PC2RON

Programmation Concurrente Réactive et Répartie

Projet d'UE 2011, à réaliser dans au moins deux langages

1 Presentation generale

Dans un futur proche où règnent sans pitié les enseignants de PC2R devenus fous, les étudiants doivent s'affronter au sein d'arènes impitoyables. L'organisation de ces joutes est laissée à des robots géants particulièrement infaillibles, puisque programmés en Esterel. Chaque affrontement se déroule selon le même rituel :

1. Quatre robots géants se placent aux milieux des côtés d'une vaste arène carrée dont le sol est couvert d'une grille de 1000x1000 rails.
2. Chaque robot pioche un pauvre étudiant sans défense dans un enclos.
3. Chaque malheureux élu se voit alors affublé d'un costume lumineux et est ligoté sur un monstrueux engin à deux roues, capable de rouler uniquement sur les rails de la grille. Pour les reconnaître, les ensembles véhicule/étudiant brillent d'une couleur bien distincte des trois autres.
4. Le *Grand Ordonnanceur* donne alors à sa guise le signal du départ, souvent simplement de la forme "3... 2... 1... GO!", mais parfois agrémenté de phrases de propagande.
5. Les quatre engins sont alors mis en route, à une vitesse difficilement soutenable. Les étudiants n'ont d'aucune manière la possibilité d'influencer la vitesse, qui est choisie par le *Grand Ordonnanceur*.
6. Les pilotes ont l'unique possibilité de diriger leurs bolides grâce à un guidon. Les engins sont cependant capables de tourner par accoups de 90 degrés uniquement, en suivant les rails de la grille.
7. Lorsqu'ils se déplacent, les véhicules laissent derrière eux un mur d'énergie, de la couleur et la hauteur de l'engin.
8. Si un véhicule heurte un mur ou un autre engin ou s'il sort de la grille, la mort est inévitable.

Le dernier survivant de l'arène est désigné vainqueur, il est remis dans un des enclos et pourra être à nouveau sélectionné pour une arène future.

2 But de ce projet

Dans ce projet, vous devrez programmer une simulation de cette (fort probable) vision de l'avenir, sous la forme d'un jeu multi-joueurs, à architecture client/serveur. Le serveur prendra en charge les rôles du *Grand Ordonnanceur* et ceux des robots. Les clients simuleront chacun un étudiant. Il vous est demandé d'utiliser au moins deux langages. La solution minimale est donc de fournir un client et un serveur dans deux langages, mais vous pouvez aussi rendre plusieurs serveurs et/ou plusieurs clients.

3 Communications dans la grille

Dans un premier temps, vous devrez implanter un mécanisme générique de protocole binaire. Ce mécanisme sera utilisé pour les communications entre le serveur et les clients, est directement inspiré du vrai protocole reliant l'ordinateur central de la grille aux ordinateurs de bord des véhicules.

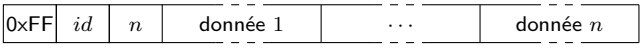
Concrètement, on réalise une couche transport au dessus du protocole TCP. La transmission est décomposée en trames successives, chacune encapsulant une succession de données primitives (entiers, chaînes, etc.).

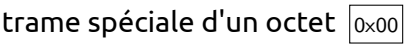
3.1 Trames

Chaque trame débute par un en-tête :

1. l'octet fixe 0xFF,
2. un octet id disponible pour identifier différents types de trames au sein du procole (sans signification au niveau de la couche transport),
3. un octet indiquant le nombre n de données primitives encapsulées dans la trame. La taille totale ne peut être calculée à l'avance, puisque les données ont une taille variable.

Puis les n sont transmises consécutivement, on a donc la forme suivante :

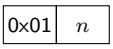
- 

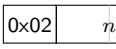
Une trame spéciale d'un octet 

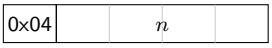
signifie la fin de la transmission.

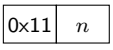
3.2 Données

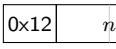
Chaque bloc de donnée débute par un octet indiquant le type du contenu. Les blocs définis par le protocole sont décris ci-dessous. Les sections de plus d'un octet s'entendent en gros boutien.

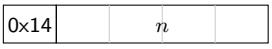
- 

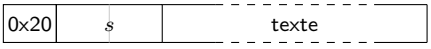
entier signé n tenant sur 1 octet.
- 

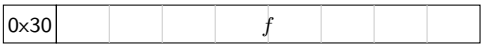
entier signé n tenant sur 2 octet.
- 

entier signé n tenant sur 4 octet.
- 

entier non signé n tenant sur 1 octet.
- 

entier non signé n tenant sur 2 octet.
- 

entier non signé n tenant sur 4 octet.
- 

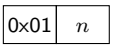
chaîne en UTF-8 dont l'encodage tient sur s octets.
- 

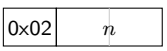
flottant en double précision

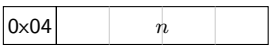
3.3 Rendu

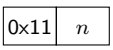
Il vous est demandé de réaliser un serveur, qui écoute sur son port 5555 et affiche sur sa sortie les trames reçues, avec le format qui suit (on réutilisera ce format dans le suite pour décrire les trames du protocole).

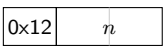
Chaque trame est affichée entre deux accolades (`'{'` et `'}'`). Les accolades englobent l'identifiant en hexadécimal sur deux caractères, puis les données séparées par des virgules, en utilisant les formats suivants.

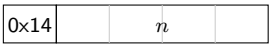
- 

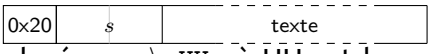
: `int8` n (n en décimal)
- 

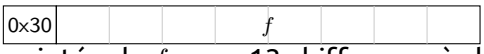
: `int16` n
- 

: `int32` n
- 

: `uint8` n
- 

: `uint16` n
- 

: `uint32` n
- 

: `string "`s`"` (où les octets < 32 et > 127 sont remplacés par `\xHH` où HH est la représentation hexadécimale de l'octet sur deux caractères)
- 

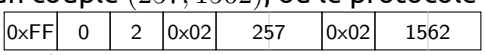
: `double` $p(f)$ (où $p(f)$ est la notation décimale pointée de f, avec 13 chiffres après la virgule)

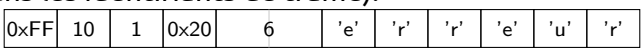
Cet exercice sera en partie **corrigé automatiquement**. Votre archive devra contenir **à la racine** un fichier de script `run_exo_protocole`, qui lance le serveur (en le compilant si besoin). Votre serveur devra se terminer s'il reçoit la trame de fin de transmission. Faites attention de ne pas laisser de messages de débogage qui fausseraient la correction automatique. Il y aura de plus une correction non-automatique, afin de juger de la clarté du code.

Il vous est demandé d'utiliser à bon escient les primitives du langage choisi (héritage/patterns en Java, définitions de types en OCaml, etc.), afin de rendre facilement extensible la couche transmission d'une part, et de construire facilement différents protocoles utilisant cette couche transport d'autre part.

3.4 Exemples

Voici quelques exemples de trames complètes (issues de protocoles fictifs utilisant cette couche transport), ainsi que les affichages formatés.

- Envoi d'un couple $(257, 1562)$, où le protocole n'utilise pas les identifiants de trame.
  Trame : 

  Affichage : `{ 0x00, int16 257, int16 1562 }`
- Envoi du message texte "erreur" de priorité 10 (le protocole a choisi d'encoder des priorités dans les identifiants de trame).
  Trame : 

  Affichage : `{ 0x0A, string "erreur" }`

4 Deroulement d'une partie

Chaque partie se déroulera selon les étapes suivantes (on donne seulement le point de vue du client, à vous de déduire celui du serveur).

1. Connexion puis inscription en spécifiant son nom et la couleur désirés.
2. Attente de connexion d'autres joueurs.
3. Réception des données de tous les concurrents (éventuellement avec correction des paramètres choisis à l'inscription s'il y a confusion possible avec un autre participant).
4. Envoi du signal "prêt" au serveur, puis réception et affichage des messages de décompte, se terminant au message "GO!".
5. Déroulement de la partie (plusieurs variantes à implanter de cette étape seront décrites en détails par la suite).
6. Au cours de la partie, annonce de la fin de partie et du vainqueur par le serveur.
7. Déconnexion du client.

5 Protocole de la grille

Dans le protocole de la grille, on utilise les identifiants de trame pour repérer des types de trame prédéfinies. On identifiera les différents types de trame par une lettre, et on utilisera le code de cette lettre comme identifiant de trame. Par exemple, une trame I aura pour identifiant l'octet 73.

5.1 Séquence d'initialisation

Cette section décrit la séquence d'initialisation pour un client.

1. Lors de l'établissement de la connexion TCP, le client envoie sans attendre la trame I (*Initiate*) suivante, contenant le nom de version du protocole, destinée à s'assurer que le serveur reconnaît le protocole utilisé. Le nom de version par défaut est `"PC2RON2011"`. Si vous modifiez le protocole en y ajoutant des extensions, veillez à le changer.

   `{ 0x49, string "PC2RON?", string "PC2RON2011" }`

   Si c'est bien le cas, le serveur répond par la trame A (*Ack*) suivante, reprenant le nom de version.

   `{ 0x41, string "OK", string "PC2RON2011" }`

   Sinon, la trame A alternative est la suivante, et contient le nom de version du serveur.

   `{ 0x41, string "NO", string "PC2RON2011_MOD" }`

2. Le client demande alors sa connexion par une trame C (*Connect*), en spécifiant son nom et sa couleur, sous la forme de trois composantes entières (rouge, verte et bleue) entre 0 et 255.

   `{ 0x43, uint8` R`, uint8` V`, uint8` B`, string` nom `}`

Le serveur répond alors une trame R (*Registered*) en précisant l'entier *id* (non signé et tenant sur 16 bits) identifiant de façon unique le joueur durant la partie. Le choix des identifiants est laissé libre.

`{ 0x52, string "OK", uint16` id `}`

En cas de problème, le serveur peut répondre une forme alternative comprenant un message d'erreur.

`{ 0x52, string "NO", string` $message$ `}`

3. Lorsque tous les joueurs sont arrivés, le serveur envoie alors à tous les clients et pour chaque joueur la trame U (*User*) suivante. dir prend les valeurs 1, 2, 3 ou 4 signifiant respectivement haut, bas, gauche et droite. $speed$ donne le nombre de pas de grille effectués en un centième de seconde par l'étudiant.

   `{ 0x55, uint16` id`, string` nom`,`
   `uint8` R`, uint8` V`, uint8` B`,`
   `uint16` x_0`, uint16` y_0`, uint8` dir`, uint8` $speed$ `}`

4. Le serveur précise la fin de la liste des joueurs avec une trame E (*End*).

   `{ 0x45 }`

5. Le serveur envoie alors les messages de décompte. Chaque message prend la forme d'une trame P (*Pause*). Le dernier message, s'accompagnant du démarrage de la partie, est une trame S (*Start*).

   `{ 0x50, string` $message$ `}`

   `{ 0x53, string` $message$ `}`

   Le serveur peut aussi envoyer ces trames durant la partie, signifiant une pause/reprise générale du jeu. La trame S permet aussi d'afficher un message en cours de partie sans mettre en pause le jeu.

5.2 Version synchrone avec état global

Cette version est celle correspondant au protocole `"PC2RON2011"`.

Cette première version est la plus simple, mais la plus gourmande en bande passante et la moins flexible en termes de délais et de disparités entre machines. Concrètement, la gestion du temps est globale et discrète (par instants partagés entre toutes les machines) et sa gestion est centralisée (c'est le serveur qui effectue la synchronisation entre les machines et décide du changement d'instant). Il n'y a qu'un seul état global, toutes les machines disposent des mêmes informations à un instant donné.

> Dans votre rapport, en plus de la description du code de chaque version, vous discuterez des avantages et inconvénients des modèles associés. Vous pourrez appuyer votre argumentation en vous basant sur des scénarios mettant en jeu des critères fonctionnels ou des environnements de déploiement différents.

Pour obtenir ce modèle, à chaque instant, le serveur effectue les opérations qui suivent.

1. En fonction de la vitesse et de l'état actuel des bolides, il calcule les nouveaux états des participants.

   Pour ne pas surcharger le réseau, il est judicieux de limiter la fréquence des instants, mais attention à conserver une bonne réactivité.

2. Il envoie les positions et directions de tous les participants encore vivants, sous la forme d'une trame T (*Turn*) comme suit (où n est le nombre de participants de la trame). La trame contient aussi le nombre t de centièmes de secondes écoulées depuis le début de la partie.

   `{ 0x54, unit32` t`,`
   `unit16` id_1`, uint16` x_1`, uint16` y_1`, uint16` dir_1`,`
   $\ldots$
   `unit16` id_n`, uint16` x_n`, uint16` y_n`, uint16` dir_n `}`

3. Si un participant meurt, il envoie la trame D (*Death*) qui suit avant la trame de mise à jour normale.

   `{ 0x44, uint16` id `}`

4. Si un participant a gagné, il envoie la trame W (*Win*) suivante, puis ferme la connexion.

   `{ 0x57, uint16` id `}`

5. Si personne ne gagne (collision entre les deux survivants), il envoie la trame D contenant les deux morts.

   `{ 0x44, uint16` id_1`, uint16` id_2 `}`

6. Il attend de chaque client un ordre, sous la forme d'une trame O (*Order*) qui peut prendre les formes suivantes :

   - `{ 0x4F, string "idle" }` pour continuer tout droit
   - `{ 0x4F, string "left" }` ou `{ 0x4F, string "right" }` pour tourner
   - `{ 0x4F, string "abandon" }` pour abandonner

5.3 Version synchrone avec état local

Cette version correspond au protocole `"PC2RON2011_local"`.

Dans cette version, on veut minimiser les transmissions d'informations pouvant être facilement calculées localement par chaque client, tout en conservant une synchronisation globale par instants.

Pour ceci, la trame T envoyée à chaque instant par le serveur ne mentionne que les n participants dont la direction a changé, et ne renseigne plus les positions.

`{ 0x54, unit32` t`,`
`unit16` id_1`, uint16` dir_1`,`
$\ldots$
`unit16` id_n`, uint16` dir_n `}`

5.4 Version asynchrone

Cette version correspond au protocole `"PC2RON2011_async"`.

Dans cette version, il n'y a plus de synchronisation globale à chaque instant. L'envoi des ordres du client, et la transmission des changements se font de façon indépendante.

1. Lorsque le client veut changer de direction, il envoie une trame de changement, identique aux versions précédentes.

Il n'aura le droit d'envoyer un nouvel ordre que lorsque le serveur aura renvoyé une trame T *(Turn)* comme suit. Si le client envoie trop d'ordres, le comportement du serveur n'est pas défini, il peut les ignorer, les mettre en attente ou même tuer le participant.

`{ 0x54 }`

2. Lorsqu'un participant change de direction, le serveur envoie à tous les participants une trame N (*Notify*) comme suit, où t est le nombre de centièmes de secondes écoulés depuis le début de la partie au moment où l'ordre a été pris en compte dans les calculs du serveur.

   `{ 0x4E, unit32` t`, unit16` id`, unit16` dir `}`

   Le serveur doit envoyer les trames de mise à jour dans l'ordre chronologique.

5.5 Rendu

Dans votre archive, vous devez rendre un client et un serveur pour chaque version du protocole. Vous fournirez en plus des sources, les scripts permettant de lancer les clients et les serveurs de chaque version. Ces scripts devront se trouver à la racine et porter les noms `run_client(_local|_async)` et `run_server(_local|_async)`.

6 EXTENSIONS

Si vous avez terminé le travail imposé, il y a de nombreuses améliorations possibles.

- Bonus/malus apparaissant sur le terrain.
- Serveur gérant plusieurs parties en parallèle.
- Mécanisme de gestion d'utilisateurs, de tableau de scores, etc.
- etc.