Université Grenoble Alpes
Adrien ZOFFRANIERI
Aleksandra BASANGOVA
Jean-Baptiste ARBAUT
Marisela FLORES HERNANDEZ
Van-Louis TRAN

Application Web Mobile - Jeu TRON

Décembre 2025

I. Architecture technique

A. Présentation générale

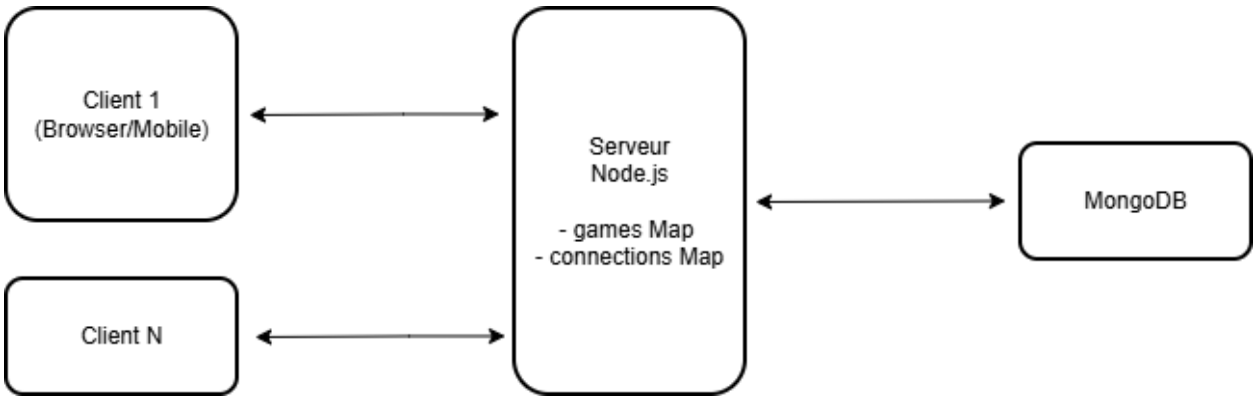
Le projet repose sur une architecture client-serveur avec une communication via WebSocket. Les technologies imposées sont les suivantes :

1. Serveur : Node.js
2. Client : HTML/CSS/Javascript et Cordova pour le multi-plateforme.
3. Base de données : MongoDB pour le stockage des comptes des joueurs et l'historique des parties jouées

Dans notre architecture, le serveur gère l'état du jeu : les positions et les états des joueurs, les collisions, les victoires, etc. Pour cela, il réalise les calculs avec la grille de jeu qu'il stocke. Il envoie l'état du jeu mis à jour à l'ensemble des joueurs de la partie dans un intervalle. Le serveur s'occupe donc de synchroniser les joueurs entre eux. Les clients, de leur côté, ont seulement besoin d'envoyer leurs commandes de mouvement et ils reçoivent les mises à jour de l'état du jeu à intervalle fixe. Les clients affichent alors l'interface graphique en fonction de l'état du jeu reçu de la part du serveur.

Nous avons décidé de séparer les rôles du serveur et des clients ainsi pour empêcher la triche du côté des clients. En effet, dans notre architecture, le serveur est l'unique source de vérité : il vérifie donc la cohérence de l'état du jeu.

Schéma général :



Flux de données :

Clients ➔ Serveur : commandes utilisateur (*login*, mouvement, création *lobby*)
Serveur ➔ Clients : état du jeu, mises à jour synchronisées
Serveur ↔ MongoDB : lecture/écriture des comptes des joueurs, écriture des parties passées

Dans les sections suivantes, nous allons présenter le serveur et le client d'un point de vue général. Nous nous attarderons plus en détail sur les raisons qui expliquent nos choix dans la seconde partie du rapport dédiée à l'application.

B. Présentation du serveur

Nous nous intéressons ici au fonctionnement du serveur. Le serveur reçoit les requêtes du client grâce au fichier `WebsocketServer.js`. En fonction de la requête client, qui peut par exemple être une demande de créer, rejoindre ou quitter un *lobby*, le serveur redirige la requête vers une fonction spécifique du fichier `GameHandler.js`. Pour prendre un exemple concret, si le client envoie une requête de création de partie, la fonction `handleCreateGame()` de `GameHandler.js` est appelée. Après avoir vérifié que la requête ne contient pas d'erreurs (données manquantes, par exemple), le serveur crée un objet de la classe `Game` (`Game.js`) qui contient la liste des joueurs. Il ajoute également la partie à la `Map` des parties en cours. Enfin, le serveur informe les clients de la création d'un nouveau *lobby*, qu'ils peuvent désormais rejoindre.

La classe `Game` contient la logique de jeu. Elle contient en effet une grille de 100 par 100, le nombre de joueurs maximum pouvant rejoindre, un statut et la liste des joueurs de la partie. A sa création, le statut de cette classe est "`lobby`". Il change à "`game`" lorsque la partie démarre et passe à "`gameEnded`" à la fin de cette dernière. Nous gardons et mettons à jour le statut de la partie pour faire des vérifications. Par exemple, nous vérifions, lorsqu'un client demande de rejoindre une partie, que cette dernière a bien le statut "`lobby`". Si ce n'est pas le cas, cela signifie que la partie a déjà commencé ou qu'il s'agit d'une ancienne partie s'étant terminée. Dans les deux cas, nous ne faisons pas suite à la demande du client. Lorsque la partie démarre, cela déclenche un intervalle qui appelle toutes les 100 millisecondes

la méthode `update()`. Cette méthode vérifie les collisions entre les joueurs et si la partie s'est terminée. Elle met également à jour les positions des joueurs. A la suite d'`update()`, la fonction `updateAllPlayerMovements()` est appelée et envoie à tous les joueurs de la partie les informations mises à jour sur l'état des joueurs, afin que leur interface soit synchronisée avec les données du serveur.

La classe `Player`, quant à elle, permet de garder dans la mémoire du serveur les informations sur les joueurs durant une partie : leur position, leur direction actuelle, leur couleur (qu'ils choisissentt à chaque début de partie), leur état (prêt ou non, en vie ou non). La classe expose des méthodes permettant de changer la couleur, la direction, la position d'un joueur.

C. Présentation du client

Pour ce qui est du client, celui-ci reçoit les réponses du serveur dans le fichier `WebsocketClient.js`. En fonction de la réponse, il s'agit d'un *handler* différent qui réalise le traitement. Par exemple, le `ConnectionHandler.js` traite la réponse qui a lieu après la connexion d'un joueur et le `LeaderboardHandler.js` s'occupe des réponses ayant lieu après le clic sur le bouton "Classement". Nous disposons de fichiers *handlers* différents, qui traitent chacun un aspect différent de l'application : nous avons le `ConnectionHandler` pour la connexion des joueurs, le `ControlHandler` pour les contrôles, le `GameHandler` pour gérer ce qui se passe durant la partie (mettre à jour de l'état des joueurs graphiquement, gérer la fin de partie, etc.), le `LobbyHandler` pour gérer ce qui se passe avant le début de la partie (quitter le lobby, mettre à jour les informations de lobbies, changer la couleur d'un joueur, etc.) et le `LeaderboardHandler` pour le classement des joueurs. Le client dispose également d'une interface HTML, divisée par section (chaque section correspondant à un écran de notre application, qui est une *Single Page Application*). Chaque *handler* ajoute sur la page HTML des *eventListeners* pour envoyer des requêtes sur le clic d'un bouton, notamment.

D. Présentation de la base de données

PlayerModel - stockage des comptes des joueurs

```
{
  username: {
    type: String,
    unique: true,
  },
  password: String,
  wins: Number,
  losses: Number,
}
```

GameModel - stockage des parties terminées pour l'historique et les statistiques

```
{
  generatedGameId: String,
  name: String,
  players: [String],
  winnerName: String,
  startedAt: Date,
  endedAt: Date,
}
```

Notre base de données est une base MongoDB qui contient deux collections uniquement : `players` et `games`. Nous n'avons pas eu besoin de plus de collections.

Nous stockons les informations liées au compte des joueurs dans la collection `players` : le nom d'utilisateur et le mot de passe. Le nom d'utilisateur est unique pour permettre une connexion et inscription rapides sur l'application. Si le nom d'utilisateur n'existe pas encore dans la base de données, l'application crée un nouveau document `Player`. Si l'utilisateur existe, l'application vérifie si le mot de passe est correct. Nous stockons également les statistiques de parties des joueurs dans `wins` et `losses`. Ce sont des attributs qui sont redondants avec ceux stockés dans la collection `Game`, mais étant donné la nature du NoSQL, nous avons jugé que la non-redondance des données était moins essentielle à respecter car procéder de cette façon nous permet de récupérer les informations du classement très facilement. En effet, il suffit de récupérer les 5 joueurs ayant le plus de victoires.

Dans la collection `games`, nous stockons pour chaque document l'identifiant généré par `bcrypt` (dans la classe `Game.js`) qui est utilisé tout au long du cycle de vie des paquets. Nous n'utilisons en réalité pas l'identifiant généré par Mongoose. Nous gardons l'identifiant `bcrypt` pour faciliter la récupération de parties spécifiques à l'avenir, dans le cas où nous voudrions ajouter d'autres fonctionnalités comme le *replay*. Nous stockons également le nom de la partie, qui n'est pas unique ici car une partie rejouée possède le même nom que l'ancienne partie, d'où la nécessité d'un identifiant. Pour ce qui est des joueurs, nous stockons uniquement la liste de leur nom d'utilisateur. Auparavant, nous stockions plus d'informations sur les joueurs, comme leur couleur. Nous faisions cela car, nous récupérions les informations de l'ancienne partie depuis la base de données pour créer une nouvelle partie avec les mêmes attributs. Nous avons simplifié ce processus en ne supprimant plus l'ancienne partie avant d'avoir créé la nouvelle, ce qui permet de ne stocker que les attributs réellement pertinents en base de données. En plus du nom du gagnant, nous avons également des attributs *timestamps* pour proposer à l'avenir un historique des parties avec la durée de chacune d'entre elles, et le gagnant (s'il y en a un).

L'implémentation de la structure de base nous a pris environ 1 heure, et nous l'avons ensuite adapté à mesure que nos besoins évoluaient (ajout de `wins` et `losses` au `PlayerModel`, notamment).

E. Format des paquets

Le format de chacun des paquets est disponible dans le fichier `specification_paquets.md`. Nous avons utilisé ce document pour s'assurer que les paquets aient le bon format entre client et serveur. Nous avons décidé d'homogénéiser les paquets en leur donnant tous un attribut "`type`", qui spécifie l'action du paquet. Les types qui contiennent le mot "`Response`" sont des réponses du serveur au client. Elles contiennent un attribut booléen "`valid`", qui indique si l'action demandée par le client a été effectuée ou non. Si "`valid`" est à "`false`", un attribut "`reason`" est présent pour indiquer la raison de l'erreur. Cela permet de débugger plus facilement et de proposer des messages d'erreur sur l'interface client. En effet, il suffit alors de vérifier la valeur de "`valid`", puis d'afficher la chaîne de caractères de "`reason`" si "`valid`" est à "`false`". Nous avons modifié la spécification des paquets tout au long du développement de l'application, à mesure que nous ajoutions des fonctionnalités. Nous avons mis environ 5 heures pour avoir notre spécification finale (documentation et implémentation).

II. Application

A. Fonctionnalités

L'application permet de jouer au jeu TRON en multijoueur de 2 à 4 joueurs. Elle présente un système d'authentification des joueurs. Pour créer un nouveau compte, il suffit de choisir un pseudonyme non utilisé et le mot de passe associé devient le mot de passe du nouveau compte créé. L'application permet la création et la recherche de *lobbies* (de 2 à 4 joueurs). Les *lobbies* sont mis à jour dynamiquement et sont supprimés automatiquement s'ils ne contiennent plus de joueurs. Quand trop de *lobbies* sont créés, un système de page se met en place automatiquement pour simplifier la navigation entre les *lobbies.* Il est possible de jouer plusieurs parties en parallèle. De plus, les joueurs peuvent choisir leur couleur parmi les 4 proposées avant le lancement de la partie. Un système de statut « Prêt » permet aux joueurs de signaler qu'ils sont prêts à jouer, la partie ne démarrant que lorsque tous les participants sont prêts. Lorsqu'un joueur meurt au cours d'une partie, que ce soit à la suite d'une déconnexion ou pendant la partie, son corps reste affiché sur la grille de jeu mais apparaît en transparence, afin de permettre aux autres joueurs d'identifier clairement les joueurs éliminés et de mieux comprendre l'état de la partie en cours. À la fin d'une partie, il est possible de relancer une partie et de voir si un autre joueur souhaite rejouer également. Il y a un système de classement par victoire où l'on peut voir les 5 meilleurs joueurs. L'interface de l'application est *responsive* et s'adapte donc à différents types d'appareils, que ce soit mobile ou ordinateur. Les contrôles sur mobile sont assurés par des boutons de direction directement intégrés à l'interface de jeu, tandis que les joueurs sur ordinateur utilisent au choix ces mêmes boutons ou les flèches directionnelles du clavier.

B. Choix globaux

Nous avons effectué des choix globaux pour notre application.

Premièrement, nous avons décidé du fait que l'égalité n'existe pas dans notre jeu. Soit un joueur gagne, soit il perd. S'il n'y a aucun gagnant, c'est que tout le monde a perdu. Par exemple, lorsque les deux derniers joueurs d'une partie entrent en collision, les deux perdent. Il s'agit d'un parti pris; dans un sens cela permet de donner aux joueurs une attitude plus offensive. Pour implémenter ce système et s'assurer que deux joueurs (ou plus) qui entrent en collision perdent, nous calculons dans `Game.js` la prochaine position des joueurs avant de les déplacer pour vérifier leur potentielle collision. C'est seulement après être passé par tous les joueurs que nous changeons le statut `alive`, pour que tous les joueurs entrés en collision meurent en même temps. Nous avons mis environ 2 heures pour implémenter ce choix.

Deuxièmement, nous avons intégré la possibilité de jouer à 3 joueurs. Les parties à 2 et à 4 joueurs sont par nature équilibrées car il y a un joueur de chaque côté. Cependant, ce n'est pas le cas pour une partie à 3 joueurs, où un des joueurs n'a pas d'adversaire en face. Même si cette configuration peut présenter un déséquilibre (qui reste léger par ailleurs), nous avons choisi de privilégier une certaine flexibilité dans le choix du nombre de joueurs. S'il n'y a que 3 joueurs jouant sur notre application, ils pourront donc jouer ensemble sans avoir à alterner en faisant des parties à 2. Nous n'avons pas fait d'implémentation spécifique pour ajouter la possibilité de jouer à 3, car notre code fonctionne avec des boucles. Pour implémenter les parties de 2 à 4 joueurs, nous avons mis environ 2 heures.

Troisièmement, nous avons décidé qu'un joueur se déconnectant au cours d'une partie meurt. Nous trouvions que procéder de cette manière était plus simple que de permettre au joueur déconnecté de revenir pour poursuivre la partie. Le joueur déconnecté arrête tout mouvement : il meurt sur place. Si nous avions choisi de permettre la reconnexion, il aurait aussi fallu décider s'il fallait arrêter le mouvement d'un joueur déconnecté et le reprendre à sa reconnexion, ou bien le laisser avancer en ligne droite tant qu'il reste déconnecté. Cependant, arrêter le joueur dans sa course et le laisser reprendre serait trop avantageux pour le joueur. D'un autre côté faire en sorte que le joueur continue en ligne droite ne laisse pas vraiment le temps au joueur de se reconnecter avant de mourir, les mouvements étant très rapides. Par ailleurs, faire disparaître le corps du joueur à sa déconnexion peut entraîner un avantage puisqu'un joueur peut être sauvé par la disparition du corps. Ainsi, pour simplifier cette situation et par souci d'équilibrage, nous avons décidé que se déconnecter signifie mourir et que le corps ne disparaît pas de la grille de jeu. Pour mettre en place cette fonctionnalité, nous avons mis environ 2 heures.

Quatrièmement, notre application permet aux joueurs de sélectionner le *lobby* qu'ils souhaitent rejoindre. Nous avons fait ce choix au début de notre conception en nous inspirant du fonctionnement de certains jeux en ligne. Ce choix a amené une complexité supplémentaire à notre application car pouvoir rejoindre un *lobby* au choix implique de posséder l'identifiant du *lobby* et donc des communications entre client et serveur supplémentaires. Il a aussi fallu gérer l'affichage de la liste des

lobbies et prendre des décisions, notamment au niveau du *broadcast* pour mettre à jour les informations sur les *lobbies*. En effet, pour mettre à jour les informations sur les *lobbies* existants, nous avons choisi de faire des *broadcasts* à l'ensemble des joueurs connectés. Il s'agit d'un choix qui peut se discuter mais nous pensons qu'il s'agit de celui qui s'adapte le mieux à notre application aujourd'hui. Nous aurions pu mettre un bouton de rafraîchissement manuel des *lobbies*. Cependant, nous trouvions cela moins pratique car un joueur n'ayant pas rafraîchi ses lobbies pourrait cliquer sur "Rejoindre un *lobby*" et voir "*Lobby* plein" alors que ce n'est plus le cas. Notre façon de faire n'est pas *scalable* à un grand nombre de joueurs et dans l'éventuel cas où l'application aurait une communauté très large, nous opterions plutôt pour un *broadcast* par intervalle : par exemple, nous mettrons à jour les *lobbies* toutes les 2 secondes, au lieu de les mettre à jour à chaque demande de joueur. Pour améliorer notre application, la meilleure solution aurait été d'offrir au joueur deux options de jeu : un bouton "Rejoindre" qui serait public et permettrait de rejoindre un *lobby* aléatoire, et un système de *lobbies* comme nous le faisons actuellement qui serait dédié à des parties privées, entre amis. Nous avons mis environ 5 heures pour faire implémenter ce choix tel que nous le souhaitions.

Cinquièmement, nous avons ajouté un bouton "Prêt" dans notre application. Quand un joueur rejoint un *lobby*, il peut se mettre "Prêt". La partie ne commence que lorsque tous les joueurs de la partie sont prêts. Plutôt que de démarrer après avoir atteint le nombre de joueurs attendu dans un *lobby*, nous avons mis cette fonctionnalité en place pour laisser le temps aux joueurs de sélectionner leur couleur (qu'ils choisissent à chaque début de partie). Une fois que tous les joueurs sont prêts, un compte à rebours se lance. Nous avons fait ce choix pour ne pas surprendre des joueurs au démarrage de la partie. Ce choix a amené à prendre d'autres décisions comme le fait d'expulser un joueur qui ne se serait pas mis "Prêt" au bout de 30 secondes. Sans cette fonctionnalité, le *lobby* aurait été bloqué par des joueurs inactifs. Nous ne l'avions pas auparavant et l'ajouter a permis d'améliorer la jouabilité. Nous avons également ajouté la possibilité de quitter un *lobby* au cas où certains joueurs ne souhaitent plus attendre. Une amélioration pour notre application serait de permettre également aux joueurs s'étant mis "Prêt" de quitter le *lobby*. Pour l'instant, nous avons désactivé cette fonctionnalité pour ne pas créer de *bugs* (au cas où un joueur rejoigne à nouveau un *lobby* dans lequel il s'était mis "Prêt"). Nous avons mis environ 10 heures pour ajouter cette fonctionnalité, avec l'expulsion au bout de 30 secondes et l'adaptation de l'interface.

C. Choix côté serveur

Voici les choix que nous avons effectués côté serveur.

Premièrement, nous avons choisi de modulariser le code côté serveur en plusieurs fichiers, pour respecter la séparation des préoccupations et rendre notre projet plus maintenable. Nous avons donc un fichier `WebsocketServer.js` qui traite les paquets avec un `switch` et des `cases` qui sont gérés par des fonctions *handlers* provenant d'un même fichier `GameHandler.js`. Nous avons également un fichier séparé `db.js` qui s'occupe de la connexion avec MongoDB et les modèles

Mongoose sont séparés par collection (`GameModel.js` et `PlayerModel.js`) dans le répertoire `models/`. Enfin, les logiques de jeu (début, mise à jour, vérification des collisions, etc.) et de joueur (déplacement, couleur, position, etc.) se trouvent dans des classes `Game` et `Player` dédiées, pour les séparer du traitement des paquets. Nous avons mis environ 7 heures pour modulariser le code, qui ne l'était pas à l'origine.

Deuxièmement, nous avons décidé de stocker les parties en cours dans une `Map` (`gameId => game`). Nous avons choisi la `Map` plutôt que d'autres structures de données car elle permet un accès en recherche constant. Nous avions envisagé de remplacer cette `Map` par une `Map` (`connection => game`), afin d'éviter d'avoir à spécifier un `gameId` dans les requêtes : il aurait alors suffit de rentrer la connexion du joueur en clé pour accéder à la partie dans laquelle il se trouve, au lieu d'avoir à boucler sur les joueurs d'une partie pour vérifier la présence d'un joueur, ce qui est bien moins rapide. Cependant, nous n'avons pas fait ce changement car nous avions besoin des identifiants de partie pour rejoindre un *lobby* en particulier (lorsque l'on clique sur `joinGame()`, qui prend un identifiant en paramètre, notamment). Nous avons donc besoin d'un attribut de partie qui soit discriminant, unique et cela ne pouvait pas être le nom d'une partie puisqu'une partie rejouée a le même nom que la partie venant de se terminer. Ainsi, une amélioration possible de notre application serait d'avoir 2 `Maps` pour les parties en cours, à la fois (`gameId => game`) et (`connection => game`), en privilégiant l'utilisation de (`connection => game`) sauf si nous avons besoin d'un identifiant de partie. Pour mettre en place ce choix initial, nous avons mis environ 1 heure.

Troisièmement, nous stockons les joueurs connectés dans une `Map` (`name => connection`). Ici, nous avons également choisi une `Map` pour les raisons évoquées précédemment. Auparavant, nous stockions une `Map` (`playerId => connection`) mais nous nous sommes rendus compte que le nom d'utilisateur d'un joueur est également unique. Cela signifie qu'il est possible de l'utiliser pour différencier les joueurs et cela permet de ne pas avoir à faire transiter dans les paquets les identifiants de joueurs générés par Mongoose qui sont généralement bien plus longs que les noms d'utilisateurs. Nous avons mis environ 4 heures pour mettre en place ce choix et ensuite modifier la structure de la `Map`.

Dernièrement, nous avons fait en sorte que tous les *handlers* vérifient des erreurs potentielles dans les paquets reçus. Il s'agit d'une architecture défensive permettant de garantir au mieux la stabilité du jeu et d'éviter au maximum un crash du serveur. Nous avons mis environ 4 heures pour mettre en place ces vérifications sur l'ensemble des *handlers*.

D. Choix côté client

Voici les choix effectués côté client.

Premièrement, comme pour le côté serveur, nous avons modularisé le code côté client en le séparant par fonctionnalité (*lobby*, contrôles, classement, etc.). La modularisation aide à garder le code

maintenable. Nous avons également un fichier `global.js` qui garde les variables et fonctions globales à tous les fichiers : il s'agit notamment des fonctions d'affichage de message sur l'interface, on encore de la fonction qui permet d'envoyer des *websockets* au serveur. Nous évitons ainsi de réécrire un code identique en le centralisant en un même endroit. Nous aurions pu modulariser davantage en séparant la partie interface utilisateur de la partie "*handler*" pour chaque fonctionnalité. Cependant, le code étant assez court, nous avons préféré garder ces parties dans un même fichier. Cette modularisation nous a pris environ 5 heures.

Deuxièmement, nous prenons soin de vérifier les requêtes envoyées par le client au serveur. Par exemple, si le client est déjà prêt et qu'il appuie sur le bouton "Prêt", nous n'envoyons pas cette requête au serveur car elle est inutile, vu qu'un joueur ne peut pas enlever son statut "Prêt" une fois activé. De même pour les directions durant la partie, si le client appuie sur la même direction ou la direction opposée de sa direction actuelle, nous n'envoyons pas de requête au serveur. Ces vérifications permettent d'éviter un *spam* du client vers le serveur. L'implémentation de ces vérifications nous a pris environ 2 heures.

Troisièmement, le HTML de l'application est organisé en sections, où chaque section correspond à un écran de l'application. Nous avons donc notamment un écran de connexion, un écran de la liste des lobbies, un écran de partie, etc. En plus d'offrir un code clair, cette organisation permet de réaliser une *Single Page Application* facilement, à l'aide d'une fonction qui va fermer tous les écrans avant d'en ouvrir un. Implémenter cette structure initiale nous a pris environ 1 heure.

Quatrièmement, concernant l'utilisation de l'application sur mobile, nous avons fait le choix d'intégrer des boutons cliquables directement dans l'interface pour le contrôle des déplacements, plutôt que de mettre en place une fonctionnalité de type *swipe*. Ce choix s'explique par plusieurs raisons. Tout d'abord, les boutons offrent un contrôle plus précis et plus fiable dans un jeu nécessitant des changements de direction rapides et exacts, comme TRON, où une mauvaise interprétation d'un geste tactile peut entraîner une collision immédiate. De plus, la gestion des *swipes* peut varier fortement selon les appareils, la taille de l'écran ou la sensibilité tactile, ce qui aurait pu nuire à l'équité entre les joueurs. Nous avons mis environ 2 heures pour mettre en place les contrôles.

Enfin, nous avons choisi d'utiliser le SVG pour la grille de jeu. De par leur nature vectorielle, les SVGs permettent de ne pas avoir de flou en zoomant, contrairement aux pixels. Nous avons commencé par créer des éléments SVGs pour chaque trace de joueur au cours de la partie. Nous avons ensuite essayé d'opter pour des *polylines* où il nous suffisait d'ajouter des coordonnées dans l'attribut "points". En procédant ainsi, nous n'avions pas besoin d'ajouter des nouveaux nœuds au DOM en continu : il y avait seulement un *polyline* par joueur (donc 4 maximum), que l'on met à jour. Cependant, quand les parties devenaient un peu trop longues et qu'il y avait trop de points dans les *polylines*, il y avait des bugs de rendu visuel. Nous sommes donc revenus vers notre solution initiale tout en l'optimisant. Ce choix (et l'essai d'autres choix) nous a pris environ 4 heures.

III. Répartition des tâches

L'équipe de 5 personnes s'est organisée en deux sous-groupes principaux (serveur et client) avec un coordinateur transversal.

Adrien ZOFFRANIERI :

- Rendu SVG
- Structure HTML par écran (sections) pour la SPA (*Single Page Application*)
- Définition des styles CSS
- Développement des fonctions de gestion des lobbies
- Développement des contrôles utilisateur (clavier et boutons)
- Gestion des couleurs personnalisées par joueur

Aleksandra BASANGOVA :

- Coordination générale du projet incluant l'organisation des réunions et la planification des tâches
- Ecriture du rapport
- Préparation de la présentation pour la soutenance

Jean-Baptiste ARBAUT :

- idem qu'Adrien
- Recherche et filtrage de lobbies
- Ecriture du rapport

Marisela FLORES HERNANDEZ :

- Développement de la gestion des paquets et de la logique de jeu côté serveur et client (connexion du joueur, classement, mise à jour de l'état des lobbies, création des lobbies, possibilité de rejoindre des lobbies, quitter un lobby, fonctionnalité de « prêt » des joueurs, rejouer une partie, expulsion d'un joueur)
- Résolution de bugs
- Optimisation de l'envoi des requêtes côté client
- Réalisation de tests sur ordinateur
- Ecriture du README

Van-Louis TRAN :

- Développement de la gestion des paquets et de la logique de jeu côté serveur et client (authentification du joueur, classement, mise à jour de l'état des lobbies, création des lobbies,

possibilité de rejoindre des lobbies, quitter un lobby, fonctionnalité de “Prêt” des joueurs, rejouer une partie, expulsion d’un joueur)
- Conception de la base de données
- Spécification du format des paquets
- Classes Game et Player côté serveur
- *Merge* de branches de développement côté client
- Résolution de bugs
- Modularisation du code côté serveur et côté client
- Réalisation de tests sur téléphone
- Ecriture du rapport
- Ecriture du README

IV. Conclusion

Ce projet nous a permis d’en apprendre plus sur la communication via WebSocket, en particulier dans le cadre d’une application multijoueur. La mise en place d’un serveur nous a confronté à des problématiques concrètes de synchronisation, de gestion d’états partagés et de cohérence des données entre plusieurs clients connectés simultanément. Le développement de ce jeu TRON nous a également permis d’approfondir nos connaissances en architecture client-serveur et en conception logicielle. Les choix effectués, tels que la séparation claire des responsabilités entre le serveur et les clients, la modularisation du code ou encore l’utilisation de structures de données adaptées (`Map`, modèles MongoDB) ont contribué à la robustesse et à la maintenabilité de l’application. La conception adoptée, avec de nombreuses vérifications côté serveur et côté client, a renforcé la stabilité globale du système et limité les risques de comportements inattendus ou de triche.

L’utilisation de MongoDB nous a permis de découvrir les avantages d’une base de données NoSQL, notamment en termes de flexibilité du schéma et de facilité d’accès aux statistiques des joueurs. Le choix d’accepter une certaine redondance des données s’est avéré pertinent au regard des besoins fonctionnels de l’application, en particulier pour le classement et, éventuellement, pour l’historique de partie. Le recours au *framework* Cordova nous a permis de transformer une application web en une application mobile multiplateforme. Cette approche nous a sensibilisé aux contraintes spécifiques du développement mobile, telles que l’adaptation de l’interface aux différentes tailles d’écran et la gestion des interactions mobiles notamment via des boutons.

Sur le plan applicatif, ce projet nous a amené à réfléchir à l’équilibrage du *gameplay*, à l’expérience utilisateur et aux contraintes liées au multijoueur en ligne, comme la gestion des déconnexions, des lobbies ou encore des joueurs inactifs. Les fonctionnalités mises en place, telles que le système de “Prêt”, les *lobbies* dynamiques, le classement ou la possibilité de rejouer une partie, ont permis

d’obtenir une application fonctionnelle, cohérente et agréable à utiliser sur différents supports grâce à son caractère *responsive* et multi-plateforme.

Enfin, ce projet a constitué une expérience enrichissante de travail en équipe. La répartition des tâches, la coordination entre les sous-groupes client et serveur, ainsi que la gestion des *merges* et des tests sur différents supports, nous ont permis de développer des compétences organisationnelles et collaboratives dans un contexte de développement logiciel. Dans l’ensemble, ce projet nous a offert une vision concrète des enjeux liés à la réalisation d’une application web temps réel complète.