# Democratic Chess

Maxime Lovino Vincent Tournier Thomas Ibanez

14 septembre 2017

## 1 Introduction

Pour ce projet d'université d'été, nous avons décidé de créer un jeu d'échec multi-joueurs collaboratif. Le principe est que les deux équipes s'affrontent et à chaque tour, les joueurs de l'équipe qui joue sont amenés à voter pour le prochain mouvement à réaliser. Le mouvement qui aura le plus de votes ou, en cas d'égalité, le mouvement qui aura le plus de points (voir partie serveur) sera choisi et effectué.

### 1.1 Technologies choisies

Nous avons réalisé notre application web en 3 parties. Nous avons donc une partie serveur qui va gérer la partie et ses différents clients connectés et communiquer avec ceux-ci pour leur fournir des informations sur ce qu'il se passe. Ensuite nous avons également une partie commune qui est le moteur d'échecs en soi, celui-ci est utilisé aussi bien par le serveur que par les clients. Finalement la partie client s'occupera de gérer les interactions des joueurs avec le jeu et d'afficher le plateau de jeu et le reste de l'interface.

Nous nous sommes basés sur des websockets pour gérer la communication entre les clients et le serveur. Nous avons défini un protocole de communication en amont qui nous permettait d'envoyer des messages de différents types et de différentes portées. Dans chaque message envoyé, que ce soit par un client ou le serveur, nous avons un `id` spécifiant qui a envoyé le message et ensuite un objet `message` qui contient un type et des paramètres. Les types et la nature de leurs paramètres associés, ainsi que le reste des détails du protocole de communication, sont définis dans le fichier `communicationProtocol.md` situé à la racine de notre projet ainsi que dans le fichier javascript `common/communication.js`

Nous avons utilisé Node JS pour faire tourner notre serveur websocket, avec le module node `ws` pour la gestion du serveur websocket en tant que tel.

La partie commune a été exportée comme un module Node pour pouvoir être utilisée par le serveur et a été utilisée directement par le client.

La partie client a elle été réalisé en HTML 5 pour le chat et la liste des joueurs et Canvas pour ce qui concerne l'échiquier, ses pièces et l'affichage des votes.

### 1.2 Cahier des charges

Initialement, le but était de réaliser un jeu d'échec simple avec la possibilité de collaborer sous forme de votes pour les mouvements à réaliser. Nous avons d'abord donc préparé un moteur d'échecs de base, sans coups particuliers comme la promotion et le roque pendant que nous nous intéressions au fonctionnement des websockets. Ensuite, étant donné le temps restant et l'avancement de l'interface graphique et du serveur qui nous permettaient de tester le moteur autrement que dans un terminal, nous avons implémenté le reste les règles des échecs dans le moteur.

Le chat était également sensé être un chat basique dans l'équipe mais suite à des tests réalisés en postant une version en développement sur un forum pour que d'autres personnes jouent avec nous, nous nous sommes rendus compte qu'il fallait ajouter la possibilité de parler à tout le monde également. De façon générale, le feedback reçu par les utilisateurs nous a poussé à implémenter beaucoup de fonctionnalités supplémentaires par rapport à ce que nous avions prévu initialement.

### 1.3 Répartition du travail

## 2 Partie commune - Le moteur de jeu d'échec

Le moteur est un objet capable de conserver l'état d'une partie d'échec, proposer pour chaque pièce tous les mouvements possible, détecter un échec ou un échec-et-mat et imprimer le plateau de jeu de manière très primaire dans le chat.

### 2.1 Fonctionnalités de base

Notre première préoccupation en implémentant ce moteur était d'avoir un jeu capable de proposer tous les mouvements basiques, sans le roque ou la prise en passant, de détecter une mise en échec et un mat. Par la suite, du fait des divers test et de l'avance prise pendant le développement, nous avons pris le temps d'implémenter les règles les plus subtiles du jeu d'échec.

Lorsqu'un nouvel objet de type ChessMotor est créé, un plateau de base avec les blancs en haut est généré. Dès lors, la fonction getAllPossibleMove() retourne toutes les cases de destination possibles pour une case de départ donnée. Cette fonction est absolument centrale dans le moteur, car c'est elle qui permettra aux clients d'afficher les mouvements possibles pour une pièce, et au serveur de valider un vote envoyé par un client comme étant un mouvement légal.
Au départ, cette fonction ne gérait que les règles de base : déplacement selon la pièce, aucun saut de pièce sauf pour les cavaliers, prise uniquement possible sur des pièces de l'autre couleur, etc... Puis nous avons implémenté le roque, intégrant de ce fait aux objets représentants les cases du plateau un booléen capable de dire si la pièce avait déjà été bougée ou non. Puis nous avons implémenté une fonction de détection de l'échec. A chaque tour, le serveur vérifie que le roi adverse n'est pas mis en échec. Pour ça, le moteur récupère tous les déplacements possibles des pièces, et si l'une d'entre elle peut atteindre le roi adverse, il y a échec. Pour tester le mat, le déplacement de toutes les pièces adverses est testé. Si après l'un d'entre eux, le roi n'est plus en échec, il n'y a pas de mat.

### 2.2 Fonctionnalités supplémentaires

Ces deux fonctions implémentée, nous avons pu ajouter un élément essentiel à notre moteur : l'impossibilité d'un déplacement qui rendrait le roi vulnérable. Pour cela, un filtre est passé sur les mouvements possibles lors de chaque appel à getAllPossibleMove(), qui ne valide que les mouvements à la suite desquels le roi n'est pas en échec. Par ailleurs, il nous a été possible d'empêcher le roque lorsque le roi, sa case d'arrivée ou la case traversée est en échec, respectant de ce fait toutes les contraintes du roque.

Pour finir, la prise en passant a été implémentée. Celle-ci a necessité l'utilisation d'une nouvelle valeur booléenne, car il fallait pouvoir identifier un pion qui vient d'effectuer une avancé de deux cases. En effet, c'est uniquement au tour suivant un tel mouvement qu'une prise en passant est possible. C'est donc la fonctione move(), qui réalise un mouvement de pièce sur l'échiquier, qui s'occupe de mettre à jour ces booléens.

### 2.3 Questions

Quest ce qui vous a positivement surpris ?
- La rapidité et au final, la simplicité que demandait le développement d'un moteur de jeu d'échec. Les conditions de déplacement étant très adaptée au langage informatique, cela ne m'a pas demandé beaucoup d'effort de développement, contrairement à ce que je pensais. Par ailleurs, Javascript est au final un langage très complet et adapté aux applications en ligne.
Quest ce qui vous a pris plus de temps ?
- Le filtre empêchant le roi de se suicider. Rendre illégal un coup mettant en danger le roi est au final relativement compliqué, car il faut pour cela vérifier si après ce coup le roi est en échec, or pour vérifier ceci on appelle getAllPossibleMoves sur chaque pièce, qui va à nouveau filtrer ses résultats pour vérifier que le roi n'est pas en échec, etc... Cet appel récursif contrariant a été particulièrement compliqué à gérer.
Quel est le pire bug que vous avez eu ?
- Lorsque le roi se fait prendre sans que le mat n'ait été détecté ou qu'un pion arrive au sommet de l'échiquier alors que la promotion n'est pas implémentée, le moteur crashe tout bonnement et simplement. C'était assez intéressant de constater l'origine de ces bugs. Mais le pire bug est aussi celui qui m'a demandé le plus de temps : l'appel récursif lors du test d'échec au roi décrit ci-dessus.
Au final si vous deviez recommencer que changeriez-vous ?
- Faire directement une version orientée objet, car j'ai commencé par produire un code semblable à du C avant de tout refactorer, ce qui m'a pris beaucoup de temps pour rien. Par ailleurs, je définierais mieux les objets représentants des pièces, afin que seuls ceux ayants besoin de tel booléen ou tel autre l'ait, et que chaque pièce possède une fonction propre capable de donner ses propres possibilités de mouvement, au lieu d'avoir une énorme fonction avec un switch qui n'en finit plus...

## 3 Partie serveur - Le serveur websocket

### 3.1 La gestion des clients

Pour la partie serveur, la gestion des différents clients est la partie qui a demandé le plus de reflexion initialement. En effet, dans le cas du serveur websocket, nous avons un callback à écrire qui va être appelé lors de la connexion d'un nouveau client au socket et un autre lors de la réception d'un message. Or, dans le callback de connexion est présent le socket en paramètre, alors que dans le callback de message, il y

a juste le contenu du message. Il nous fallait donc un moyen d'identifier de qui provenaient les messages et pouvoir répondre à une personne spécifiquement. Pour ce faire, nous avons décidé de générer des uuid pour chaque client que nous avons transmis à la connexion et qui nous sont envoyés avec chaque message. Au niveau de la structure d'un client côté serveur, nous avons un objet `client` qui contient l'id, le socket et un objet `player` pour chaque client. Dans l'objet `player`, nous avons les informations "publiques" du joueur, comme son nom ou son score. Cela nous permet par exemple d'avoir une liste de client avec les infos utiles côté serveur, mais de transmettre une liste des joueurs (en utilisant map sur le tableau des clients) aux différents joueurs connectés.

Ces clients sont stockés dans une table associative qui stocke les clients avec leur id comme clé (pour un accès direct) et dans un tableau des clients pour pouvoir utiliser toutes les méthodes ES6 sur les array pour filtrer etc.

Au niveau de l'envoi de messages aux clients, deux fonctions ont été réalisées qui permettent de broadcaster un message à toute une équipe ou à tous les clients.

## 3.2 Le callback de gestion de message

Pour ce qui est du callback à la réception d'un message, tout d'abord nous vérifions que le message contient l'id d'un client qui est valide et ensuite selon le type de message, nous effectuons différentes actions. Les messages que le serveur peut recevoir sont de 3 types : l'envoi du nom d'un joueur, un vote pour un mouvement ou un message de chat.

Dans le cas de l'envoi du nom d'un joueur, nous allons enregistrer le joueur dans la partie et lui envoyer toutes les infos actuelles sur la partie, comme par exemple l'état du jeu d'échecs, l'équipe qui lui a été assignée etc. Nous allons aussi envoyer une nouvelle liste de joueurs à tous les joueurs.

Dans le cas d'un vote pour un mouvement, nous allons stocker ce vote dans la liste des votes (uniquement si le vote provient de l'équipe qui joue actuellement) et pour ce faire nous avons une table associative qui utilise comme clé l'objet vote en JSON et comme valeur la liste des joueurs ayant voté pour celui-ci.

## 3.3 Gestion de la partie et des timings

Le timing de la partie est géré par des timeouts qui vont appeler la fonction de choix des votes à la fin d'un tour et refaire un timeout au moment du changement d'équipe. En cas de votes de tout le monde avant la fin du tour, nous effaçons ce timeout et appelons la fonction de choix directement. Autrement dit, à chaque fois qu'un nouveau vote est ajoutée ou qu'un joueur quitte l'équipe en cours, nous appelons la fonction `canWeChoose` qui va vérifier si nous pouvons déjà procéder au choix des votes.

Un deuxième timeout est également présent pour envoyer un message aux utilisateurs quand il faut attendre de nouveaux joueurs pour commencer une partie.

Finalement, la gestion des fins de jeux est faite de cette façon : si au moins toute une équipe part, le jeu est remis à zéro ; si une équipe ne vote pas pendant un tour, l'équipe est effacée et le jeu est remis à zéro ; en cas d'échec et mat, le jeu est remis à zéro après 5 secondes et annonce du résultat.

### 3.4 Comptage des votes

Le choix du vote est effectuée dans la fonction `sortVotes` qui va trier les clés des votes en fonction tout d'abord du nombre de joueurs ayant voté et ensuite, en cas d'égalité, en fonction du poids de ces votes. En effet, chaque joueur accumule des points lors de la partie en fonction de son abilité à proposer rapidement des votes qui seront choisis. Les points ne sont accumulés que si son vote est choisi et avait été voté par au moins un autre joueur. Les points sont attribués de façon décroissante selon l'ordre d'arrivée. Imaginons que 3 personnes ont voté pour un vote qui a été selectionné. Le premier va recevoir 3 points, le deuxième 2 points et le dernier 1 point.

### 3.5 Questions

Quest ce qui vous a positivement surpris ?
- La facilité de déploiement de JavaScript côté serveur avec Node JS et le nombre de modules disponibles pour faire facilement beaucoup de choses fastidieuses et pas forcément intéressantes, comme par exemple un package de log.
Quest ce qui vous a pris plus de temps ?
- La compréhension du fonctionnement des websockets et la gestion des utilisateurs et leur identification lors de l'envoi de messages. La documentation du package que j'utilisais n'était pas forcément très claire à ce sujet.
Quel est le pire bug que vous avez eu ?
- Le jour de la dernière démo, nous avions posté une version en dev sur un forum et avions récolté des retours intéressants et nous nous sommes lancés sur l'ajout d'une multitude de fonctionnalités à quelques heures de la démo finale. Les plus gros changements étaient côté serveur et suite à ces changements, un bug a été introduit qui, parfois, renvoyait tous les joueurs de la partie sans raison valable. Dans le stress, j'ai réussi à trouver ce qui avait introduit ce bug et le corriger peu avant la démo.
Au final si vous deviez recommencer que changeriez-vous ?
- Je pense que du point de vue déploiement, on changerait un peu la modularisation du projet pour ne pas avoir deux applications totalement distinctes (serveur et client) mais plutôt que le serveur serve aussi de serveur web pour accéder au client de jeu (au lieu de devoir hoster le site avec un serveur apache en plus), tout serait donc intégré à l'app node, le serveur pour les websockets et le serveur pour accéder au client (avec Express par exemple). Aussi, je pense que si nous étions partis sur l'option socket.io, bien que moins standard, nous aurions eu plus de documentation concernant son utilisation.

## 4 Partie client - Le frontend et l'interaction avec le jeu

### 4.1 Interface Graphique Basique

L'interface graphique à été inspirée par celle du site lichess.org. Nous avons a gauche une chatbox ainsi que nos informations. Au centre un canvas sur lequel sera dessiné l'échiquier et a droite la liste des joueurs au dessus du timer.
Un des défis pour la mise en place de cette interface était de reprendre le langage CSS que nous n'avions plus utilisé depuis bien longtemps.
Une fois cette disposition mise en place, il a fallu créer le rendu de l'échiquier et des pieces sur le canvas. Etant donné que nous avions bien détérminé le protocole réseau et la structure des données avant de se lancer dans le développement cette partie n'as pas présenté de difficultés particulières et s'est terminée

plutôt vite.
La suite a été d'ajouter la mise en avant des cases sur lesquels un mouvement était possible. Cette étape a été ralentie par un malentendu entre le moteur et le frontend. Pour Vincent la coordonnée x représentait l'axe verticale et la coordonnée y l'axe horizontal (ordre de parcours de deux boucles imbriquées) alors que pour Thomas c'était l'inverse (coordonnées cartésiennes).
Le but pour le client est de faire un maximum de calculs en local (notemment les mouvements possibles, les échecs, la detection de promotion, etc...) afin de minimiser le travail du serveur qui n'aura qu'a valider les données.

## 4.2 Communications avec le serveur

Une fois l'interface basique réalisée il a fallu implémenter les communications avec le serveur afin de pouvoir tester le moteur à plusieurs joueurs. J'ai donc commencé par implémenter les messages de bases tels que la proposition d'un déplacement, le chat et l'execution d'un déplacement. Ceci nous a permis de débugger plus facilement le moteur avec un retour visuel permettant de bien visualiser ce qu'il se passe.
La suite des commandes s'est ajoutée sans soucis bien que le nombre de messages ai augementé avec le temps pour permettre un plus grande souplesse dans la gestion de la partie. Au final il y a 17 messages envoyable par le serveur et 3 par le client.

## 4.3 Interface Graphique Finale

Une fois les fonctionnalités réseau implémentées il a fallu revoir l'interface graphique afin de livrer une version du jeu plus agréable a utiliser. Celle-ci fut influencée par les tests réalisés avec des inconnus sur internet et leurs retours.
Parmis les changements il y a eu :

- Les flèches de déplacement qui grandissent en fonction des votes
- Le choix de chaque joueurs qui s'affiche sur la liste
- Les scores de chaque joueurs qui s'affichent sur la liste
- La case du roi en échec qui s'illumine en rouge
- Le timer qui s'affiche en rouge si le temps est inférieur ou égal à 10 secondes
- Une notification si le joueur doit voter et qu'il ne regarde pas le jeu

ainsi qu'un écran d'accueil avec les instructions des bases pour jouer.
La dernière étape a été d'ajouter un menu pour gérer la promotions si un joueur veut enmener son pion jusqu'a bout de l'échiquier.

## 4.4 Questions

Quest ce qui vous a positivement surpris ?
- Sans doute la satisfaction de développer le frontend, on arrive rapidement a avoir un résultat visuel ce qui est motivant pour continuer.
Quest ce qui vous a pris plus de temps ?
- Faire une disposition a 3 colonnes en CSS (c'est triste mais c'est vrai) au final j'ai du aller regarder sur stackoverflow comment faire
Quel est le pire bug que vous avez eu ?
- L'inversion x/y avec le moteur, c'était très perturbant
Au final si vous deviez recommencer que changeriez-vous ?

- Je ferait directement de l'orienté objet car changer les déclarations de fonctions, rajouter "this." devant les noms de variable, etc... fut un travail long et peu intéressant.

# 5 Conclusion

En conclusion, nous avons fortement apprécié travailler sur ce projet, et nous avons envie de continuer à travailler dessus dans le but de le publier. Pour ce faire, nous avons encore une longue liste de fonctionnalités que nous aimerions implémenter :

- Une rotation du jeu d'échecs en fonction de l'équipe pour laquelle on joue de façon à voir toujours son équipe en bas. Ceci est la fonctionnalité la plus compliquée à implémenter car elle demande de revoir le méchanisme de détection des clicks sur le jeu, ainsi que l'affichage des votes etc.
- Un affichage un peu plus gratifiant à la fin de la partie, un type de message RESULT est déjà prévu pour ça du côté serveur, mais nous n'avons pas eu le temps de l'implémenter.
- Un mode spectateur, qui permettrait à quelqu'un de suivre la partie tout en voyant les votes de toutes les équipes et tous les messages de chat.
- Un kick d'un joueur lorsqu'il ne vote pas pendant plusieurs tours de suite, ceci permettrait de fluidifier le jeu en évitant qu'une équipe attende toujours sur un joueur qui ne vote jamais.

## 5.1 Bugs connus à l'heure actuelle

Les notifications web implémentées dans le client, bien que suivant le standard de notifications web HTML5 qui est supporté par tous les navigateurs actuels (sauf IE, mais est-ce vraiment actuel ?) ne fonctionnent que sur Firefox pour l'instant et nous n'avons pas réussi à les afficher sur Chrome par exemple.

## 5.2 Ce qu'on devrait faire différemment

Il faudrait revoir comme dit dans la partie serveur une partie de la modularisation dans le cas d'une mise en production car actuellement le déploiement du serveur websocket et d'un serveur apache dédié pour le client uniquement demande trop de travail. Il faudrait tout intégrer dans node en utilisant express pour router les requêtes HTTP tout en écoutant toujours le websocket également.