Rapport Projet PC2R

Elisabeth Abbas Zadeh

April 2019

1.Manuel utilisateur

Voir l'archive, manuel_utlisateur.txt.

2. et 3. Description détaillée et illustrée du projet et de ses composants

Mon projet est un client-serveur sur le jeu type "Asteroids". J'ai choisi d'implémenter mon serveur en Ocaml et mon client en Java.

Serveur

Mon choix s'est porté sur Ocaml pour la partie serveur grâce a la facilité de l'implémentation que ce langage propose pour ce genre de structure. En effet, Ocaml permet de réaliser un serveur complet sans que la taille du code ne devienne trop conséquente.

Les variables globales

```
let outchans:((int, out_channel) Hashtbl.t) = Hashtbl.create 50
let inchans:((int, in_channel) Hashtbl.t) = Hashtbl.create 50
```

Elles permettent la communication entre client et serveur (elles contiennent les channels).

```
let tps_dmr = 30.;;
let server_tickrate = 60.;;
let server_refresh_tickrate = 100.;;
let turnit=15;;
let thrustit=1.8;;
let fin_session = ref false;;
let ob_radius = 10.0;;
let ve_radius = 15.0;;
```

```
let joueurs:((int, string) Hashtbl.t) = Hashtbl.create 50
let numeros:((string, int) Hashtbl.t) = Hashtbl.create 50
let coordonnees:((string, (float*float)) Hashtbl.t) = Hashtbl.create 50
let vcoordonnees:((string, (float*float*float*float*float)) Hashtbl.t) = Hashtbl.create 50
let obstacles:(float*float) list ref = ref []
let coordonnee_obj:(float*float) ref = ref (0.,0.)
let scorestable:((string, int) Hashtbl.t) = Hashtbl.create 50
let phase = ref ""
```

Elles contiennent les variables servant à modifier l'exécution du serveur. Une partie correspondant aux informations concernant les entités du jeu (taille des vaisseau, vitesse de déplacement) ou encore des variables de rafraîchissement étant toutes indépendantes de l'interaction que le joueur aura avec le programme. Les autres variables sont celles qui seront modifiées par l'action du joueur ou encore les identifiants propres à un seul utilisateur.

Détail de certaines fonctions

```
let commande_tick () =
  let vcoords = vcoordonnees_to_vcoords() in
  print_endline vcoords;
  let tick = "TICK/"^vcoords^"/\n" in
  Hashtbl.iter (fun a b -> (output b tick 0 (String.length(tick)); flush b)) outchans;;
```

Cette fonction est exécutée en continue dans la fonction `serveur_tickrate_task ()`, elle permet de faire la mise à jour des coordonnées côté client.

```
let rec serveur_tickrate_task () =
  Thread.delay (1./.server_refresh_tickrate);
  commande_tick();

  if(not !fin_session) then
    serveur_tickrate_task ();
;;
```

J'utilise `Thread.delay()` afin de donner une fréquence de rafraîchissement au serveur, par conséquent il n'y a pas d'attente active. Les autres threads travaillent en parallèle de cette exécution, mais on manque de précision sur l'intervalle de durée entre les ticks, ils peuvent être plus longs que prévu car on ne maîtrise pas le scheduler.

Initialement, je souhaitais utiliser les threads coopératifs pour avoir une durée exacte de `server_refresh_tickrate`. Cependant j'ai rencontré un problème d'import et d'utilisation avec la librairie de fair threads Ocaml HirondML.
Ensuite, j'ai voulu implémenter ma **scheduled task** en Ocaml en utilisant les promesses et les futures mais je n'ai pas réussi à installer et utiliser Lwt : http://ocsigen.org/lwt/4.1.0/manual/manual.

J'avais changé ensuite hésiter à implémenter la fréquence de rafraichissement à l'aide d'un signal sigalarm pour avoir plus de précision : cristal.inria.fr/ remy/poly/system/camlunix/sign.html, mais j'ai finalement opté pour un `Thread.delay()` comme dans la fonction ci-dessous.

```
(*let attente_debut_session s =
  if !phase!="attente"
  then ignore(Sys.signal Sys.sigalrm (Sys.Signal_handle(de_attente_a_jeu))); ignore(Unix.al
  phase:="attente";; *)
```

J'ai cherché à implémenter la durée de la phase d'attente avec un `Sys.sigalrm`, mais vu que plusieurs threads auraient utiliser ces signaux, que les signaux ne se cumulent pas, et qu'ils sont partagés, ça n'auraient pas fonctionné. J'avais une erreur du style `Fatal error: exception Unix.Unix_error(Unix.EINTR, "accept", "")`.

Je me suis finalement rabattue sur des `Thread.delay()`.

Ci-dessous un exemple de synchronisation entre threads dans mon programme :

```
class connexion_maj sd sa b =
object(self)
  inherit connexion sd sa b as super
  method run () =
    while !continue do
      begin
        begin
          Mutex.lock m_session;
          if not (String.equal !phase "attente")
          then
            begin
              phase:="attente";
              ignore(Thread.create attente_debut_session self);
              nettoyer_variables_globales();
              commande_newobj ();
            end;
          Stack.push (self#getnum()) joueurs_session;
          while (Stack.length joueurs_session < Hashtbl.length joueurs)
          do Condition.wait c_session m_session
          done;
          (*si on n'attend plus personne.*)
          (* >= au cas où quelqu'un deco*)
          if (Stack.length joueurs_session >= Hashtbl.length joueurs)
          then
            begin
              Stack.clear joueurs_session;
```

```
              Condition.broadcast c_session
            end;
          Mutex.unlock m_session
        end;
        fin_session:=false;
        let lecteur = (Thread.create lecture self)
        in
        Thread.join lecteur;
      end;
    done
end ;;
```

Ma pile `joueurs_session` sert à envoyer en broadcast un signal permettant de débloquer la condition variable bloquant les autres threads.

```
if not (String.equal !phase "attente")
    then
    begin
        phase:="attente";
        ignore(Thread.create attente_debut_session self);
        nettoyer_variables_globales();
        commande_newobj ();
    end;
```

Le premier thread représentant un joueur exécute le code qui démarre le thread du minuteur avant la fin de la phase d'attente.

Lorsque le dernier des threads représentant un joueur arrive à la ligne `while (Stack.length joueurs_session` il ne se bloque pas. Les autres si. Il avance jusque-là :

```
if (Stack.length joueurs_session >= Hashtbl.length joueurs)
    then
    begin
        Stack.clear joueurs_session;
        Condition.broadcast c_session
    end;
```

et débloque les autres threads. Le tout est englobé de `Mutex.lock` et `Mutex.unlock`. Tous les threads lecteur débuteront un peu plus au même moment (indéterminisme lié au scheduler pour la suite de l'exécution).

Client

Pour la partie client, Java proposent une bibliothèque Swing permettant de réaliser des interfaces graphiques simples et identiques quel que soit le système d'exploitation. De plus l'utilisation de sockets pour la connexion au serveur est claire et expressive en Java, ce qui en fait un langage adapté pour la partie client de ce genre de structure.

Elle est structurée de la même sorte que le serveur en ce qui concerne le lecteur de commande (dans un thread, parsing des commandes avec des if). Lutilisation des sockets est simple, on lit sur le canal entrant de la socket, et on écrit sur le canal sortant selon la commande reçue.
Le reste de mes classes gèrent l'interface graphique.

Ma classe `JeuAsteroids` crée une Jframe et les Jpanel panelJeu et panelAttente, correspondant respectivement à l'affichage de la phase jeu et de la phase attente.

La fonction la plus représentative des problèmes de concurrence est la fonction qui met à jour la position des vaisseaux. J'utilise `synchronized()` quand je souhaite faire une section critique sans ajouter de signaux ni de `await()`. Sinon, j'ustilise `Lock` et `Condition`.

Ici, je pose une condition variable. Si le vaisseau est null alors qu'on souhaite envoyer les coordonnées au serveur dans `LecteurCommandes`, on attend qu'il soit mis à jour dans updateVaisseaux et on réveille ensuite l'autre thread pour qu'il envoie ses données au serveur.

```
public void updateVaisseaux(HashMap<String, Coordonnee> coordonnees){
        synchronized(vaisseaux) {
                for(String pseudo : coordonnees.keySet()) {
                        Coordonnee c = coordonnees.get(pseudo);
                        if(pseudo.equals(Client.pseudo)) {
                                LecteurCommandes.lockVaisseau.lock();
                                if(vaisseau==null)
                                        vaisseau=new Vaisseau(c.x, c.y);
                                else {
                                        vaisseau.setX(c.x);
                                        vaisseau.setY(c.y);
                                        vaisseau.setTheta(c.t);
                                }
                                LecteurCommandes.vaisseauNonNull.signal();
                                LecteurCommandes.lockVaisseau.unlock();
                        }
                        if(vaisseaux.containsKey(pseudo)) {
                                vaisseaux.get(pseudo).setX(c.x);
                            vaisseaux.get(pseudo).setY(c.y);
                        }
                        else
                                vaisseaux.put(pseudo, new VaisseauEnnemi(c.x, c.y));
                }

                repaint();
                Toolkit.getDefaultToolkit().sync();
        }
}
```

Problèmes rencontrés

- Swing de Java, l'utilisation de `paintComponent()`
- structurer proprement son interface graphique.
- liaison des fair threads ocaml HirondML.
- installer et utiliser Lwt (http://ocsigen.org/lwt/4.1.0/manual/manual)
- utilisation du signal `Sys.sigalrm` dans plusieurs threads.
- savoir qu'il faut activer le flush instantané du buffer qui contient les informations du paintComponent(), pour éviter de donner un effet de lag sur l'interface graphique Swing. `Toolkit.getDefaultToolkit().sync();`