) du message peut contenir ce qu'on veut. Il peut s'agir de texte, de données brutes ou encore de fichiers. La longueur et la position des trois champs sont définies dans le tableau ci-dessous.

Format de paquet

Type (1 octet)	Longueur (2 octets)	Valeurs (65535 octets)

Il est à noter que si Type est:

- pair, alors le message est de direction: client vers serveur
- impair, alors le message est de direction: serveur vers client

On remarque qu'il y a quelques limitations dans le protocole. On ne peut pas avoir plus que 256 types de messages différents. Ceci devrait être amplement suffisant puisque le logiciel a des exigences définies et il est possible d'énumérer tous les types de messages. Pour la longueur de la section valeurs, il est impossible d'envoyer plus de 65535 octets. Ceci est dû au fait que la longueur ne peut pas contenir un nombre plus grand dans deux octets.

Une convention a été retenue pour la numérotation du type. Le nombre décimal du type indique la direction du message. Cette implémentation n'est pas retenue dans le code, mais elle a pour but de simplifier le débogage. Lorsque le nombre est pair, le message suit la communication client vers serveur. Lorsque le nombre est impair le message suit la direction serveur vers client.

De plus, lorsque des nombres sont encodés dans les messages, ceux-ci doivent utiliser l'encodage gros-boutiste (Big endian). Il est important de respecter cet encodage afin d'obtenir les bonnes valeurs.

4.3 Messages de gestion

Les messages suivants sont les paquets que le client ou le serveur peuvent envoyer pour la section gestion. Ces messages sont utiles pour faire la gestion de la session. Ils devraient être annoncés dans toutes les parties de l'application, car il s'agit d'informations utiles et nécessaires au bon fonctionnement de l'application. Par exemple, si le serveur se déconnecte, il faut que le client gère cela de son côté.

Connexion au socket

Type: 0 | 0x00	Valeur: octets du jeton (encodage utf-8)

Ce message est utilisé pour authentifier la session. Il faut donc passer un jeton au socket pour vérifier que l'utilisateur soit bien authentifié. On obtient le jeton à partir de la route */auth* dans la section REST. Ce message doit être envoyé une seule fois par session.

Réponse du serveur

Type: 1 | 0x01	Valeur: 1 octet de réponse

Ce message est utilisé pour répondre à la demande d'authentification du client. Dans le cas où l'authentification est valide, l'octet 0x01 est envoyé en réponse au client. Dans le cas contraire, l'octet 0x00 est envoyé. Si le client n'a pas envoyé de message d'authentification au préalable, une réponse de connexion refusée sera envoyée pour tout autre type de message. Les valeurs de réponses sont les suivantes.

- 0x01: Connexion acceptée
- 0x00: Connexion refusée

Test santé serveur

Type: 9 | 0x09	Valeur: vide

Ce message est utilisé par le serveur pour détecter si le client existe toujours. Le serveur envoie ce message toutes les 5 secondes. Si le client ne reçoit pas ce message à l'intérieur de 10s la connexion est considérée comme coupée par le client. De plus, le client doit répondre au message à l'aide du message 10.

Test santé client

Type: 10 | 0x0A	Valeur: vide

Ce message est utilisé par le client pour répondre au serveur suite au message 9. Si le serveur ne reçoit pas la réponse dans les 10 secondes qui suivent, la connexion est considérée comme coupée et le serveur va fermer la connexion.

Erreur du serveur

Type: 255 | 0xFF	Valeur: { ErrorCode: int, Type: int, Message: str}

Ce message est utilisé par le serveur pour aviser le client qu'une erreur s'est produite. Ceci peut être par exemple lorsqu'une action ou le message envoyé par le client n'était pas attendu.

4.4 Messages pour clavardage

Les messages suivants sont les paquets que le client ou le serveur peuvent envoyer pour la section clavardage. Ces messages peuvent donc être routés dans les parties respectives de l'application.

Envoi de messages

Type : 20 | 0x14	Valeur: { Message:str, ChannelID:str}

Ce message est pour envoyer un message à un canal. Ce message est envoyé du client vers le serveur. Le serveur s'occupe d'acheminer le message aux clients connectés. Le message est envoyé au client par la suite sous la forme du message de type 21.

Réception de message

Type : 21 | 0x15	Valeur:{ ChannelID:str, Timestamp:int, SenderID:str, SenderName:str}

Ce message est utilisé par le client pour recevoir des messages dans les canaux auxquels il est abonné. Le timestamp est l'heure de réception du message par le serveur. Il s'agit d'un timestamp Unix.

Rejoindre un canal

Type: 22 | 0x16	Valeur: ChannelID UUID 16 octets

Ce message est utilisé par le client pour aviser le serveur qu'il désire s'inscrire à un nouveau canal. Il n'a pas de sérialisation, on doit juste mettre les octets du channelID.

Notification d'arrivée dans le canal

Type: 23 | 0x17	Valeur:{UserID:str, Username:str, ChannelID:str, Timestamp:int}

Ce message est envoyé au client pour l'aviser qu'un nouvel utilisateur a rejoint le canal. Ce message est également envoyé au client à titre de confirmation lorsqu'il envoie le message 22 au serveur.

Quitter un canal

Type: 24 | 0x18	Valeur: ChannelID UUID 16 octets

Ce message est envoyé au serveur pour aviser que le client veut quitter un canal. Le serveur répond avec le message de type 25 pour aviser le client que le serveur a bien reçu la réponse.

Notification de départ du canal

Type: 25 | 0x19	Valeur: {UserID:str, Username:str, ChannelID:str, Timestamp:int}

Ce message est envoyé du serveur pour aviser le client qu'un utilisateur s'est déconnecté du canal.

Créer un canal

Type: 26 | 0x1A	Valeur:{ChannelName:str}

Ce message est envoyé par le client pour créer un nouveau canal. Le serveur répond à la création avec le message de type 27.

Notification création d'un canal

Type: 27 | 0x1B	Valeur: {ChannelName:str, ChannelID:str, Username:str, UserID:str, Timestamp:int, IsGame: boolean}

Ce message est envoyé par le serveur pour aviser qu'un nouveau canal a été créé. IsGame indique si le canal est le canal de partie.

Détruire un canal

Type: 28 | 0x1C	Valeur: ChannelID UUID 16 octets

Ce message est envoyé par le client pour demander au serveur de détruire un canal de communication. L'ID du canal en question en envoyé en Big Endian.

Notification d'un canal détruit

Type: 29 | 0x1D	Valeur: ChannelID UUID 16 octets

Ce message est envoyé par le serveur pour aviser qu'un canal de communication a été détruit.

4.5 Messages pour dessin

Ces messages sont utilisés pour communiquer les traits aux clients. Ils sont habituellement accompagnés de messages de parties qui sont utilisés durant une partie. Les messages de dessins peuvent également être utilisés dans la fenêtre de création de jeu.

Morceau de trait client

couleur (1 octet) efface MSB, 2e MSB brushType	UUID du trait (16 octets) unique à chaque trait RFC4122	UUID utilisateur (16 octets) RFC4122 nul	taille brush (1 octet)
Point x (2 octets)	Point y (2 octets)...	Point x (2 octets)	Point y (2 octets)

Ce message est utilisé par le client pour aviser le serveur du trait qu'il est en train de dessiner. Les traits sont échantillonnés tous les 20ms et sont envoyés dans ce message au serveur. Ceci permet donc d'avoir un taux de rafraîchissement plutôt élevé. Le nombre maximum de points par message est de 16370. La position d'un point doit être faite entre 0 et 65535. Les points doivent toujours être envoyés dans le même référentiel. Ce référentiel correspond à un canevas d'une taille de 1125x750. Sur le client lourd, l'application devra se charger de modifier l'échelle de rendu en mode fenêtré. Ceci assure que le dessin garde sa forme et ce peu importe la taille de la fenêtre du client lourd. Le UUID utilisateur est nul, car le serveur est responsable de l'assigner. Toute valeur dans ce champ sera ignorée par le serveur. Le but est d'avoir un message commun au client et au serveur afin de réutiliser la structure.

Le premier octet est dédié pour la couleur, et le type d'outil. La définition de cet octet est la suivante.

Tableau 24.1: Significations des bits de l'octet de couleur

Bit	**Définition**		**Bit**	**Définition**
0	Couleur LSB		4	Non utilisé
1	Couleur		5	Non utilisé
2	Couleur		6	Type pointe | 0: cercle, 1: carré
3	Couleur MSB		7 MSB	Outil | 0: pinceau, 1: efface

Tableau 24.2: Valeurs des couleurs de l'octet couleur

Valeur	**Couleur**	**HTML**		**Valeur**	**Couleur**	**HTML**
0x0	Noir	#000000		0x4	Bleu	#0000FF
0x1	Blanc	#FFFFFF		0x5	Jaune	#FFFF00
0x2	Rouge	#FF0000		0x6	Cyan	#00FFFF
0x3	Vert	#00FF00		0x7	Magenta	#FF00FF

Il est théoriquement possible d'avoir 16 couleurs cependant, il semble avoir assez de couleur pour le jeu. Dans le cas contraire, le tableau sera mis à jour pour supporter les nouvelles couleurs.

Morceau de trait serveur

Type: 31 | 0x1F	couleur (1 octet) efface MSB, 2e MSB brushType	UUID du trait (16 octets) unique à chaque trait RFC4122

....

Ce message est le même que celui envoyé par le client voir type 30. Le serveur fait le routage des messages envoyés par le client. La seule différence est que ce message contient un UUID qui n'est pas nul lorsque les traits sont dessinés dans une partie. Lorsqu'il s'agit d'une prévisualisation, le UUID est nul également. Le client doit dessiner ce morceau de trait en moins de 20ms pour s'assurer que l'échantillonnage se fait bien.

Début dessin client

Type: 32 | 0x20	Valeur: DrawingID UUID 16 octets

Ce message informe le serveur que des messages de traits de dessin vont commencer à arriver. Ce message est utilisé seulement dans une partie. Le *drawingID* est un ID qui est envoyé dans un autre message que le client doit renvoyer au serveur.

Début dessin serveur

Type: 33 | 0x21	Valeur: DrawingID UUID 16 octets

Ce message informe le client que des messages de traits de dessin vont commencer à arriver. Le client peut utiliser ce message pour mettre le canevas dans un état de réception de message. Le DrawingID représente un ID unique du dessin. Dans le cas d'une image dessinée par un autre utilisateur et non un joueur virtuel cet ID a été généré et est éphémère. Il est utilisé seulement pour faire un lien entre les paquets de début et fin.

Fin dessin client

Type: 34 | 0x22	Valeur: DrawingID UUID 16 octets

Ce message informe le serveur que tous les traits ont été envoyés. Le *drawingID* est le même que celui du type 32.

Fin dessin serveur

Type: 35 | 0x23	Valeur: DrawingID UUID 16 octets

Ce message informe le client que tous les traits ont été envoyés. Il est surtout utilisé pour signifier la fin de la prévisualisation du dessin au client. Il est aussi envoyé à la fin lorsqu'un dessin est terminé ou un mot est deviné dans la partie. Le client peut donc utiliser ce message pour passer dans un autre état.

Prévisualisation dessin

Type: 36 | 0x24	Valeur: DrawingID UUID 16 octets

Ce message est utilisé par le client pour demander que le serveur dessine un dessin. Ceci est utilisé lors de la conception de jeu pour s'assurer que le jeu soit bien ce que l'utilisateur désire. Il s'ensuit d'une réponse par le serveur confirmant que le message a été reçu et que le jeu existe bien (type 37). Par la suite, des messages de traits de dessins sont envoyés par le serveur (type 31).

Réponse prévisualisation dessin

Type: 37 | 0x25	Valeur: 1 octet réponse

Ce message est envoyé par le serveur pour répondre au message de type 36. Dans le cas où la demande est refusée, il s'agit probablement que l'identifiant fourni au message 36 n'existe pas. Si la demande est acceptée, un message de type début de dessin (33) va être envoyé par le serveur ensuite. Les valeurs du champ sont les suivantes.

- 0x01: Demande acceptée
- 0x00: Demande refusée

4.6 Messages salle d'attente

Ces messages sont utilisés dans la salle d'attente pour signifier au joueur l'état changeant de la salle d'attente. On retrouve donc des messages pour commencer la partie et lorsqu'un joueur se connecte dans un groupe.

Demande de rejoindre un groupe

Type: 40 | 0x28	Valeur: GroupID UUID 16 octets

Ce message est envoyé par le client pour demander de rejoindre un groupe. Le serveur répond avec le message 41 pour mentionner si la demande est acceptée ou refusée. On obtient le *GroupID* à partir d'une requête REST à l'api.

Réponse de rejoindre un groupe

Type: 41 | 0x29	Valeur: {Response:bool, Error:str}

Ce message est envoyé par le serveur au client pour l'informer si l'utilisateur est accepté dans la salle d'attente. Dans le cas contraire, un message d'erreur est envoyé sinon celui-ci est nul. Dans le cas d'une réponse positive, le message 43 suit indiquant qu'un nouvel utilisateur est ajouté dans le groupe de la salle d'attente pour aviser les autres utilisateurs du groupe. De plus, le message 47 est envoyé à l'utilisateur qui a rejoint le groupe pour lui donner quelques informations sur le groupe.

Nouvelle personne rejoint un groupe

Type: 43 | 0x2B	Valeur: {UserID:str, Username:str, GroupID:str, IsCPU: boolean}

Ce message est envoyé par le serveur pour indiquer qu'une nouvelle personne a rejoint le groupe. Ceci permet donc aux clients de mettre à jour leur statut sur le nombre de joueurs restants pour participer à la partie. IsCPU indique si la personne est un joueur virtuel.

Demande de quitter un groupe

Type: 44 | 0x2C	Valeur: vide

Ce message indique au serveur que le client quitte le groupe. Le serveur va ensuite aviser les clients restants dans le groupe que la personne a quitté à l'aide du message 45. Il ne va pas envoyer de message à l'utilisateur qui vient de quitter la partie.

Personne quitte un groupe

Type: 45 | 0x2D	Valeur: {UserID:str, username:str, GroupID:str}

Ce message est envoyé par le serveur aux clients restants dans le groupe pour les aviser qu'une personne est partie.

Début de partie

Type: 48 | 0x30	Valeur: vide

Ce message est envoyé par le client pour aviser le serveur que la partie peut commencer. Seulement le créateur du groupe peut faire cette commande. Le message 49 est envoyé pour confirmer ou refuser la requête.

Réponse début de partie.

Type: 49 | 0x31	Valeur: {Response:bool, Error:str}

Ce message est envoyé par le serveur en réponse au message 48. Si le message contient la réponse vraie, le message de début de partie, va suivre (51). Dans le cas contraire, le champ *Error* possédera une erreur textuelle pour afficher à l'utilisateur. Il s'agit principalement de critères non respectés pour la création de parties.

Notification nouveau groupe créé

Type: 51 | 0x33	Valeur: {ID: str, Name: str, OwnerName: str, OwnerID, str, PlayersMax: int, Mode: int, Players: player, Difficulty: int, Language: int}

Ce message est envoyé par le serveur aux clients pour leur aviser qu'un nouveau groupe a été créé. Pour Language, la valeur est 0 pour un groupe en anglais et 1 pour un groupe en français. Pour Mode, la valeur est 0 pour Mêlée générale, 1 pour sprint solo et 2 pour sprint coopératif.

Notification groupe effacé

Type: 53 | 0x35	Valeur: GroupID UUID 16 octets

Ce message est envoyé par le serveur aux clients pour leur aviser qu'un groupe a été effacé. La valeur du message est l'ID du groupe effacé en Big Endian.

Renvoyer un joueur du groupe

Type: 54 | 0x36	Valeur: PlayerID UUID 16 octets

Ce message est envoyé par un client au serveur pour lui aviser qu'on désire renvoyer un client d'un groupe. La valeur est l'ID du joueur qu'on souhaite renvoyer.

Demande de rajout d'un joueur virtuel

Type: 56 | 0x38	Valeur:{NbJoueurs: byte}

Ce message est envoyé par au serveur par l'hôte d'un groupe pour demander l'ajout de joueurs virtuels. NbJoueurs représente le nombre de joueurs virtuels à rajouter.

4.7 Messages de gestion partie

Les messages de gestion de partie sont utilisés pour gérer la partie en cours. Ils représentent donc des événements qui se passent durant la partie. Ils aident à garder le flot dans la partie pour le client et le serveur. L'étape de la salle d'attente doit être franchie avant de recevoir ou d'envoyer des messages de parties. Tout message qui ne respecte pas cette séquence résultera d'une connexion fermée sur le socket.

Partie sur le point de commencer

Type: 61 | 0x3D	Valeur: {Players:[UserID:str, Username:str, IsCPU:bool], GameType:int, TimeImage:int, Laps:int, TotalTime:int}

Ce message est envoyé à tous les clients pour qu'ils préparent l'environnement de jeu de leur côté. Une fois que l'environnement est prêt de leur côté, ils doivent envoyer un message (52) pour dire qu'ils sont prêts à débuter la partie. La valeur *GameType* représente le type de partie, *TimeImage* représente le temps que le joueur a pour trouver une image. Les deux derniers champs sont utilisés dépendamment du type de la partie. Ils sont nuls s'ils ne sont pas nécessaires. Le champ *laps* est utilisé pour indiquer le nombre de tours que les joueurs font avant que la partie se termine cette option est seulement utilisée dans la partie mêlée générale. L'option *TotalTime* est utilisée pour l'option sprint coopératif et sprint collaboratif et représente le temps en millisecondes alloué à l'utilisateur pour trouver les images. Le champ *IsType* est utilisé pour identifier si le joueur est un joueur virtuel ou non. Les valeurs de *GameType* sont les suivantes.

- 0x01: mêlée générale
- 0x02: sprint solo
- 0x03: sprint coopératif

Prêt à commencer

Type: 62 | 0x3E	Valeur: vide

Ce message est envoyé par le client pour indiquer qu'il est prêt à commencer la partie. Il va recevoir le message (63) pour indiquer que tous les clients sont prêts lorsque ce sera le cas.

Partie commence

Type: 63 | 0x3F	Valeur: vide

Message utilisé pour signifier à tous que la partie est commencée. Le temps de chaque client peut commencer à être calculé. Les messages périodiques de la partie vont commencer ainsi que le message qui indique le début des traits (33).

Quitter partie

Type: 64 | 0x40	Valeur: vide

Ce message est envoyé par le client pour aviser le serveur que le joueur a quitté la partie. Le serveur va répondre avec le message 65 à tous les clients signifiant que la demande a bien été reçue. Il est possible que la partie soit annulée si les conditions de partie ne sont pas respectées. Il est possible que le serveur envoie en plus un trophée au joueur pour avoir quitté la partie en cours de route.

Joueur quitte partie

Type: 65 | 0x41	Valeur: {UserID:str, Username:str}

Ce message est envoyé à tous les joueurs incluant le joueur qui vient de quitter. Ce message avise qu'un joueur vient de quitter la partie en cours. Ce message est suivi d'un message de statut de partie pour mettre à jour les informations de tous les clients.

Tour du joueur de dessiner

Type: 67 | 0x43	Valeur: {UserID:str, Username:str, Time:int, DrawingID:str, WordLength: int}

Ce message à tous les joueurs de la partie. Il est utilisé pour indiquer aux autres utilisateurs qui est en train de dessiner. Le joueur qui dessine ne reçoit pas ce message, il reçoit le message 69 qui lui donne le mot à dessiner. Dans le cas d'un joueur virtuel, ce message ne lui est pas acheminé. Le champ *time* est utilisé pour indiquer le temps en millisecondes que le joueur à pour dessiner. Ce message est suivi du message 33 pour aviser le client que des messages de morceaux de traits vont commencer à arriver. Le champ *DrawingID* est utilisé pour identifier les messages de traits qui rentrent. WordLength représenter la longueur du mot à deviner.

C'est ton tour de dessiner

Type: 69 | 0x45	Valeur: {Word:str, Time:int, DrawingID:str}

Ce message est envoyé par le serveur au joueur que son tour est à dessiner. Le champ *Word* contient le mot que le joueur doit dessiner. Il peut donc envoyer le paquet 32. Il doit utiliser le champ *DrawingID* qui est un ID généré par le serveur pour identifier son dessin. Le champ *Time* représente le temps en millisecondes alloué au serveur pour dessiner.

Time's Up

Type: 71 | 0x47	Valeur: {Type:int, Word:str}

Ce message est envoyé par le serveur pour indiquer que le temps est écoulé pour le mot, ou la partie. Ce message est envoyé pour tous les participants. Les participants doivent donc afficher un message dès que ce message est reçu ou dès que leur temps local se termine. Lorsque la partie est terminée, il s'en suit du message 81 qui donne les informations sur la partie et annonce le gagnant. Le champ type peut contenir les valeurs suivantes.

- 0x01: fin temps mot
- 0x02: fin temps partie

Le champ *Word* est nul dans le cas où le type est 0x02. Sinon il s'agit du mot qu'il fallait trouver.

Synchronisation des joueurs

Type: 73 | 0x49	Valeur: {Players:[UserID:str,Username:str, Points:int], Laps:int, Time:int}

Ce message est envoyé de périodiquement à tous les joueurs pour les informer sur les informations de la partie. Les clients doivent modifier toutes les valeurs qu'ils ont en local pour refléter celle de ce message. Dans le cas du temps si le temps du client est inférieur à celui du message le temps local est gardé. La valeur du pointage du serveur est la valeur officielle et celle qui est gardée en mémoire. Le champ *Laps* est utilisé pour indiquer le nombre de tours restant à la partie. Ce champ est seulement utilisé dans la mêlée générale. Le champ *Time* est le temps restant en millisecondes pour trouver le mot. Ce champ est seulement utilisé dans le mode sprint.

Tentative deviner un mot

Type: 74 | 0x4A	Valeur: octets mot (encodage utf-8)

Ce message est utilisé par le client pour tenter de deviner un mot. Le serveur répond avec le message 75 pour indiquer si le mot a été trouvé. Il s'en suit du message 77 pour indiquer aux autres joueurs que le joueur a deviné le mot si c'est le cas.

Réponse tentative

Type: 75 | 0x4B	Valeur: {Valid:bool, Point:int, PointsTotal:int}

Ce message est la réponse au message 74 pour deviner un mot. Si le mot est vrai, les champs de pointage ne sont pas nuls. Dans le cas d'une tentative réussie, il est suivi du message point de contrôle dans le mode sprint.

Joueur a deviné mot

Type: 77 | 0x4D	Valeur: {Username:str, UserID:str, Point:int, PointsTotal:int}

Ce message est envoyé par le serveur pour aviser le client que le mot a été trouvé par un joueur. Le champ *Point* est le nombre de points qui a été obtenu pour avoir détecté le mot. Le champ *PointsTotal* est le nombre de points totaux que le joueur possède.

Point de contrôle

Type: 79 | 0x4F	Valeur: {TotalTime:int, Bonus:int}

Ce message est seulement envoyé par le serveur dans les modes sprints. Il s'agit du temps ajouté à la suite d'une détection d'un mot. Le champ *TotalTime* est le nombre de temps restant pour la partie avec le bonus. Le champ *Bonus* est le nombre de millisecondes à rajouter au chronomètre. Cette information est fournie pour l'interface. Il serait préférable d'utiliser le champ *TotalTime* pour mettre à jour le temps.

Fin de partie

Type: 81 | 0x51	Valeur: {Players:[Username:str, UserID:str, Point:int], Winner:str, Time:int, WinnerName: str}

Ce message est envoyé par le serveur pour signaler la fin de la partie. Le champ *Winner* contient le *UserID* du joueur gagnant. *WinnerName* est le nom d'utilisateur du gagnant. Le champ *Time* est le temps total de la partie en millisecondes. Ce champ est plus utile dans la partie en mode sprint.

Demande d'indice

Type: 82 | 0x52	Valeur: vide

Ce message est envoyé par un client au serveur pour demander un indice durant une partie. Ce message est suivi par le message 83.

Réponse de demande d'indice

Type: 83 | 0x53	Valeur: {UserID: string, HintsLeft: int, Hint:string, Error:string, BotID:string}

Ce message est envoyé par le serveur en réponse du message 82. Lorsqu'il y a une erreur, comme un manque de points ou de temps pour demander un indice, le champ erreur est rempli avec le message d'erreur, sinon il est vide. *UserID* est l'ID du joueur qui a demandé l'indice. *HintsLeft* est le nombre d'indices restant. *Hint* est l'indice à afficher au joueur. *BotID* est l'ID du joueur virtuel qui donne l'indice au joueur.

Fin de round

Type: 85 | 0x55	Valeur: {Players:{UserID: string,Username: string, IsCPU: boolean, Points: int, NewPoints:int}, Word:string}

Ce message est envoyé par le serveur à la fin d'un round pour informer les joueurs d'informations importantes lors de la fin d'un round. *Points* est le nombre total de points du joueur, *NewPoints* est le nombre de points que le joueur a gagné dans le round. *Word* est le mot qu'il fallait deviner durant le rond.

Mot deviné par un coéquipier

Type: 87 | 0x57	Valeur: {UserID: string, Username: string, Word: string, Points: int, NewPoints: int}

Ce message est envoyé par le serveur lorsqu'un coéquipier a deviné le mot dans une partie coop. *UserID* est l'ID du joueur qui a deviné le mot, *Username* est le nom d'utilisateur du joueur qui a deviné le mot, *Word* est le mot deviné et *NewPoints* est le nombre de points gagné pour avoir deviné le mot.

Mot manqué par un coéquipier

Type: 89 | 0x59	{UserID: string, Username: string, Lives:int}

Ce message est envoyé par le serveur lorsqu'un coéquipier fait un mauvais essai pour deviner un mot. *UserID* est l'ID du joueur qui a fait une mauvaise tentative, *Username* est le nom d'utilisateur du joueur qui a fait la mauvaise tentative et *Lives* est le nombre de vies restantes.

Partie annulée

Type: 91 | 0x5B	Valeur: {Type:int}

Ce message est envoyé par le serveur lorsqu'une partie est annulée. La valeur de *Type* indique pour quelle raison la partie a été annulée. Lorsque la valeur de *Type* est 1, la partie est annulée parce qu'un joueur n'a pas envoyé le message 62. Lorsque la valeur de *Type* est 2, la partie a été arrêtée en cours s'il ne reste plus de joueurs humains.

4.7 Messages génériques

Ces messages sont utilisés pour informer les clients de divers changements.

Avis qu'un joueur a modifié son profil public

Type: 110 | 0x6E	Valeur:{UserID: string, PictureID: string, OldName: string, NewName: string, IsCPU: boolean}

Ce message est envoyé par le serveur à tous les utilisateurs connectés lorsqu'un utilisateur modifie son nom d'utilisateur et/ou photo de profil. *UserId* est l'ID du joueur qui a modifié son profile, *PictureID* est l'ID de la photo, *OldName* est l'ancien nom d'utilisateur et NewName est le nouveau nom d'utilisateur du joueur. *OldName* est vide si l'utilisateur n'a pas modifié son nom d'utilisateur.

Avis qu'un utilisateur change sa langue

Type: 112 | 0x70	Valeur: {Langue: int}

Ce message est envoyé au serveur par un client lorsque celui-ci change sa langue. Le champ *Langue* est 0 si la langue est anglais et 1 si la langue est français.