Projet PROG6 : Réseau

Guiroux Hugo

17 mai 2013

1 Plan

- Architecture de l'application répartie
- Protocole d'interaction
- Messages
- programmation avec sockets TCP/IP en Java

2 Architecture de l'application répartie

Deux modèles : **liaison directe entre deux machines** et **serveur central + clients**.

3 Liaison directe

Deux instances du jeu, un joueur par instance
Eventuellement adaptable à N joueurs par instances mais plus compliqué.
Dans ce schéma, chaque instance exécute à la fois l'interface et le moteur de jeu
L'état du jeu est dupliqué et doit rester cohérent
Problème à considérer

3.1 Démarrage

Les deux instances jouent un rôle symétrique.
Mais du point de vue de TCP, il faut nécessairement un client et un serveur
Le rôle de client ou serveur au niveau TCP n'a pas nécessairement d'impact sur les caractéristiques applicatives (ordre de passage, rôle dans le jeu, ...)

3.2 Mélange des interactions GUI et réseau

La GUI peut être gelée en attente d'un message de l'autre joueur.
Si le (seul) thread de la GUI effectue une lecteur bloquante sur la socket de dialogue.
Pise de solution
Découpage des activités en utilisant plusieurs flots d'éxécution (ici, des threads).

- 1 thread GUI
- 1 ou 2 thread réseau (émission/réception)
- 1 thread moteur

Schéma producteur-consommateur :

- Chaque thread dispose d'une file d'attente (boite aux lettres) dans laquelle on peut lui poster des messages.

- Chaque thread traite ses messages un par un ... et un traitement peut produire un ou plusieurs messages à destination des autres threads
- Pour le therad réception réseau, la file correspond à la socket de dialogue
- Pour le thread gui ...

4 Serveur central

N instances (1 joueur par instance) + 1 serveur central
Eventuelement adaptable à M joueurs par instance mais plus compliqué.
Dans ce schéma
Le serveur exécute le moteur de jeu et gère son état.
Les clients exécutent uniquement l'interface du jeu.

Problème sà considérer
Mélange des interactions GUI et réseau
Propagation des mises à jour liées aux actions d'un client.
Vers tous les autres clients.

Pises de solutions :
Client : cf piste proposée pour le premier modèle
Mais pas de véritable thread moteur. Seulement une simple conversion (bidirectionnelle) entre événements réseau et événements GUI.

Serveur centra l
- 1 thread pour le moteur
- 1 ou 2 thread de dialogue par client
- Schéma producteur-consommateur(s)
  - 1 file dans le sens client <-> serveur
  - N files (1 par client) dans un sens serveur clients

5 Serveur salon de jeu

Quelle que soit l'architecture retenue, on peut éventuellement utiliser un serveur faisant office de salon de jeu
Intérêt :
Point de rencontre / annuaire des utilisateurs (dont les machines ou adresse IP peuvent changer d'une fois sur l'autre)
Authentification des utilisateurs.
Mémorisation des scores, etc.

Pour le 1er modèle : Le serveur salon est seulement un point de rendez-vous. Il n'est pas impliqué au cours d'nue partie.
Le serveur salon créé une instance de serveur central pour chaque partie en cours.

6 Protocol d'interaction

6.1 L'application répartie est composée d'un ensemble d'instances qui interagissent

Il faut spécifier clairement les séquences d'interactions possibles.
Sous la forme d'une ou plusieurs machines à états : En particulier, dans un état donné, quels sont les transitions possibles

- Réception d'un message ? Et le cas échant, quel message ?
- Action de l'utilisateur local ? Et le cas échant, quelles actions ?
- Les deux types d'événements

Bien vérifier que la spécification du protocole est correcte, en considérant les critères suivants :

- **Sûreté : Est-ce que l'état global de l'application est toujours cohérent ?** Exemple : Des instances peuvent elle avoir une vue divergente de la séquence de coups déjà joués ? Ou du nombre de pions/cartes/points dont dispose chaque joueur ?
- Progrès : L'application peut-elle toujours avancer (en supposant que les joueurs répondent en temps borné) ?
  - Peut-on tomber dans une situation où les différentes instances s'attendent mutuellement ?
  - Peut-on tomber dan sun dialogue sans fin entre plusieurs instances (non lié à l'attitude des joueurs) ?

Gestion des cas d'erreur et de terminaison.
Parmi les cas d'erreurs possibles quels sont ceux :
qui nécessitent seulement un traitement local ?
Qui nécessite d'informer une ou plusieurs des autres instances ?

A quelles phases de l'application un utilisateur peut-il demander sa terminaison ?

Comment/où est gérée la déconnexion brutale d'un utilisateur ?
Détection ?
Réaction ?
Il vaut mieux empécher le problème dans le protocol (évite annulation de coup => frustration).

7 Messages

Rappels :
Un canal TCP est un flux d'octets
En réception, l'application doit lire et découper ce flux en une séquence de messages.
Le protocol doit définir :

- Les types de emssages utilisés
- Le format de chaque type de message

Selon les besoins on peut utiliser

- Des messages de taille fixe (peut êtr econtraignant)
- Des messages de taille variable
- Nécessitent un en-tête de taille fixe et/ou des marqueus

Transmission des arguments d'un message :
Par valeur
Pas de difficulté pour les types de base
Nécessité de "sérialiser" les structures de données plus complexes. Deux aspects : Transformer une représentation spatiale (avec des liens d'indirection) en une séquence d'octets.
Obtenir une représentation auto-contenue, c-a-d sans références (adresses mémoire) qui n'ont de sens que localement.

Attention à bien découpler :
La manipulation d'un message applicatif.
Et l'interaction avec une socket (écrire un message dans une socket avec sérialisation préalable si nécessaire).
Lire un message.

Les différentes étapes de lecture (ou d'écrire) d'un message dansune socket ne doivent pas être éparpillées à différents endroits du code.