ENSEIRB-MATMECA

Filière informatique : Première année

Rapport de Projet S6 :

Carcassonne

BURE / CASTERES / KUHN / MARANNE
Encadrant : Steve Nguyen

Mai 2018

Table des matières

1 Introduction

Le jeu Carcassonne est un jeu de société dans lequel les joueurs gagnent des points en construisant des édifices (routes, châteaux, monastères ...). Leur objectif est de placer leurs cartes de manière à obtenir le plus de points, de la manière la plus rentable possible. Pour cela, il faut terminer ses constructions ou éventuellement voler celles de ses voisins.

Ce projet a pour but d'implémenter le jeu Carcassonne pour pouvoir faire jouer plusieurs personnes (dont les stratégies seront éventuellement différentes).

1.1 Travail en groupe

Nous avons travaillé pendant ces sept semaines en divisant les taches entre nous : travail sur les algorithmes, tests, rédaction du compte rendu... Le travail a été réalisé en dehors des heures de projets prévues lorsque nous rencontrions des problèmes. Chaque séance de travail commençait par un point sur nos avancements et nos objectifs.

1.2 Problématique

L'objectif est de réaliser un programme respectant l'ensemble des règles du jeu, avec des clients indépendants du serveur et des représentations adaptées pour nos données.

1.3 Plan

Dans un premier temps, nous analyserons le cadre de départ et l'organisation des fichiers. Ensuite, nous présenterons la représentation des objets et le fonctionnement client/serveur. Nous décrirons ensuite les tests, les problèmes rencontrés et terminerons par les évolutions possibles.

1.4 Outils utilisés

Nous avons utilisé le système de version Subversion ce qui nous a permis de conserver chacune des versions de notre code et ainsi gérer nos erreurs tout au long du projet.

Nous avons utilisé les éditeurs de texte Emacs et Atome pour notre code ainsi que les outils valgrind et gdb pour le débuggage.

Nous avons utilisé Doxygen pour la documentation.

Ce rapport est rédigé en LaTeX.

2 Analyse des problèmes liés au cadre de départ

Pour simuler une connexion entre le serveur et les clients, une interface commune a été implémentée afin que le serveur puisse envoyer les informations aux clients et qu'il puisse récupérer leur action. Ainsi, les fonctions utilisées sont identiques pour chaque client. De plus, les clients sont compilés sous la forme de bibliothèques partagées afin d'être chargés dynamiquement par le serveur.

Les clients sont complètement indépendants du serveur. Un client peut uniquement faire ses calculs avec les données qu'il a reçu du serveur. Ainsi, chaque client doit gérer sa propre version du plateau de jeu. Pour cela, il aura besoin des mêmes fonctions que le serveur. Il y aura donc un problème sur le découpage du code.

Un autre problème se pose sur la manipulation des cartes. Le serveur doit avoir en mémoire les cartes jouées précédemment par chaque joueur et les cartes contenues dans la pioche. Il est alors nécessaire de pouvoir ajouter, rechercher et supprimer des cartes rapidement.

3 Organisation des fichiers

L'implémentation de ce jeu nécessite deux parties :

— Un serveur : son rôle est d'initialiser le plateau de jeu ainsi que l'ensemble des clients. Il devra demander à chaque joueur comment il souhaite jouer la carte donnée en lui envoyant les coups précédents après avoir vérifié que la carte est jouable.

— Un ou plusieurs clients : chaque client possède sa stratégie de jeu et recevra , au début de chaque tour, des informations sur le plateau de jeu. Il devra ainsi jouer des coups valides grâce aux données qu'il a reçues et stockées.

Pour au mieux faire jouer les clients et le serveur, nous avons implémenté plusieurs structures :

— une structure **moves** , qui permet de lister les différents coups joués par les joueurs.

— un structure **player**, qui liste les joueurs.

— un TAD **deck**, qui s'occupe de la gestion des différentes cartes.

— une structure **card** pour représenter correctement les cartes.

Les différents fichiers contenant ces TAD et structures sont : deck.c, deck.h, board.c, board.h, card.c, card.h, player.c, player.h, server.c.

4 Représentation des objets

4.1 Card

La partie *card* est constituée de card.c et card.h. On y décrit la structure card composée d'un tableau d'entiers représentant le découpage de la carte, comme illustré ci-dessous et d'un booléen indiquant la présence ou non d'un bouclier sur la carte.

```
1  struct card {
2    int sides[NB_SIDES_CARD];
3    int shield;
4
5    /* the index of the array "sides" represents the position
6    of the point like shown below (like the enum "place")
7         3   2   1
8        ___________
9       |  *  *  *  |
10     4 |*         *| 12
11     5 |*   13    *| 11
12     6 |*         *| 10
13      |  *  *  *  |
14      |___________|
15         7  8  9
16
17         The array can be fit with ROAD, PLAIN, or CITY */
```

L'énumération *side* représente les différents types d'éléments existants sur une carte : route, champ, abbey et ville.

```
1  enum side{
2    ROAD,
3    PLAIN,
4    CITY,
5    ABBEY,
6    LAST_SIDE
7  };
```

Associés à cette structure, 3 fonctions ont été implémentées :

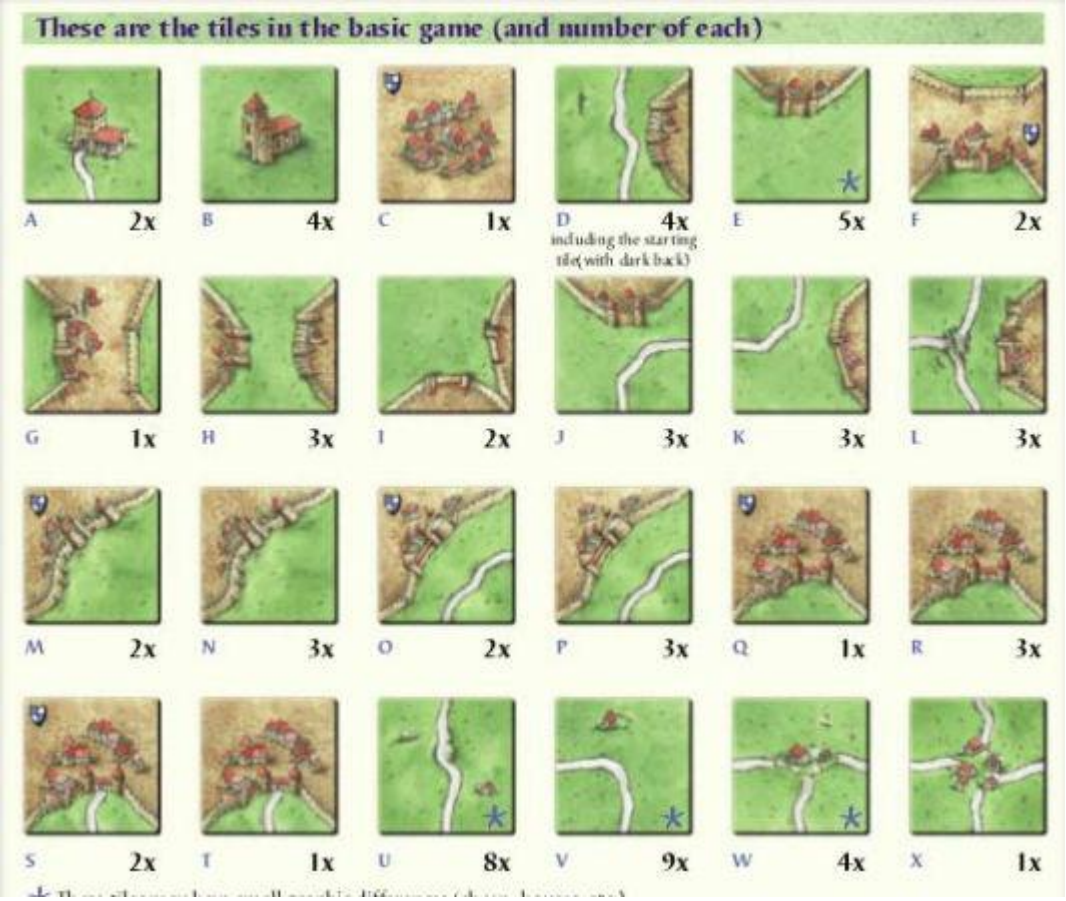
— struct card use_card : cette fonction, en fonction de la carte passée en paramètre, initialisera les 13 paramètres de la carte.

— struct card* create_card : initialise un tableau de struct card avec les 25 types de cartes différentes.

— free_card : qui est là pour libérer la mémoire que nous avons alloué.

.

La liste de cartes et ses paramètres sont fournis dans le fichier 'interface.h'.



4.2 Players

La partie *player* est constituée de player.c et player.h. Elle consiste en l'implémentation des joueurs grâce à la structure suivante :

```
1 struct player {
2   int id;       // id of the player
3   int meep;     // meeples available
4   int points;   // points earned by the player
5   void* handle; // the handle of the player
6 };
```

L'entier id représente l'indice du joueur, meeple représente le nombre de pions que possède actuellement le joueur, points représente les points du joueur et handle stocke le retour de la fonction "dlopen()" afin de pouvoir gérer les librairies externes des clients.

Les seules fonctions crées pour gérer les joueurs sont :

— create_player qui créé un joueur à l'aide d'une allocation dynamique et initialise ses champs.

— free_player qui libère l'espace alloué dynamiquement.

4.3 Board

Le plateau de jeu est représenté par une matrice de taille 200 X 200, contenant des tableaux de taille 3. Ces tableaux contiennent trois entiers représentant : l'indice de la carte, son orientation, la présence où non de pion sur cette carte.

'board.c' est un fichier découpé en plusieurs parties.

Tout d'abord, une partie contienant de nombreuses fonctions auxiliaires nous permettant de gérer les orientations et coordonnées de nos cartes :

- posit : qui renvoie les coordonnées de la carte demandées par rapport à celle donnée. (Exemple : si on veut la carte située au nord de celle en 3,4 il nous renvoie 3,5.)
- cardin : qui retourne l'entier associé à la coordonnée demandée sur la carte. (Exemple : North->2.)
- side : qui renvoie l'entier associé au côté demandé en fonction de l'orientation de la carte.
- opposite_side : qui renvoie l'ID du côté opposé à celui envoyé.

Dans un deuxième temps, sont implémentées des fonctions nous permettant d'utiliser le board :

- init_board : qui créé et retourne la matrice à l'aide de malloc.
- start_game : qui place la première carte du plateau en [100 ;100].
- playable : qui indique si la carte passée en parametre peut etre placé sur le plateau à ce moment de la partie.
- valid_move : cette fonction vérifie que les quatre bords de la carte coïncident avec les cartes déjà placées sur le board. Elle vérifie également que le joueur n'essaie pas de poser une carte dans le vide.
- place_move : qui pose définitivement une carte sur le board.
- fill_board_player : cette fonction utilise place_move pour placer sur le plateau l'ensemble des move contenus dans une liste.
- free_board : qui libèrera la mémoire allouée dynamiquement.
- display_board : qui a été créée pour réaliser une interface graphique très sommaire afin de pouvoir visualiser un minimum l'évolution du jeu.

Deux fonctions supplémentaires ont été codées :

- convert_position_in
- convert_position_out

Elles ont été créées suite à l'échange entre groupes des clients et des serveurs de notre salle de projet. En effet, au début du projet, nous avons codé de manière torique la grille. Dans un second temps, nous avons codé les positions à la manière d'une matrice (voir figure 1), tandis que certains autre groupes sont partis sur un codage de positions de type grille d'unsigned int. Cette différence de codage créait des soucis de placement lorsqu'un coup était joué (des cartes étaient placées au mauvais endroit alors qu'elles auraient pu être jouées de manière valide).

'convert_position_in' sert à transformer les coordonnées fournies par les autres clients en coordonnées valables pour notre serveur, et inversement, 'convert_position_out' transforme les coordonnées de notre client en coordonnées valides pour un serveur externe.

4.4 Moves

Chaque coup est représenté par la structure donnée dans le fichier interface.h :

```
1 struct move {
2   enum action check;     // validation of the move by the server
3   unsigned int player;   // player issueing the move
4   enum card_id card;     // card played
5   struct position onto;  // position on the board
6   enum direction dir;    // direction of the north of the card
7   enum place place;      // place of the meeple on the card
8 };
```

A chaque coup joué par un client, le serveur modifie son plateau puis stocke le coup dans une liste. Cette liste de coups est transmise aux joueurs lors de leur tour pour qu'ils puissent représenter le plateau de leur côté en ajoutant les coups joués.

4.5 TAD deck

La partie *deck* est constituée de deck.c et deck.h. Elle consiste en l'implémentation de la pile de cartes sous forme d'un tableau géré par des pointeurs.

Pour cela, on a créé la structure suivante :

```
1 struct deck {
2   int * deck;           // points the array of cards
3   int * head;           // pointer to the next card
4   int nb_cards_left;    // number of cards left in the deck
5   int nbCards;          // total of cards
6 };
```

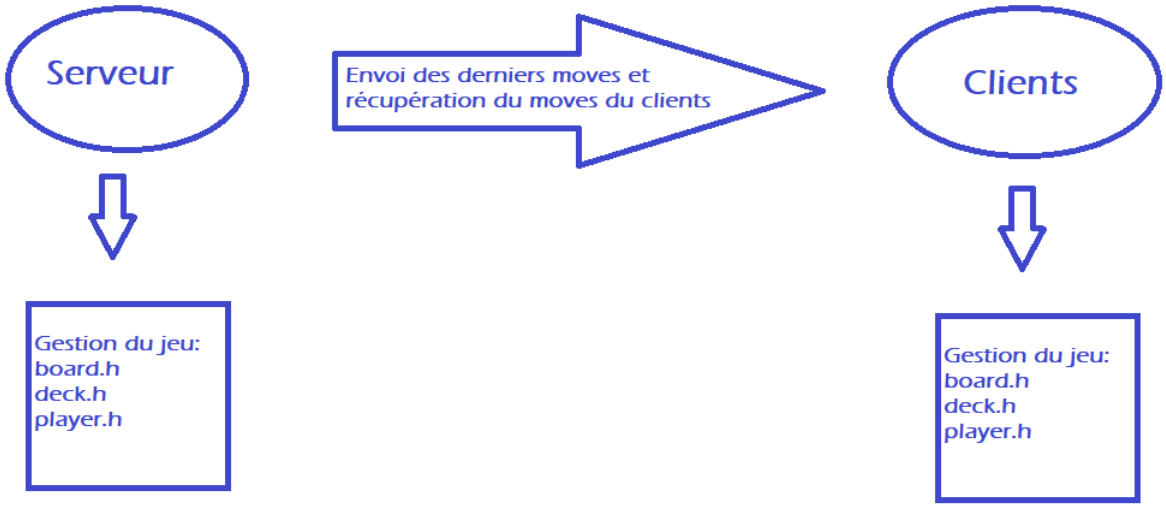
Les pointeurs *head* et *tail* pointent respectivement sur le début et la fin de la liste. Le pointeur *deck* pointe sur le début du tableau. L'entier *nbCards* représente le nombre de cartes au début de la partie. Le tableau cards_repartition donne le nombre total pour chaque carte différente, les indices correspondant à l'énumération donnée dans interface.h. Cette implémentation simple est suffisante pour la gestion de la pioche.

Les fonctions inhérentes au deck sont les fonctions permettant de créer un deck vide, de l'initialiser, de le remplir, de le libérer et de piocher.

5 Clients/Serveur

Notre programme repose sur deux parties distinctes : les clients, qui regroupent un ensemble d'algorithmes sous formes de bibliothèques dynamiques, et le serveur, qui organise une partie.

Malgré tout, plusieurs fonctions des fichiers 'board' ou 'card' ou même 'deck' sont communes aux deux cotés et sont placés dans ses fichiers pour découper le code et éviter une duplication des fonctions.



5.1 Les clients

Les clients sont sous forme de bibliothèques dynamiques. Leur interface est commune et donnée dans le fichier *interface.h* afin de faciliter l'échange des clients entre groupes sans trop de problèmes de compatibilité.

De plus, il est primordial d'avoir des clients avec des stratégies de jeu différentes. Cela permet de varier l'évolution du jeu au cours de la partie. Pour cela, ils auront une base commune (les fonctions des fichiers décrits précédemment).

Fonctionnalités communes à tous les clients

Les fonctions principales du programme introduites dans le fichier d'en-tête commun sont :

— char const* get_player_name

— void initialize

— struct move play

— void finalize

'void initialize' permet d'initialiser la structure board stockant les différentes informations du jeu. Elle permet l'allocation en mémoire de la structure de jeu.

Par opposition, la fonction finalize permet de libérer la mémoire de l'ensemble des allocations faites par chaque joueur.

La dernière fonction play est la fonction gérant la stratégie du joueur. De plus, dans c'est dans cette fonction que seront analysés les mouvement précédents de tous les joueurs, communiqués par le serveur. Lorsque le serveur fait appel à la fonction play d'un joueur, il passe en argument l'ensemble des coups joués en commençant par le dernier coup joué par ce joueur. Chaque joueur met alors à jour sa version du jeu en intégrant chacune des informations à son plateau. Ces informations n'ont pas besoin d'être vérifiées par le client car le serveur vérifie en amont l'action et indique si l'action est invalide.

Implémentation de la stratégie

Les clients communiquent leurs actions par le biais de la fonction 'play' appelée par le serveur pour lui retourner une structure move. Celle-ci est composée du nom de la carte, de l'action, du joueur concerné, de la position où elle est jouée et de la direction de la carte. Le rôle de la fonction principale du client est donc de manipuler cette structure.

Ainsi, après avoir initialisé la structure et analysé les mouvements des précédentes actions, il faut vérifier que le joueur réalise un coup qui lui est permis et donc éviter de se faire éliminer par le serveur.

Ensuite, le joueur suit la stratégie qui lui est fournie : il s'agit de jouer naïvement. Le but de la stratégie naïve est de parcourir le plateau jusqu'à trouver un coup valide et poser la carte à cet emplacement là.

A la fin de l'algorithme, il faut ainsi retourner la structure move mais aussi libérer la mémoire .

5.2 Le serveur

Une fois chaque structure initialisée, une boucle while s'occupe de faire joueur les joueurs jusqu'à la fin d'une partie. Une partie est terminée lorsque toutes les cartes ont été posées ou que tous les joueurs ont été exclu.

Pour chaque joueur, le serveur récupère le coup proposé dans $'move\_to\_check'$ et vérifie sa validité grace à $'valid\_move$. Le coup est ensuite stocké dans le tableau de coup.
Le serveur envoie ensuite tous les coups joués par les joueurs pendant le tour à chaque client.

5.2.1 Initialisation

Partie

Le serveur a donc accès à l'ensemble des paramètres de la partie ainsi que la totalité des cartes pouvant être jouées. Il a alors la possibilité de générer le plateau de jeu.
Il contient la boucle principale du jeu, est chargé d'organiser une partie et d'exclure les joueurs qui envoient des coups invalides. Il s'occupe aussi de charger les bibliothèques dynamiques.

En premier lieu, il créé un tableau de joueurs et les initialise. Il créé ensuite également un tableau de 'struct moves' et un tableau de cartes. Il initialise ensuite la pioche et le plateau en créant 'board' puis en plaçant la première carte au centre du plateau.

Clients

Maintenant que le serveur est initialisé, le serveur doit charger dynamiquement les clients puis les initialiser.

L'ensemble des bibliothèques partagées représentant les clients est passé en paramètre de la ligne de commande. Le serveur doit charger les clients un par un et charger l'ensemble des symboles qu'il souhaite utiliser grâce à la librairie "Dynamically Loaded". Pour pouvoir accéder à l'ensemble des clients, le serveur utilise un tableau de struct client. Cette structure contient le "handle" renvoyé par dlopen et les adresses pointant vers les symboles chargés en mémoire renvoyées par dlsym.
Un joueur est éliminé dès qu'il joue un coup invalide.

5.2.2 Manches

A chaque début de manche, le serveur établit les clients.

Il peut ensuite exécuter la fonction initialize pour chaque client.
Durant la manche, les clients jouent tour par tour. A chaque coup, le serveur doit vérifier la validité du coup . Il faut vérifier que les chemins liens créés entre les cartes sont bons (les côtés sont compatibles) et qu'elle sera bien reliée au départ du plateau. La fonction valid_move permet de faire cette vérification. :

Une fois toutes les cartes posées (et donc le jeu terminé), il est nécessaire de libérer le plateau ainsi que les joueurs . Pour cela, le serveur appelle les fonctions finalize qui permet de libérer la mémoire des clients et fait aussi appel à free_board qui permet de libérer le plateau..

6 Présentation des tests

Avant de tester les fonctions implémentant le plateau, il faut définir celui-ci. En effet, tester un plateau déjà connu sur les fonctions permet d'avoir une prévision des résultats attendus, puis de comparer les résultats obtenus par la fonction avec ceux que l'on devait avoir.

Après initialisation, le plateau de jeu est utilisé pour tester chaque fonction. Pour cela, on teste des assertions qui correspondent aux données du plateau et donc qui retournent une réponse vraie, comme certaines qui sont fausses et donc qui revoient un assert faux.

Pour tester chaque fonction au mieux, les tests effectués concernent d'abord des cas généraux, puis couvrent les cas particuliers (possibles ou non en fonction des données entrées) lors des tests par `assert`.

Si nécessaire, d'autres plateaux, en plus du plateau défini initialement, peuvent être définis puis utilisés (exemple : utilisation d'un plateau avec une case vide, puis recherche d'une case qui n'est pas dans le plateau, etc...) .

Dans un premier temps, ce sont les fonctions de la pile de cartes qui ont été testées (fichier 'test_deck'), avant que ce ne soient celles autour des cartes (fichier 'test_card'), et finalement celles du plateau (fichier 'test_board')

7 Problèmes rencontrés

Pendant le projet, nous avons rencontrés quelques problèmes de mémoire suite à des non libérations de programme que nous avons pu résoudre grâce à Valgrind.

Le deuxième soucis qui s'est présenté à nous (et l'un des soucis majeurs) a été que, dans les dernières semaines, nous avons voulu tester nos clients/serveurs avec ceux des autres groupes. Suite à ces tests, nous nous sommes confrontés à des problèmes de compatibilité entre nos entités et celles des autres. En effet, lors de l'utilisation d'un client autre, des coups non valides sont intervenus alors que le jeu continuait son cours. En effet, est survenu un soucis de compatibilité entre notre grille et celle de nos camarades, et nous avons du modifier le sens de rotation de nos cartes. De plus, lorsqu'un client joue mal une carte, il est censé ne plus pouvoir jouer.

Ces problèmes ne sont pas encore tous résolus à l'heure actuelle : les grilles ne correspondent pas encore bien, ce qui engendre des valid_move incorrects, malgré l'écriture de deux fonctions pour palier ça :

— convert_position_in, qui converti les positions du client en des positions compatibles avec notre grille

— convert_position_out, qui converti les positions de notre client en des positions compatibles avec un serveur externe.

8 Évolutions possibles

8.1 Utilisation de graphe

Nous avons prévu de représenter le plateau de jeu de deux façons distinctes. Une partie physique déjà implémentée sous forme de matrice, et une partie codée avec des graphes afin de pouvoir relier les différentes parties de chaque carte entre elles, dans le but de pouvoir placer des pions suivant selon les règles et ainsi pourvoir compter les points.

Chaque carte serait un noeud de graphe, et nous aurions autant de graphes que de structures différentes. Par exemple une route serait représentée par la succession des noeuds de chaque cartes qui la compose. Nous aurons alors des graphes indépendants pour chaque type de structure (un graphe pour un château, un graphe pour une route etc...). La lecture de ces graphes nous permettra : coté client, d'élaborer une stratégie et coté serveur, de compter les points.

8.2 Compter des points

Partie pas encore codée : Pour compter des points, le serveur va devoir parcourir notre graphe. Dès lors qu'une entité est finie (elle sera alors délimitée par des "frontières"), il suffira de sommer le poids des arcs reliant les différents noeuds du graphe.

Compter les points nécessite de placer des pions sur dans le jeu, or cette fonction n'est pas implémentée.

```
1 Compter_points()
2
3     pour chaque partie du graphe:
4         parcourir le graphe
5         compter le nombre de pions de chaque joueurs
6         attribuer les points // en fonction du nombre de pions et du type de graphe
```

8.3 Complexité

Enfin, une amélioration importante pour notre code serait de travailler sur les complexités de nos codes. Notre jeu utilise une quantité importante de mémoire qui pourrait être optimisée.

9 Conclusion

Grâce à ce projet, nous avons pu mettre en oeuvre les différentes méthodes vues en cours de programmation impérative 2 (gdb, svn, Makefile, tests...) comme en atelier algorithmique et programmation. Cela a été un bon moyen d'approfondir nos connaissances. Certaines notions intéressantes ont été abordée, comme l'architecture client/server. Ce fut enfin une opportunité de nous faire travailler en groupe. Nous avons du nous organiser afin d'optimiser notre temps.

Nous avons cherché à toujours adopter un regard critique sur notre code, à le clarifier, le tester.

Le projet fonctionne avec notre client et notre serveur mais il présente quelques problèmes de compatibilité quand nous essayons d'utiliser des clients ou serveurs extérieurs.

Quelques difficultés ont été rencontrées et les stratégies des clients auraient pu être plus sophistiquées en terme "d'intelligence" du client. En effet, bon nombre améliorations sont encore possibles.