# Programmation avancée

## TP n° 3 : un jeu en réseau - le diamant

**Détails**

Écrit par stéphane Domas

Catégorie : Programmation avancée (/index.php/menu-lpsil/objets-connectes)

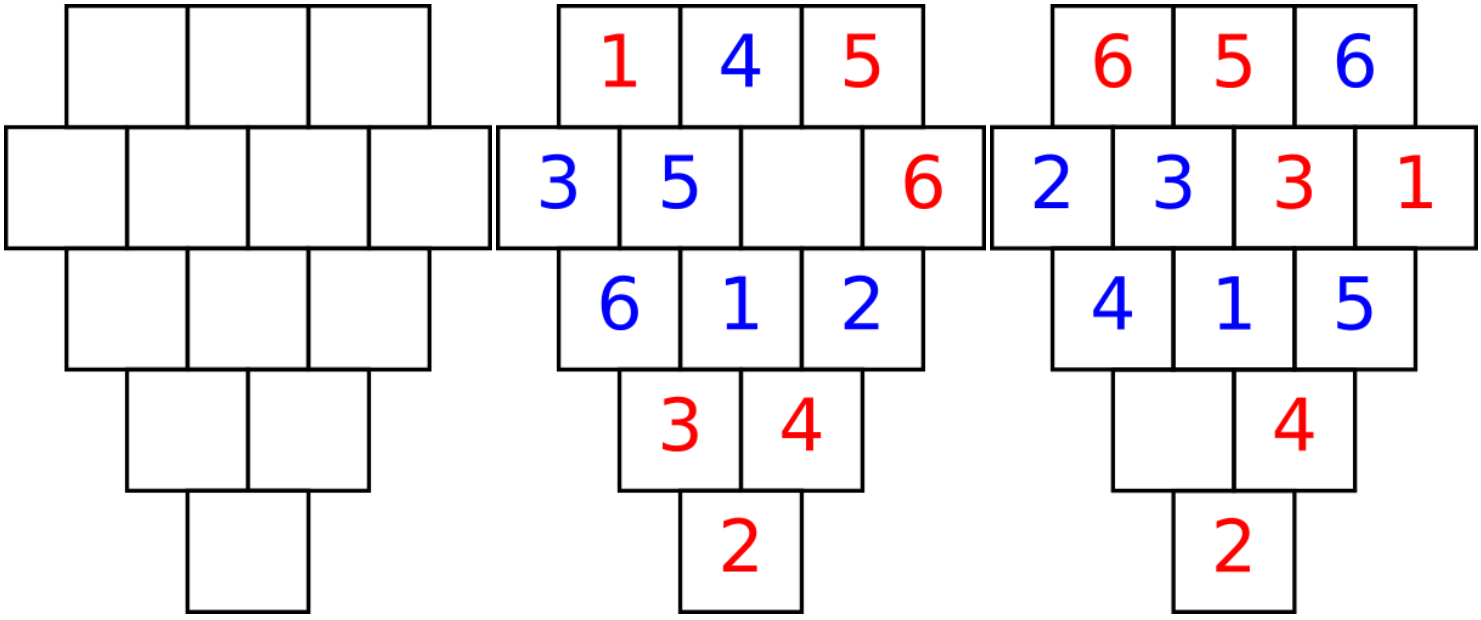
Publication : 26 septembre 2014

Affichages : 1016

## 1°/ Principe du jeu

- Ce jeu est un jeu de placement stratégique à deux joueurs, un bleu et un rouge
- Le plateau du jeu ressemble à un diamant avec 13 cases, comme indiqué par la figure ci-dessous.
- Chaque joueur possède 6 pions numérotés de 1 à 6.
- En commençant par le joueur bleu, chacun à son tour pose un pion sur le plateau, sachant que l'ordre des pions est tiré aléatoirement.
- A la fin de la phase de pose, il reste donc une case libre.
- On fait le total des valeurs des pions bleu et rouge autour de cette case vide.
- Le gagnant est celui dont le total est le plus bas.

Illustrations :




le plateau vide — bleu perd : 12, rouge gagne : 11 — bleu gagne : 5, rouge perd : 6

## 2°/ Le diamond en réseau

Ce TP permet d'utiliser les connaissances sur les mutex et attente d'événements, dans le cadre d'un serveur multi-threadé, mais non basé sur des requêtes. Dans ce contexte, la structure du serveur principal est relativement identique puisqu'il s'agit d'attendre des connexions et de créer des threads. En revanche, le code des threads est organisé de façon très différente, puisqu'il n'y a pas de notion de requête.

### 2.1°/ Fonctionnement général

Les principes de fonctionnement généraux du jeu sont les suivants :

- chaque fois qu'un client se connecte, le serveur principal crée un thread pour communiquer avec le client.
- le thread commence par envoyer au client s'il lui est possible de jouer, ... ou pas quand il y a déjà une partie commencée.
- ensuite le thread attend du client un pseudo. S'il est valide (=non existant), le client est considéré comme un joueur valide pour la partie à venir et le thread prévient son client qu'il peut continuer l'exécution.
- quand il y a deux joueurs valides, la partie commence. Le serveur principal continue d'accepter des connexions, mais les threads ainsi créées envoient immédiatement à leur client qu'il n'est pas possible de jouer.
- la partie se termine soit normalement après placement de tous les pions, soit lorsqu'un joueur se déconnecte en cours de jeu.
- l'état du jeu n'est géré qu'au niveau serveur. Les clients n'ont donc pas la possibilité de tester la validité d'un placement de pion, la fin de partie, etc. A noter que ce n'est pas toujours une bonne idée pour des raisons de performance : si le client peut faire lui même des vérifications, cela allège la charge du serveur. Le choix est dépendant du type de jeu.

Ces contraintes permettent de déduire les éléments de programmation suivants :

- les threads doivent se partager un objet représentant l'état de la partie,
- cet objet contient des méthode/blocs synchronized permettant de mettre à jour ou obtenir l'état de la partie.
- il y a plusieurs attentes d'événement à mettre en place :
  - attente pour commencer la partie,
  - attente de fin de tour de jeu (le thread qui communique avec le joueur dont ce n'est pas le tour, doit attendre celui dont c'est le tour).
  - attente de fin de partie, que ce soit normalement ou bien sur déconnexion, afin de réinitialiser proprement l'état de la partie avant que tout autre client puisse
- Ces attentes étant liées à l'état de la partie, elles sont logiquement incluses dans l'objet représentant la partie.
- Dans tous les cas, ce sont des barrières de synchronisation. On peut donc utiliser le principe du sémaphore pour les implémenter simplement.
- A la fin de chaque tour de jeu, l'état "visuel" du plateau doit être envoyé à chaque client, plus l'état de la partie afin que le client sache si elle est terminée ou pas.

## 2.2°/ Protocole de communication

Comme l'application est un jeu à tour de rôle, le protocole de communication n'est pas orienté requête. C'est plutôt le thread côté serveur qui "prévient" les clients de ce qu'ils doivent faire :

- juste après la connexion, le thread envoie au client un message "OK" si le client peut continuer, ou bien "ERR PARTY_STARTED" si une partie est déjà en cours et que le client doit se terminer.
- dans le premier cas, le client envoie ensuite un pseudo. Si le pseudo est libre, le thread renvoie "OK".
- si le pseudo est déjà pris, le serveur renvoie "ERR PSEUDO". Dans ce cas, le client doit renvoyer un nouveau pseudo.
- en cas de pseudo libre, le thread tente de l'enregistrer comme joueur valide pour la partie à venir. Cela ne fonctionne que s'il n'y a pas déjà 2 autres threads qui ont enregistré un pseudo valide.
- si l'enregistrement fonctionne, le thread renvoie "OK" au client pour lui signifier qu'il peut continuer. Sinon, il renvoie "ERR PSEUDO_REJECTED" qui signifie au client qu'il doit s'arrêter.
- après connexion réussie de 2 clients, la partie commence et chaque thread entre dans une boucle qui se termine sur fin de partie (ou déconnexion) :
  - envoi le visuel courant du plateau
  - envoi au client du numéro du joueur courant (en début de partie = 0).
  - si thread du joueur courant :
    1. envoi de la valeur du pion à jouer
    2. attente de l'ordre de jeu
    3. vérification de sa validité : si oui renvoie "OK", sinon renvoie "ERR ordre invalide" et retour en 2
    4. calcul du résultat du coup joué et modification de l'état de la partie, numéro nouveau joueur, ...
    5. signal fin de tour
  - sinon, attente de fin de tour de jeu
  - envoi de l'état de partie : "CONT" si elle n'est pas terminée, "END X Y" si elle est finie, avec X le score du joueur 0 et Y le score du joueur 1. On en déduit celui qui gagne en prenant le minimum de X et Y
  - envoi de l'état visuel du plateau de jeu

## 2.3°/ Problématiques d'implémentation

L'état de la partie est partagé entre les threads et doit contenir toutes les méthodes permettant de manipuler cet état de façon "thread-safe", c'est-à-dire sans que des threads entrent en conflit. Pour cela, il suffit de créer une classe `Party`, avec toutes les informations nécessaires au déroulement de la partie (pseudo des joueurs, liste des pions à jouer pour chacun, le plateau de jeu, sémaphores, ...). La plupart des méthodes sont relativement simples à écrire, et dans un cas aussi simple que ce jeu, elles peuvent être simplement déclarées `synchronized` pour utiliser le mutex de l'objet `Party`. Il y a cependant une exception avec les méthodes qui manipulent les sémaphores, qui ne doivent surtout pas être `synchronized` (cf. explications dans les sources à télécharger).

Les problèmes principaux de ce genre de jeu sont plutôt dans l'écriture du code du serveur principal et des threads, notamment pour la gestion :

- du début de partie avec un nombre précis de joueurs,
- des déconnexions client en plein milieu d'une partie.

Pour le jeu "diamond", une partie se joue uniquement à 2 joueurs. Pour autant, il ne suffit pas d'attendre 2 connexions clientes et créer 2 threads pour commencer une partie, par exemple :

```
1  party = new Party();
2  ...
3  while(true) {
4    for(int i=0;i<2;i++) {
5      sockComm = waitClient.accept();
6      ThreadServer t = new ThreadServer(sockComm, party, ...);
7      t.start();
8    }
9    party.waitParyEnd(); // attendre fin de partie et donc fin des 2 threads
10 }
```

En effet, les joueurs doivent d'abord envoyer un pseudo valide avant d'accéder potentiellement à une partie. Cet envoi de pseudo ne peut évidemment pas être géré au niveau du serveur principal, sinon ce dernier serait bloqué dès qu'un joueur met du temps à choisir un pseudo. Il faut donc que ce soient les threads qui gèrent l'échange de pseudo.

Le problème avec l'exemple de code ci-dessus est qu'un client peut se déconnecter durant l'échange de pseudo. Le thread va détecter cette déconnexion et s'arrêter. Il faudrait donc que le serveur principal puisse accepter un nouveau client, ce qui n'est pas possible puisqu'il fait une boucle à 2 itérations !

Une solution simple et générique à ce problème est de toujours autoriser la connexion, mais à chaque étape principale de l'exécution, de prévenir le client s'il peut continuer, ou bien s'il doit s'arrêter parce qu'une partie est déjà en cours. Cependant, cela implique d'en tenir compte dans le protocole de communication entre le client et le serveur. C'est ce qui est fait dans celui de la section 2.2, puisque le premier envoi consiste à dire au client nouvellement connecté si une partie est déjà en cours, ainsi qu'après l'envoi d'un pseudo valide s'il peut effectivement être enregistré comme joueur. Cette solution simplifie également le serveur principal car celui-ci n'a jamais besoin d'attendre la fin de partie, comme dans le code bancal ci-dessus.

*Remarque* : une solution bien plus complexe consiste à mettre en place une file d'attente d'accès à une partie. C'est notamment utile lorsque l'on veut garder le client en attente de l'accès à une partie et le prévenir dès que c'est possible.

Gérer les déconnexions brutales est souvent encore plus complexe. En effet, il faut obligatoirement qu'une partie se termine proprement afin d'être réinitialisée. Cela implique de mettre une barrière de synchronisation juste à la fin de la méthode `run()` et de s'assurer que TOUS les threads passent par cette barrière. Le premier à passer la barrière indique que la partie doit se terminer, et le dernier réinitialise l'état de la partie. Le premier problème est donc de structurer le code pour qu'un thread puisse atteindre directement cette barrière en cas de déconnexion de son client. Ce n'est pas forcément évident d'obtenir une telle structuration, surtout quand il faut jongler avec les cas d'exception.

Malheureusement, ce n'est pas suffisant. En effet, il est fort possible qu'un thread atteigne cette barrière alors que d'autres sont bloqués dans une autre barrière, ce qui produit un interblocage. Voici un tel cas pour le "diamond" :

- c'est au tour du joueur 0 => son thread lui envoie quel pion il doit jouer.
- pendant ce temps le thread du joueur 1 passe directement à la barrière qui lui permet d'attendre que le 0 ait fini de jouer.
- le joueur 0 ne répond rien et se déconnecte => son thread détecte la déconnexion et va directement à la barrière pour attendre la fin de partie.
- comme il est le premier à la barrière, il signale que la partie doit se terminer et attend que l'autre thread arrive à la barrière => interblocage puisque le thread 1 attend que le 0 finisse de jouer.

Une solution simple consiste à inclure dans la barrière de fin de partie des instructions qui permettent de débloquer des threads qui seraient en attente dans d'autres barrières. Ce n'est malheureusement pas totalement satisfaisant dans certains jeux complexes car les threads ainsi débloqués peuvent ensuite être en attente de données venant du client. Et si ce dernier n'envoie rien, alors il est impossible de "forcer" la fin de partie. Ce deuxième problème se résoud en ajoutant dans le code des vérifications de l'état de la partie avant chaque réception (NB : les envois n'étant pas bloquant, pas besoin d'une telle précaution). Si la partie doit se terminer, alors pas besoin de faire la réception et le thread saute à la barrière de fin de partie. C'est une solution "lourde" mais efficace.

En conclusion, on voit que cette gestion des déconnexion pose de fortes contraintes dans l'écriture du code des threads et même dans celle des objets partagés.

## 3°/ Exercice

L'objectif est simple : implémenter un client et un serveur afin de jouer au diamond, selon les spécifications données ci-dessus.

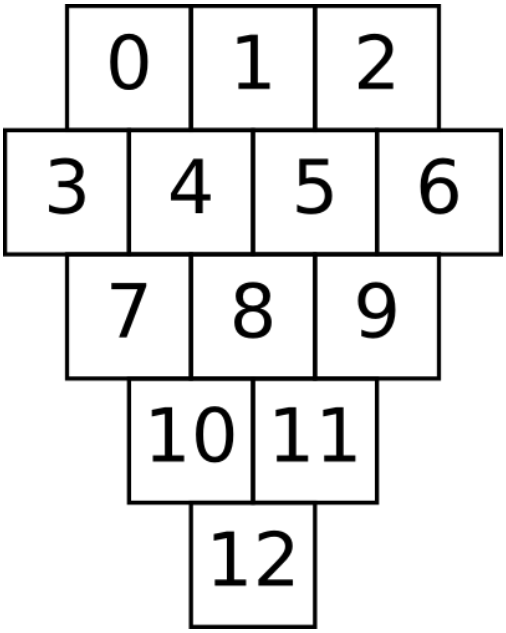
Pour vous faciliter la tâche, vous avez accès à une archive comportant des sources à compléter pour le [ client (/upload/supports/S5/ProgAvancee/TP/tp3/diamond-client-canevas.tgz) ] et le [ serveur (/upload/supports/S5/ProgAvancee/TP/tp3/diamond-server-canevas.tgz) ]. Les parties à compléter sont indiquées par un commentaire `/* A COMPLETER ...*/` dans lequel les étapes nécessaires sont indiquées. Bien entendu, vous devrez lire et comprendre les autres classes fournies pour compléter correctement les codes.

Pour information, les classes à compléter sont :

- `ClientTCPDiamond` : contient le code principal du client,
- `Party` : la classe permettant de gérer l'état de la partie et son déroulement. NB : une majorité du code est écrit, notamment tout ce qui concerne les attributs et leur initialisation.
- `ServerThreadDiamond` : le code des thread serveur.

Commentaires additionnels :

- La classe `ServerTCPDiamond` représente le serveur principal, qui va créer les threads. On remarque que son code est identique à celui donné dans le canevas serveur multi-threadé car il accepte des connexions même si une partie est déjà en cours (cf. remarques sur la gestion du début de partie ci-dessus).
- La classe `Semaphore` permet d'implémenter un semaphore classique, sous la forme d'une boite à jetons. On peut mettre et retirer autant de jetons que l'on veut avec `put()` et `get()`. Pour cette dernière, s'il n'y a pas suffisamment de jetons, le thread appelant est mis en attente.
- La classe `Board` représente le plateau de jeu sous la forme d'un simple tableau de byte (en Java). Chaque case du plateau correspond à une case de ce tableau dont le numéro est donné par la figure ci-dessous.




- Ces cases sont représentées par un attribut de type tableau de 13 `byte` nommé `board`
- Quand on place un pion dans une case donnée, cela revient simplement à mettre une valeur dans une cellule associée de `board`. Par exemple, si un joueur veut poser un pion au milieu de la 3ème ligne, la valeur du pion est mise dans `board[8]`. Encore faut-il déterminer quelle valeur !
- Dans le vrai jeu, les pions bleu et rouge sont chacun numérotés de 1 à 6. Pour les différencier dans le tableau `board`, une numérotation différente est utilisée : de 1 à 6 pour les bleus, de 7 à 12 pour les rouges.
- Attention, cela ne concerne que leur représentation dans le tableau et en aucun cas leur valeur. Par exemple, le pion n°7 correspond toujours au pion rouge valant 1. Cela est important pour que le comptage des scores soit fait correctement.
- Chaque case est environnée par 3 à 6 cases voisines. Pour déterminer les points des joueurs, il faut connaître quelles sont les cases voisines de la case vide. Pour éviter de calculer à chaque fois quelles sont ces cases, le tableau à deux dimensions `neighbors` permet de facilement retrouver les voisins de chaque case. Comme il y a 13 cases avec au maximum 6 voisins, ce tableau 2D est déclaré `char neighbors[13][6]`
- A noter que pour les cases n'ayant pas 6 voisins, le choix a été fait de mettre -1 dans les cases inutiles. Par exemple, pour la case 0, on initialise comme suivant : `neighbors[0][0] = 1, neighbors[0][1] = 3, neighbors[0][2] = 4`, et les 3 derniers avec -1.