# Chapitre 1

# Serveur OCaml - Béatrice Carré

Le serveur a été réalisé en OCaml. Il utilise les modules Thread, Mutex Unix, Arg (pour parser les options du serveur) et Str (pour les expressions régulières du module Protocol), ainsi que ocamllex et ocamlyacc pour la gestion des commandes.

Nous détaillerons par la suite les choix d'implémentation qui ont été faits.

## 1.1 Etablissement du serveur

L'établissement du serveur a été fait en s'inspirant du chapitre 20 de *Développement d'Applications avec Objective Caml*. Comme spécifié dans le sujet du projet, la connexion s'appuie sur un protocole TCP sur le port 2013 par défaut. Le serveur a des paramètres par défaut modifiables par option au moment du lancement :

- le *timeout*, 30 secondes par défaut
- le nombre de joueurs *max* pour le lancement de la partie, 4 par défaut
- le *port* utilisé, 2013 par défaut
- le nom du fichier *dico*, "dico.txt" par défaut
- le nombre $n$ de joueur(s) devant dénoncer un dessinateur pour qu'il soit considéré comme tricheur, 2 par défaut

Avant l'établissement du serveur, les arguments sont parsés grâce au module *Arg*.

## 1.2 Traitement d'une commande - camllex et camlyacc

Afin de traiter les commandes reçues, en respectant le protocole, il était intéressant d'utiliser les générateurs d'analyseur syntaxique ocamllex et de parser ocamlyacc. Le site *http ://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html*

Après avoir créé des constructeurs de commande dans le fichier *protocol.ml*, il a fallu définir la grammaire du parser (dans *parser.mly*) et les expressions régulières manipulées (dans *lexer.mll*). La grammaire traduit les expressions définies en commande en utilisant les constructeurs du module Protocol.

Ainsi, commande et arguments sont aisément identifiés et manipulés. Une fois cela fait, pour traiter une commande reçue, il suffit de combiner le lexer et le parser, pour

obtenir le type *command* défini dans le protocole. Pour envoyer une commande, il suffira d'utiliser les constructeurs définis dans le module Protocol et de transformer la commande obtenue en chaîne de caractère, grâce à la fonction string_of_command du fichier protocol.ml

Le traitement des commandes se reposera par la suite sur le type somme défini et les fonctions associées à sa manipulation.

## 1.3 Connexion

La phase de connexion est séparée de la boucle principale de jeu afin d'identifier les conditions de connexion (joueur ou spectateur) et lancer le traitement adéquat. Si le pseudo reçu est déjà utilisé (joueur présent), alors un nouveau pseudo est généré en rajoutant un entier entre parenthèse à la fin. Au bout du max-ième joueur se connectant, la partie est lancée, dans un autre thread.

## 1.4 Structures de données

Lors de la connexion, la structure représentant le *joueur* est alors initialisée.

```
type player = {
  chan : Unix.file_descr; (*la socket par laquelle le serveur communique*)
  thread : Thread.t;
  name : string;
  mutable role : role;
  mutable already_draw : bool; (*s'il a deja dessine pendant la partie *)
  mutable has_found : bool; (* s'il a deja trouve le mot*)
  mutable score_round : int;
}
type role =
  | Undefined
  | Drawer
  | Guesser
```

Le role du joueur est représenté par un type somme qui nous permettra aisément de vérifier si une commande est licite à un moment de la partie donné selon son rôle.

Les joueurs sont stockés dans une liste (de taille maximale max), ce qui permet d'utiliser les nombreuses fonctions du module List. Les spectateurs, stockés dans une autre liste, ne sont représentés que par leur socket.

Ces listes sont dans une structure représentant l'état de la partie du *server* :

```
type server = {
  mutable players : player list;
  mutable spectators : Unix.file_descr list;
  mutable mots_rounds : string list; (*les mots deja sortis pour
  eviter de tomber sur les meme mots*)
  mutable is_game_started : bool;
  mutable commandes : Protocol.command list (* les commandes envoyees
  depuis le debut de la partie*)
}
```

Le *round* est représenté par la structure présentée ci-dessous :

```
type round = {
  timer : timer;
  mutable drawer : player option;
  mutable winner : string option;(*nom du premier joueur ayant trouve*)
  mutable word_to_find : string;
  mutable cpt_found : int; (* nombre de joueur ayant trouve le mot*)
  mutable nb_cheat_report : int;
  mutable color : color; (* par defaut noir*)
  mutable size : int; (* par defaut 0*)
}
```

Et enfin, le *timer*, qui a été implémenté en objet, pour pouvoir le manipuler facilement avec un accès simple aux méthodes. Ce timer est créé avec un temps initial (le temps de la partie) et une fonction de callback (ici, next_round). Il comprend :

```
class timer init_delay callback =
object(self)
  val mutable time = init_delay
  val mutable running = false
  val mutex_delay = Mutex.create ()
  val mutex_running = Mutex.create ()
  method start_count ();
  method restart_count ();
  method get_current_delay;
  method set_delay new_delay;
  method stop_timer ();
end
```

## 1.5 Extensions réalisées

### 1.5.1 Discussion instantanée

Cette extension a été plutôt simple à implémenter, dès que le serveur reçoit la commande *Talk message* correctement formée, il renvoie à tout le monde (clients et spectateurs) le message passé en argument précédé de son émetteur.

### 1.5.2 Courbes de Bézier

Du côté serveur, cette extension est de la même manière rapide à implémenter. Il a juste fallu ajouter une commande au protocole fourni pour permettre au dessinateur de dessiner une courbe : “SET_COURBE/x1/y1/x2/y2/x3/y3/x4/y4/”.

### 1.5.3 Spectateurs

Pour cette extension, j’ai supposé que les seules commandes à envoyer au spectateur est celle envoyées à tous les joueurs. Il a donc fallu ajouter l’envoi des commandes aux spectateurs présents lorsqu’on fait appel à la fonction *broadcast* et de stocker la commande pour permettre l’envoie de toutes les commandes précédemment envoyées depuis le début de la partie, à un spectateur se connectant en pleine partie

## 1.6 Architecture

L'architecture n'est pas forcément très rigoureuse car elle n'a pas été développée dans un esprit de réutilisabilité. Lors de la connexion d'une nouvelle socket, un thread exécutant *init_ new_ client* est lancé. Cette fonction lance la phase de connexion évoquée plus haut, et selon le résultat exécute *start_ player* ou *start_ spectator*. La deuxième fonction se contente de récupérer les commandes déjà effectuées et d'ajouter sa socket à la structure du server. La première effectue une boucle qui match les commandes reçues afin de lancer le traitement adapté selon son rôle.

## 1.7 Choix d'implémentation

L'énoncé étant parfois libre de choix, voici les décisions prises les concernant :

- Pour la commande Pass, j'ai respecté ce qui est décrit dans l'énoncé : cela produit un effet que si aucun joueur n'a deja trouvé le mot. Donc si un joueur a trouvé le mot, Pass ne marche pas du tout.
- fin partie TODO L'enchaînement de plusieurs parties n'est pas géré.
- La connexion d'un joueur en cours de partie n'est pas autorisé, pour éviter qu'un joueur ne commance la partie en ayant du retard au niveau des points.
- Les possibilités du spectators ne sont pas bien précisé, j'ai donc supposé qu'après se connexions, il ne pouvait que quitter le jeu avec la commande Exit. Cette commande prenant un nom en argument, elle est acceptée avec n'importe quel nom.