Rapport CrazyGame

MOREAU Ameline
ZECCHINI Aymeric
VIEIRA NORO Kevin
SALMON Jeremy

Sommaire

Sommaire 2

1. Environnement de développement du projet 3

2. Architecture Clients-Serveur 3

3. Architecture Client 3

3.1 Arborescence du projet 4
3.1 Modèle vue controller 5
3.1 Navigation entre les Activitiés 6
3.2 Multilingue 7
3.3 L'interface graphique 8
3.4 Stockage des informations 9
3.5 Connexion à internet 9

4. Architecture Serveur 10

4.1 Le serveur 10
4.2 Context 11
4.3 Protocole 12
4.4 Les Logs 14
4.5 Configuration 14

5. Répartition du travail 15

6. Difficultés rencontrées 16

7. Conclusion 17

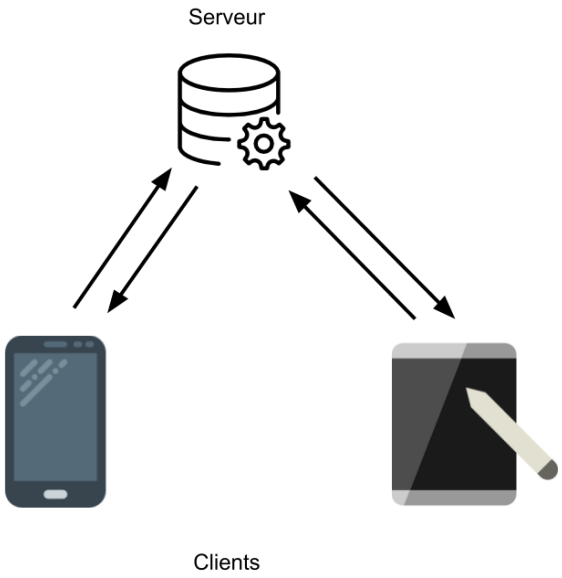
1. Environnement de développement du projet

Version Android studio : 3.0.1
Version Android : 5.0
Version APK : 21

L'application est développée en Java sous Android studio.
Le serveur est développé en java 8 sous Eclipse.

2. Architecture Client-Serveur

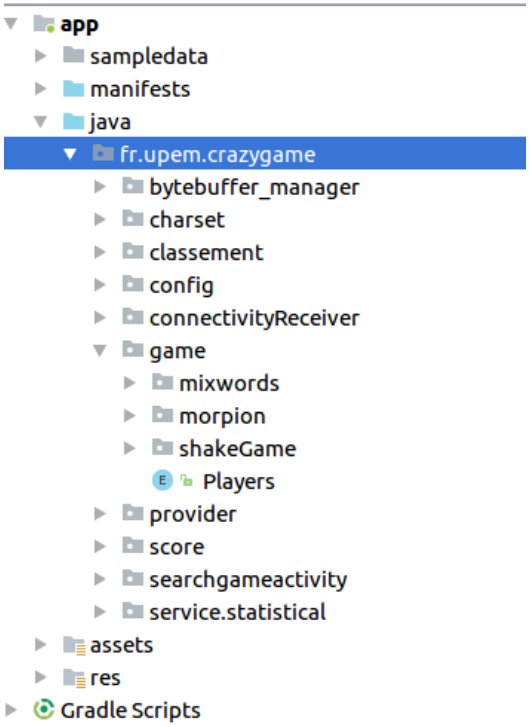
Pour l'application Crazy Game, nous sommes partie sur une application client-serveur afin de faire communiquer les différents utilisateurs de l'application.



Le serveur permet de stocker les joueurs qui recherchent un jeu sur l'application afin d'établir un pont entre ses deux joueurs.

3. Architecture Client

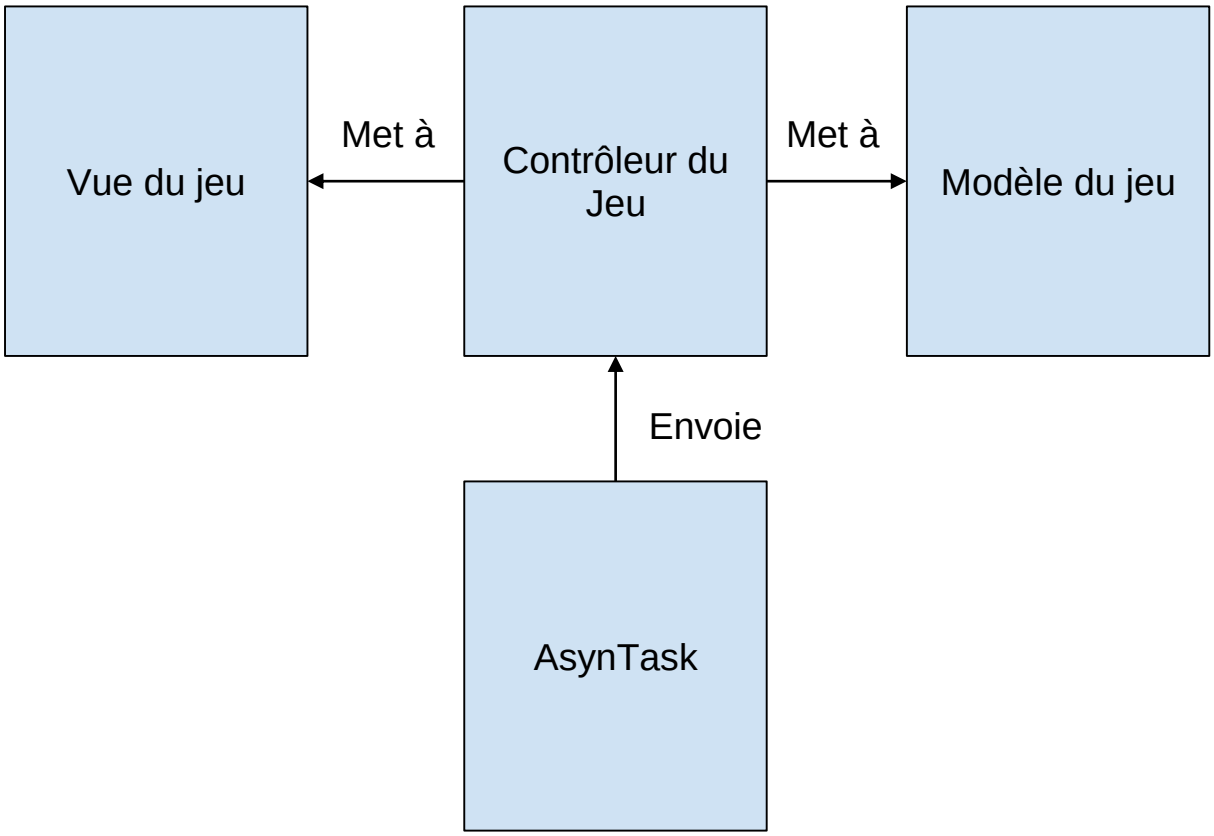
3.1 Arborescence du projet



Notre projet est décomposé en 10 package différents:

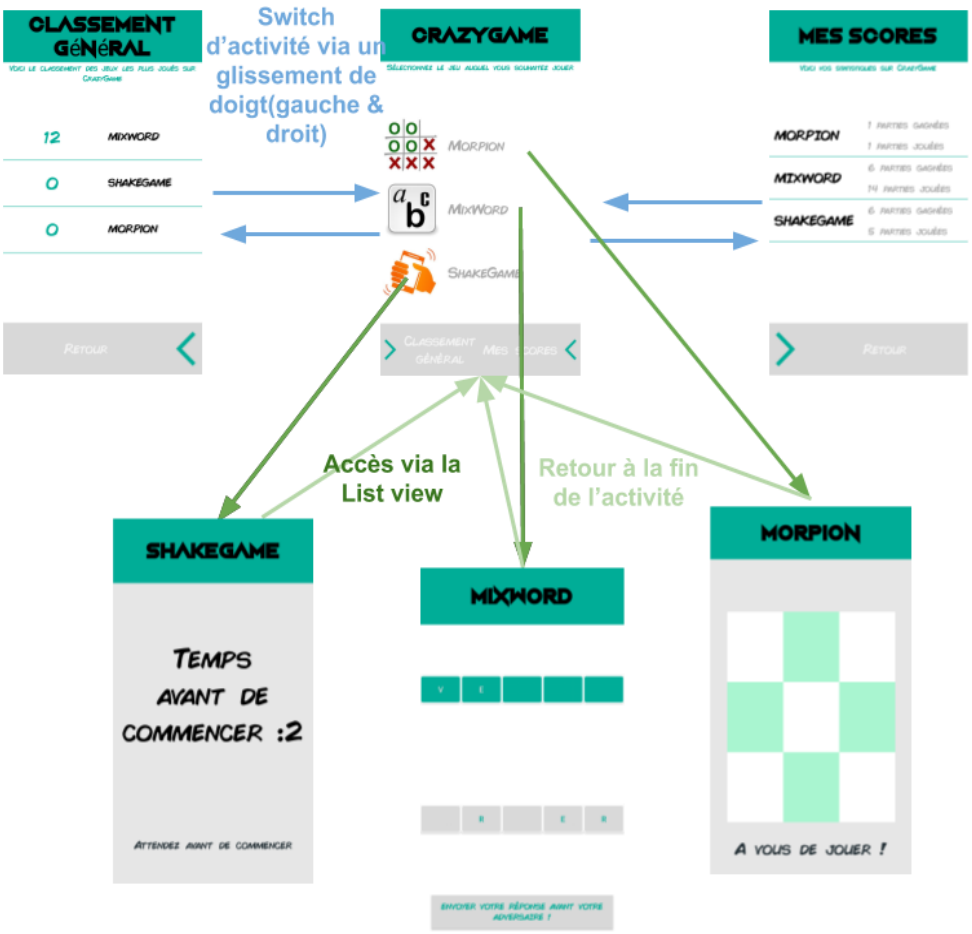
- bytebuffer_manager : Représente des méthodes statiques sur des bytebuffers
- charset: Contient notre Charset en UTF-8
- classement : Contient les classements des jeux les plus jouées dans le monde
- config: Permet de représenter les choix de volumes et de vibreurs
- connectivityReceiver: Permet de tester la connexion internet
- game: Représente nos jeux
- provider: Représente le provider
- score: Représente les jeux auxquels le joueur a joué.
- searchgameactivity: Représente l'activité principale
- service.statistical: Représente notre service

3.1 Modèle vue contrôleur



Pour les jeux de CrazyGame, nous nous sommes fortement inspirés du pattern MVC. Nous avons une classe Modèle qui sera les données du jeu, le contrôleur qui permet de contrôler les données et va notamment instancier une AsyncTask qui s'occupe de gérer l'échange de donnée à travers le réseau. La vue est tout simplement l'activité de notre jeu.

3.1 Navigation entre les Activités



Le menu principal de CrazyGame est composé de la liste des jeux. Pour accéder à un jeu il suffit de cliquer dessus, une fois qu'un second joueur sélectionne le même jeu alors la partie est lancée. A la fin de la partie on retourne au menu de CrazyGame.

Depuis ce menu il est également possible d'accéder à ses scores et au classement général. Lors de ses changements d'activités, on applique **une animation** qui permet à l'activité de glisser vers la gauche ou vers la droite en fonction du geste effectué par l'utilisateur.

3.2 Multilingue

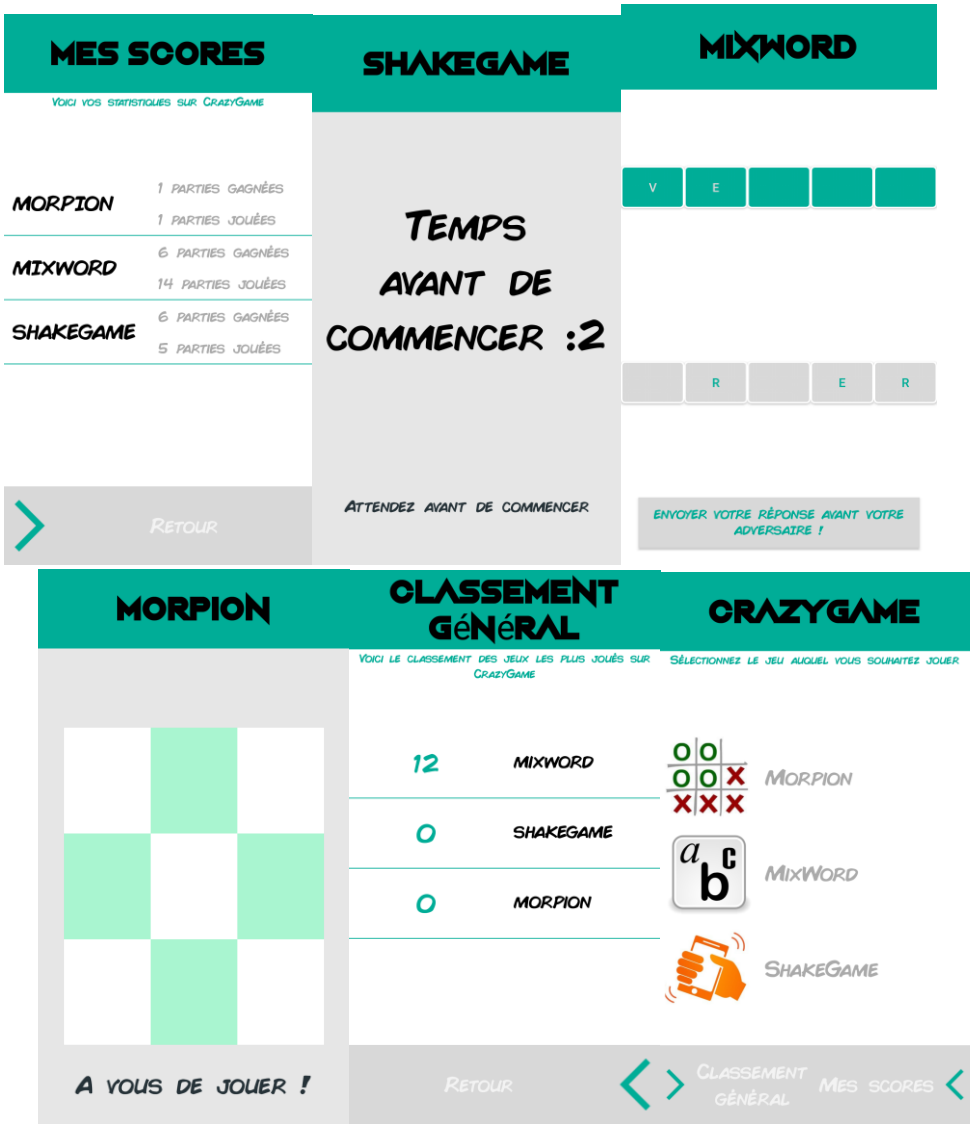
CrazyGame est disponible en français ainsi qu'en anglais. La langue de notre application s'applique en fonction de la langue par défaut du téléphone.
Pour permettre à notre application d'être multilingue, nous avons utilisé le fichier string.xml. Ce fichier se décline en 3 string.xml, l'un contenant les champs en français, un autre contenant ces même champs en anglais et celui par défaut est aussi en anglais.

3.3 L'interface graphique

Afin de rendre notre application plus « fun » nous avons utilisé des fonts et des images dans nos layouts. Les différentes fonts utilisées sont situé dans res>font. On initialise les fonts dans le code Java à la création de l'application en cours.

Pour les images, elles sont situé dans le répertoire res>drawable. Les images sont appelées directement dans le code xml.

Toutes nos pages ont le même template :



Ce template s'adapte aussi bien en mode portrait qu'en mode paysage et peut s'adapter à toutes les tailles d'écrans. Cependant, nous avons rencontrées des difficultés pour la reprise d'activité que nous détaillerons dans la partie "difficultés rencontrées".

3.4 Stockage des informations

Le stockage des données de notre application se fait en interne, à l'aide d'un **Content Provider**, et sur le serveur dans une hashmap.

Dans le content provider nous stockons pour chaque jeu le total des parties joués ainsi que le nombre de parties gagnées. Ces données sont affichées dans les scores du joueur.

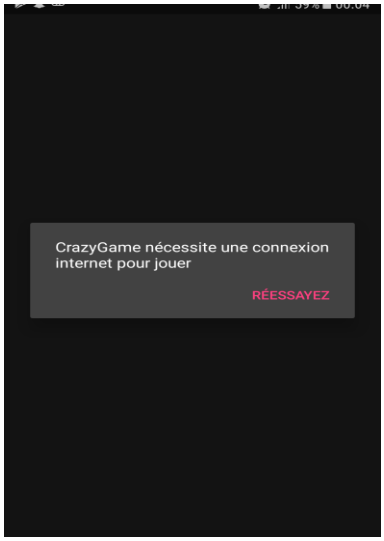
On a également ajouté pour chaque jeu le nombre de partie jouée depuis un certain laps de temps. Cette colonne est utiliser par un **service** qui récupère cette données pour l'envoyer au serveur toutes les X secondes (nous avons mis 10 secondes pour tester). Une fois récupérer, cette colonne est remise à 0. Ce service est lancé dès le démarrage du device à l'aide d'un **broadcaste receiver déclarer dans le manifeste**.

```
<receiver android:name=".service.statistical.broadCastReceiverBoot.BroadCastBoot">
    <intent-filter android:priority="100">
        <action android:name="android.intent.action.BOOT_COMPLETED" />
    </intent-filter>
</receiver>
```

Ces données permettent au serveur de faire des statistiques sur l'utilisation de CrazyGame par les utilisateurs. Pour l'instant nous affichons le classement général des jeux les plus joués sur CrazyGame en les classant par nombre de partie joué. Mais nous pouvons imaginer par la suite que nous pourrions récupérer bien plus d'informations dans le but d'améliorer le jeu (classement des meilleurs joueurs par exemple).

3.5 Connexion à internet

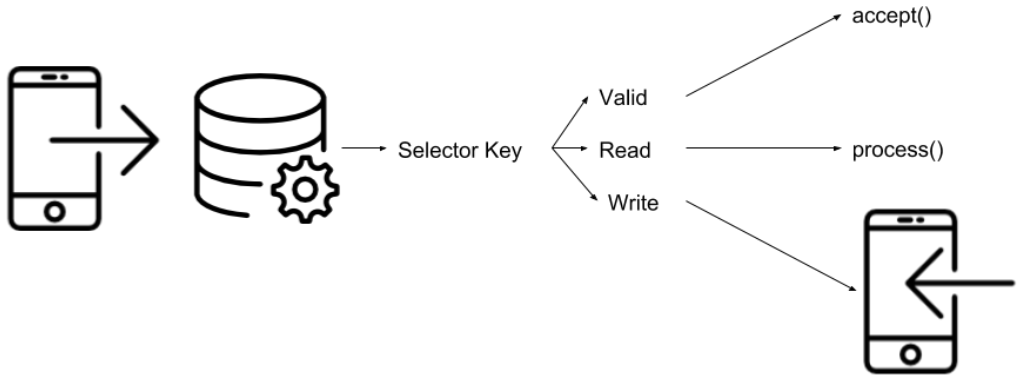
CrazyGame nécessite une connexion internet pour pouvoir lancer une partie avec un autre joueur. Un broadcast receiver déclarer dynamiquement permet de vérifier la connexion internet (Wifi et données mobile). Si l'utilisateur n'a pas de connexion internet un message s'affiche.



4. Architecture Serveur

4.1 Le serveur

Pour CrazyGame, nous avons créé notre propre serveur basé sur des connexions TCP et un échange de requête au format UTF-8. Le serveur est écrit en Java et afin de faire un serveur sans concurrence, nous nous sommes servis des Selectors et des ByteBuffers proposé par Java.



En effet, l'instruction "for (SelectionKey key : selectedKeys)" permet de parcourir les connexions qui ont envoyé des données et cela va réveiller la selectedKey et il nous suffit donc de vérifier l'état de la key (client) et d'établir le traitement correspondant:

- isAcceptable va permettre de tester si c'est une nouvelle connexion et dans que cas nous acceptons la connexion en lui attribuant un nouveau Context
- isRead va vérifier que le buffer d'entrée à des données et alors nous allons effectuer le processus afin d'interpréter ses données. Cela va notamment appelé la méthode process() qui contient tous les états et les instructions à effectuer en fonction de l'état du client.
- isWrite va permettre d'envoyer les données contenu dans le buffer de sortie au client.

Ces traitements vont être synchrones et chaque key réveillé va être traité une après l'autre.

4.2 Context

Chaque connexion client dans le serveur est représentée par une selectedKey. Une selectedKey peut contenir un objet et nous nous sommes servies de cela pour créer un contexte pour chaque key.

Lorsque nous recevons une nouvelle connexion, nous lui attachons une classe Context qui va permettre notamment de contenir:

- buffer d'entrée
- buffer de sortie
- socketChannel associé à la key qui va permettre d'envoyer ou de lire des données à travers le réseau avec les méthodes read et write
- Instance du serveur afin notamment de pouvoir loguer le client avec
- .PlayerService qui va permettre de gérer le nom de jeux que le joueur recherche et de renvoyer une partie s'il en trouve une avec le numéro de joueur qui commence, 1 si il commence et 2 sinon
- Une instance de ServerGameManager qui s'occupe de la gestion des parties afin de contenir les joueurs qui sont en attente de partie. Cela va permettre de s'ajouter dans la liste des joueurs en attente si le context le demande.
- Une instance de ScoreManager, cela va permettre au client qui sera dans ce cas un service Android de demander le nombre de partie totale joué sur le serveur.

4.3 Protocole

Sur le serveur CrazyGame, le client ne peut être que dans un état la fois. Nous avons représenté ces états par une enum dont nous nous servons dans la méthode process d'un Context:

- WAIT_LENGTH_LANGUAGE: Lorsqu'un client se connecte, la première chose qu'attends un client est la langue du client en question. Le client va donc envoyer une requête de la forme:

  2 fr

- WAIT_LANGUAGE: Lorsque le client va recevoir l'int de la langue, ce dernier rentrera dans cet état qui permettra de traiter la chaîne de caractère de la langue.
- WAIT_PACKET: Une fois que nous avons reçu la langue du joueur, il se retrouve en attente. Quand nous recevons un paquet, selon le premier entier que nous recevons nous switchons dans les autres états.

  1 signifie que nous recherchons un jeu. Rentre dans l'état WAIT_LENGTH_NAME_GAME

  4200 permet de signifier que le client envoie des informations du nombre de partie qu'il a fait sur chaque jeux. Ce client est en fait le service Android. Rentre dans l'état WAIT_SCORE

  4300 permet de demander les scores totaux des joueurs. Rentre dans l'état WAIT_SCORE_WORLD

- WAIT_SCORE_WORLD: Cet état permet de renvoyer les scores totaux. Nous rentrons dans cet état quand nous recevons 4300. On va renvoyer une requête au client de la forme suivante;

  3 7 Morpion 12 9 ShakeGame 4 7 MixWord 3

Le premier « 3 » signifie le nombre de jeux, puis nombre de caractère, nom du jeux, nombre de partie jouées.

- WAIT_SCORE: Le serveur additionne le nombre de partie auquel a joué le joueur dans une hashmap qui stocke les nombres de parties jouées. Nous aurions pu directement additionnée une partie dans le serveur mais nous pensions faire des jeux hors ligne donc il y avait un réel intérêt à faire que ce soit le client Android qui envoie l'information.

  Le client va envoyer des requêtes de la forme:

  7 MixWord 3 9 ShakeGame 400

  Le serveur sait que le client a finis d'envoyer des données quand nous recevons le code 400.

- WAIT_LENGTH_NAME_GAME: Attend la longueur de la chaîne de caractère du nom du jeu auquel veut jouer le client, exemple:

  1 7 Morpion
- WAIT_NAME_GAME: récupère la chaine de caractère auquel veut jouer le client.
- WAIT_ADD_GAME: Ajoute le jeu auquel veut jouer le client dans la liste d'attente
- WAIT_GAME: Attend de trouver une partie. La méthode sendGameFound() du serveur permet de parcourir la liste des joueurs en attente et de faire varier les états lorsqu'une partie est créé. Quand le serveur crée une partie, le serveur envoie au joueur une requête de la forme:

  1 7 Morpion 1

  Le dernier 1 représente le joueur qui commence.

Si c'est une partie de MixWord, on doit rajouter le mot mélangé:

1 8 MixWord 1 7 jourbon

- GAME_MORPION: Le client envoie un coup et le serveur le renvoie à l'autre client.

  Le joueur qui joue le coup va envoyer une requête de la forme:

  1 0 0

  1 est un id qui veut dire que la partie n'est pas finie, et 2 pour dire que la partie est fini et qu'elle peut être supprimé du serveur. (0,0) représente le coup joué par le joueur.

- WAIT_PLAYER_MORPION: Le client attend de recevoir le coup de l'autre joueur
- GAME_MIXWORD: Le client envoie un mot. Si le client trouve alors on renvoie au second client qu'il a perdu. La requête va être de la forme:

  7 bonjour

  Le serveur va renvoyer **3** si le mot est mauvais et

  1 si il a trouvé le mot et envoie **2 7 bonjour** au joueur qui a perdu puis on supprime la partie.

- GAME_SHAKE: Le client envoie un score et le renvoie à l'adversaire. On reçoit un score et on le renvoie à l'autre joueur puis on supprime la partie.

4.4 Les Logs

Il est très difficile de déboguer un serveur. Pour cela, nous avons créé des logs dans notre environnement et pour cela nous avons créé une classe de Logs. Elle contient des méthodes qui vont permettre d'écrire dans un fichier à la date du jour.

Sat Mar 31 17:28:54 CEST 2018 /90.3.251.211 a envoyé une longueur de taille de langage 2
Sat Mar 31 17:28:54 CEST 2018 /90.3.251.211 a envoyé le langage fr
Sat Mar 31 17:28:54 CEST 2018 /90.3.251.211 Integer receptionné
Sat Mar 31 17:28:54 CEST 2018 /90.3.251.211 est prêt à envoyer des scores de jeux
Sat Mar 31 17:28:54 CEST 2018 /90.3.251.211 a envoyé des données de jeux
Sat Mar 31 17:28:55 CEST 2018 /90.3.251.211 a envoyé des données de jeux
Sat Mar 31 17:28:55 CEST 2018 /90.3.251.211 Integer réceptionné
Sat Mar 31 17:28:55 CEST 2018 /90.3.251.211 Action demandée: 1
Sat Mar 31 17:28:55 CEST 2018 /90.3.251.211 Longueur du nom: 9
Sat Mar 31 17:28:55 CEST 2018 /90.3.251.211 Nom: ShakeGame

4.5 Configuration

Pour notre serveur, nous avons créé un fichier de configuration. Il est au format JSON et ne contient que 2 lignes.

```
{
	"Port": 8086,
	"RepLogs": "./Logs/Log"
}
```

Nous pouvons désérialiser notre objet simplement à l'aide d'une bibliothèque (Jackson) et d'annotations.

5. Répartition du travail

Pour CrazyGame, nous devions réaliser un serveur et un client avec plusieurs jeux. De là, il nous apparut évident de décomposer les tâches de la façon suivante:

Une personne a dû se charger du **serveur**, nous avons attribué cela à Salmon Jérémy.

Nous avons défini que pour les **jeux** Ameline allait se charger plutôt des **layouts** car elle a un côté créatif plus développé que nous tous.

Kévin s'est chargé de créer la structure des jeux avec Aymeric et Aymeric s'est chargé également de s'assurer de la bonne communication entre le client et le serveur et de s'assurer que le protocole définit répondait à notre besoin. Aymeric s'est donc chargé de l'implémentation des **AsyncTasks**.

Nous devions également mettre en place le provider, le **multilingue**, le **service** et les différents **receivers**.

Ameline s'est plutôt chargée du **service** et de son **receiver** au démarrage du téléphone, Kévin du receiver qui permet de tester la **connexion internet** et Aymeric du **provider**.

Une fois que tous les écrans étaient définis, nous avons essayé de faire remonter au maximum les bugs que ce soit côté client ou serveur et nous les avons corrigées au maximum pour la soutenance.

Après la soutenance, nous avons continué à corriger quelques bugs et nous avons rajouté quelques fonctionnalités ainsi que quelques modifications:

- bugs de provider
- bloquer la rotation de l'écran pendant le jeu (cf. difficultés)
- vibreur quand on perd une partie
- layout
- Pour ShakeGame, nous faisons désormais la moyenne du secouement et non plus le maximum comme score.
- Ajout du son
- Enregistrement du choix de volume et de vibreur
- Choix d'activité le son et la vibration

6. Difficultés rencontrées

Lors de ce projet, nous avons rencontrées une suite de difficulté plus ou moins critique que nous avons corrigée.

La première grosse difficulté et l'utilisation de **font sur nos layout**s. Nous les avons ajoutés dans le dossier assets et dans le xml mais certains téléphones ne détectaient pas ces fonts. Nous n'avons pas réussi à comprendre pourquoi ce phénomène et nous avons dû trouver une alternative en créant les font directement dans le code Java.

Une seconde difficulté que nous n'avons pas réussi à implémenter est la **rotation d'écran pendant un jeu.** En effet nous utilisons des AsyncTask et il est très difficile reprendre proprement une activité alors qu'une AsyncTask est en cours de traitement. Nous avons donc bloqué la rotation de l'écran quand nous sommes en cours de partie. Seule la rotation de l'écran est possible dans le menu principal et nous gardons cette orientation pendant le cour du jeu.

La plus grosse difficulté a été la gestion du protocole. En effet il était impératif que le client soit toujours dans un **état cohérent par rapport au serveur**. Pour simplifier le problème, nous avons défini que le client ne pouvait être que dans un seul état et c'est pour cela par exemple que nous n'avons pas implémenté de chat. En effet, nous voulions à la base que le client puisse parler en même temps que jouait nous avons eu peur de ne pas avoir le temps de bien gérer cela. Nous avons essayé de corriger au maximum la possibilité d'état incohérent et de gestion d'exception du serveur. Le serveur est assez stable mais nous avons relevé encore quelques problèmes que nous n'avons pas eu le temps de corriger.

Nous avons rencontré une difficulté au niveau de la **listview du menu**. A un moment donné, nous avons voulu modifier le layout de la listView et cela ne fonctionnait pas. Le contenu était changé mais la vue restait identique. Pour pallier ce problème nous avons dû créer un nouveau projet.

7. Conclusion

Le projet Android fût une bonne expérience de développement mobile. Il nous a permis de choisir un sujet libre, et cela nous a permis de laisser cour à notre imagination. Cependant, l'imagination et la réalité ne font pas toujours bon ménage. En effet, nous avons été confronté à choisir l'implémentation de notre serveur et nous n'avions pas spécialement d'expérience dans ce domaine et nous nous rendons compte que ce dernier n'est pas très modulable dans le sens où nous devons pour chaque jeux et même action définir un nouvel état et protocole.

La structuration du projet Android a elle aussi été difficile à mettre en place et nous sommes parties sur un MVC pour l'implémentation de nos jeux. Pour ce choix, je pense que nous avons fait une bonne organisation mais le code aurait quand même pu être plus optimisé et plus structuré dans le sens ou chaque jeux aurait peut-être pu définir une structuration définit par une interface afin de faciliter l'ajout de nouveaux jeux et de ne pas devoir ajouter du code dans l'activité SearchGame par exemple.

Pour pallier ces problèmes, nous avons essayé de définir un protocole plus générique qui permettrait de ne pas avoir à modifier l'ajouter de code dans le serveur lors d'ajout d'un jeu. Cependant cela n'est pas réalisable pour le cas de notre MixWord par exemple.

En effet, nous générons un mot et nous testons si le mot est correct et cela est spécifique à notre jeu. Nous pouvons quand même générique avec le protocole suivant mais nous aurons toujours besoin pour certain type de jeux d'intervention côté serveur:

Chercher une partie:
1 7 MixWord
Trouver Partie MixWord:

1	7	MixWord	1	7	jourbon
Id	**Taille**	**jeux**	**nom jeux**	**joueur commence**	**taille data**	**data**

Trouver Partie Morpion:
1 7 Morpion 1 0
Le transfert de donnée entre deux clients se ferait sous la forme

Id	taille jeux	nom jeux	Data

Les ids du serveur permettent de laisser libre court au client de définir à quoi correspond un id si besoin.

Avec ce protocole, nous n'aurions besoin d'ajouter que du code si les jeux en question nécessite l'envoie de data particulière lorsqu'un joueur trouve une partie.

Pour le client, nous avons aussi essayé de réfléchir à une généricité de nos jeux. Pour cela, nous devrions définir des interfaces et l'idée et qu'il n'y aurait plus besoin d'ajouter du code directement dans l'application mais de chargé dynamiquement les jeux à partir d'une source. Cela permettrait à n'importe qui de fournir cette source et de le faire partager à tous les possesseurs de l'application. Nous n'avons pas trouvé de solution mais la principale idée et que nous avons réfléchi à du code stocker sur un serveur au format jar que nous stockerons sur serveur spécifique. Au lancement de l'application, le client chargerait les jeux qu'il ne possède pas avec un ClassLoader par exemple mais nous n'avons pas trouvé de solution pour stocker ces classes et éviter d'avoir à les charger à chaque lancement. L'idée retenue serait peut-être l'utilisation d'une base de donnée avec des objets sérialisé mais nous ne sommes pas sûr que cela soit possible.