Grimmer Remy
Guichon Maxence

Projet Poker

Sommaire

AD 2014
Disponible sur : https://github.com/CapitaineCookies/ScholarNetworkPoker

I. Architecture

Pour avoir une meilleur découpe et lisibilité du code, nous avons mis en place une architecture adapté au projet.
Deux design pattern ont été combinés afin d'obtenir le meilleur résultat possible.

Le premier se base sur un pattern à état. L'avantage de ce design pattern est que chaque protocole est répartit dans un état, ainsi le code pour la distribution de nombres ne se mélange pas au code de l'échange de cartes. La répartition est mieux faite et il est bien plus simple de retrouver le code pour un protocole donné.

De plus les protocoles demandent généralement d'utiliser des variables qui leur son propres : mémorisation des ids reçus, mémorisation du nombre d'échanges... Chaque classe contient ces propres variables en plus de celles communes à toutes .

Le second design pattern est un pattern visiteur. Celui-ci est utilisé pour la réception de message. Étant donné qu'il existe un grand nombre de messages échangés dans le programme, nous avons pensé qu'il était judicieux d'utiliser ce pattern afin de ne plus recevoir tous les messages via la seule méthode receive. Avec ce pattern, chaque message a sa propre fonction receive. Nous n'avons plus qu'à implémenter les méthodes qui nous intéressent. Si d'autres messages sont reçus mais non traités, ils sont simplement ignorés et un message de log nous le signal.

En parallèle, nous avons profité de cette structure de donnée pour implémenter une base pour tous nos design pattern. C'est le rôle de la classe GameState. En effet, Certaines opérations sont redondantes entre chacun des GameStates. On exécute un GameState via la fonction start(). En interne, cette fonction déroule le protocole suivant :

- preExecute() : initialise certaines variables
- Execute() : réalise les actions spécifiques à l'état
- postExecute() : affecte les résultats obtenu aux variables globales

Entre chaque exécution, la possibilité de synchroniser tous les participants est offerte. Pour cela, il faut surcharger les méthodes makePreSyncExecute() et makePostSyncExecute() en retournant vrai.

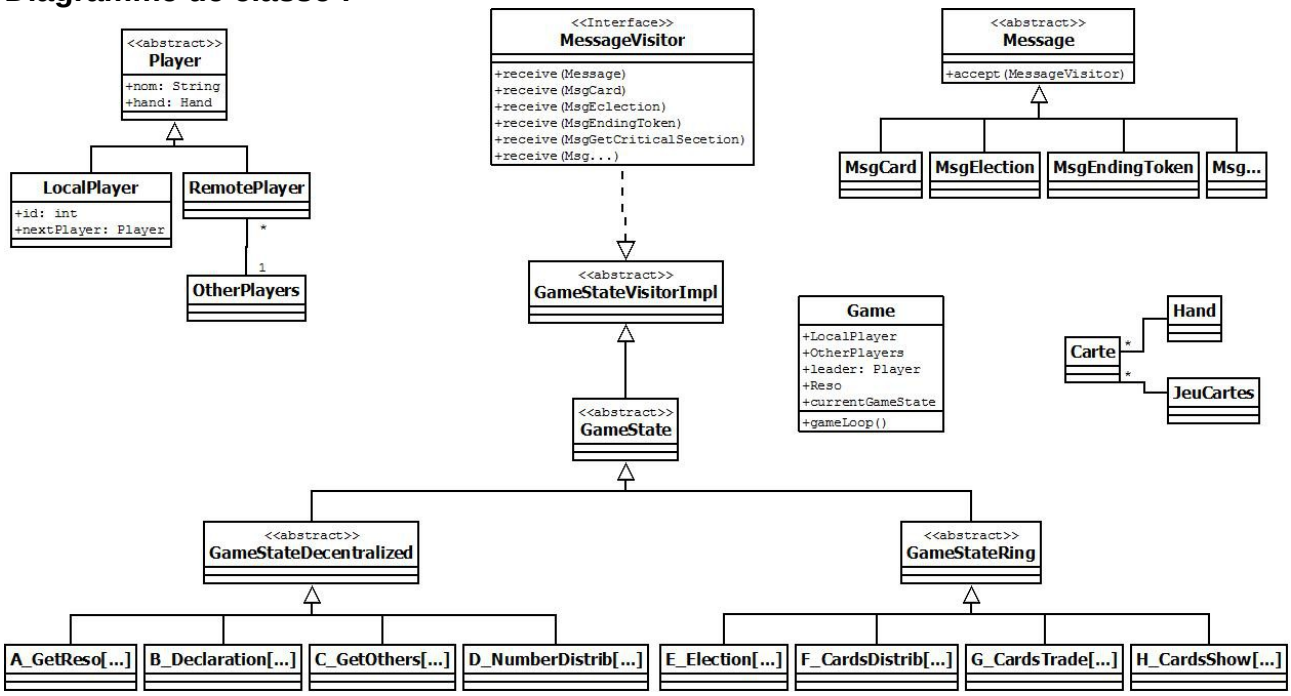
La synchronisation se fait différemment si on hérite de GameStateDecentralized ou GameStateRing. La première inonde les participants de messages $O(n^2)$ et la seconde utilise un algorithme de terminaison en anneau $O(n \log(n))$ mais nécessite que les joueurs soient répartis en anneau.

L'inconvénient, dû au fait que le programme est asynchrone et non fifo, est que l'on peut recevoir des messages d'un état avant le notre ou après le notre. Pour pallié ce problème, nous n'avons pas utilisé une pile comme dans le design pattern mais une Map. De plus, chaque message contient une énumération désignant quel est l'état qui doit recevoir le message. De cette façon, à la réception, les messages sont dirigés vers l'état correspondant.

Ainsi, il est possible de recevoir des messages pour l'élection même si l'état courant est la distribution de nombre, et cela sans qu'il n'y ai d'interférences.

Au final, l'implémentation d'un nouveau protocole ce fait très simplement par la surcharge d'une des classe filles de GameState : soit GameStateDecentraized pour une topologie quelconque, soit GameStateRing pour une topologie en anneau.
Ensuite il faut surcharger la fonction principale execute() et les receives des messages voulu. Il sera également demandé l'énum correspondant à l'état et l'énum de l'état suivant.

Diagramme de classe :



II. Les algorithmes mis en œuvre

1) Récupération des joueurs

Cet algorithme s'occupe de récupérer la liste de tous les participants.

Protocole :
- On commence par envoyer un message de signalement(MsgPlaying) en broadcast
- Lorsque l'on reçoit ce message, on signale en retour que l'on participe aussi (MsgPlayingToo) au joueur nous l'ayant envoyé
- Une fois le temps écoulé, on envoi en diffusion la liste des participant.
- La première liste reçu fait fois, les autres sont ignoré.
- Si notre nom ne figure pas dans cette liste, on quitte la partie.

Problèmes rencontrés :
- Lors du broadcast, on reçoit soit même son message. On l'enregistre comme les autres mais on n'envoie pas de message de réponse.
- Le message de réponse est obligatoire dans la mesure où il est possible qu'un joueur qui se connecte ne connaisse pas les joueurs qui le précède et peux envoyer sa liste avant eux.

Améliorations :
Dans la théorie, il n'est pas garanti que tous les joueurs aient la même liste.
Pour apporter cette garanti, il faudrait que tous les joueurs s'échange la liste reçu.
Tous les noms qui ne seraient pas en communs seraient supprimés.

Algorithme en pseudo code :

Variables utilisées par chaque joueur :
playersWantToPlay : tableau de String contenant les noms des joueurs qui veulent jouer
receivePlayers : booleen initialisé à FAUX

Messages :
MsgPlayers(players) : message contenant la liste des joueurs
MsgPlaying() : message d'information que le joueur est dans la partie
MsgPlayingToo() message de réponse à un MsgPlaying pour lui montrer que l'on joue aussi

Règle 1 : init
début
broadcaster le message MsgPlaying
dans 60 secondes, **broadcaster** le message MsgPlayers(playersWantToPlay)
fin

Règle 2 : i reçoit le message MsgPlayers(players) de j

```
début
        locked = VRAI
        Si !receivePlayers Alors
                receivePlayers = VRAI
                locked = FAUX
                otherPlayers.addAll(players)

                notifier fin de l'état
        Sinon
                locked = FAUX
                ignorer le message
        Fsi
fin

Règle 3 : i reçoit le message MsgPlaying de j
début
        Si !playersWantToPlay.contains(j) Alors
                playersWantToPlay.add(j)
                Si i != j Alors
                        envoyer MsgPlayingToo à j
                Fsi
        Fsi
fin

Règle 4 : i reçoit le message MsgPlayingToo de j
début
        playersWantToPlay.add(j)
fin
```

Propriétés :

propriété de vivacité :
un message de type MsgPlayers sera toujours reçu (Règle 2)
un message de type MsgPlaying sera toujours reçu (Règle 3)
un message de type MsgPlayingToo sera toujours reçu (Règle 4)

propriété de terminaison :
Un message MsgPlayers est émis au bout de 60 secondes ce qui termine l'algorithme.

2) Numérotation des joueurs

Cet algorithme s'occupe de donner des identifiants uniques aux joueurs

Protocole :

- Tous les joueurs génèrent un nombre aléatoire compris entre 0 et le nombre de joueurs. Ils envoient ensuite ce nombre aux autres participants.
- Chaque joueur attend de recevoir les nombres des autres joueurs.
- Une fois tous les messages d'id reçu, on vérifie si notre id entre en collision avec un autre joueur.

Si il y a une collision, on envoie un message de résolutions aux joueurs en conflit avec nous.
- On attend ensuite que les joueurs ayant un conflit nous envoi leur nouvel id.

Résolution de conflits :
Les messages de résolution de conflits contiennent un nombre aléatoire.
- On attend la réception de tous les messages de conflit qui nous sont destinés
- On envoi ensuite un message de syncro afin de s'assurer que toutes les personnes ont bien reçu notre message
- On calcul notre nouvel id de la façon suivante :

Parmi la liste des ids libres on prend la ième valeur

i étant le nombre d'id en conflit avant le notre + notre position dans le conflit pour mon id

exemple : monID = 3, lesAutresID = [0,0,2,3]

Les id dispos sont [0, 1, 3, 4]

Nombre d'id en conflit avant le mien : 2

Les valeurs que je peux prendre sont 3 et 4.

La première valeur est attribué au joueurs ayant le plus petit aléatoire et ainsi de suite dans l'ordre croissant.

- Une fois notre ID déterminé, on l'envoi aux autres joueurs.

En considérant que la résolution de conflits converge rapidement, la complexité en nombre de message est de l'ordre de O(n2).
La cause principale étant la topologie du réseau qui nécessite l'envoi de messages en diffusion.

La première version implémentée était basé sur le même principe.
Cependant la résolution de conflits se faisait en redonnant un nombre aléatoire parmi les nombres non choisis.
L'algorithme était relativement plus simple mais mettait plus de temps à converger à cause de conflits pouvant se répéter si les nouveaux nombres choisis étaient identiques.

Algorithme en pseudo code :

Variables utilisées par chaque joueur :

myID : id du joueur. Init à une valeur aléatoire

othersID : tableau d'entier contenant l'id des autres joueurs

nbIdChoice : entier, stocke le nombre de messages MsgIdChoice reçus

nbConflictCount : entier, stocke le nombre de messages MsgResolveConflict reçus

nbMsgConflictsResolved : entier contenant le nombre de conflits résolus

nbMsgConflicts : entier, stocke le nombre de messages MsgSyncConflict reçus

Messages :

MsgIdChoice(id) : message contenant un id

MsgResolveConflict(weight) : message contenant un nombre aléatoire pour la résolution de conflits

MsgSyncConflict() message de synchronisation

Règle 1 : init joueur i

début

envoyer MsgIdChoice(id de i) aux autres joueurs

fin

Règle 2 : i reçoit un message MsgIdChoice(id) de j

début

othersID[j] = id

nbIdChoice++

Si nbIdChoice == nombre d'aversaires **Alors**

copyOfOthersID = othersID

//conflit

Si l'id de j est dans othersID **Alors**

envoyer MsgSyncConflict aux joueurs en conflits avec j

attendre(nbMsgConflicts == nombre de joueurs en conflits avec j) ;

envoyer MsgResolveConflict() à tous les joueurs en conflits

Fsi

attendre (nbMsgConflictsResolved == nombre de conflits)

notifier la fin de l'état

Sinon

nbMsgConflictsResolved++

Fsi

fin

Règle 3 : i reçoit un message MsgResolveConflict(weight) de j

début

nbConflictCount++

othersMsgConflicts[j] = weight

Si nbConflictCount == nombre de joueurs en conflits avec j **Alors**

nbConflictCount = 0

envoyer MsgSyncConflict() à tous les joueurs en conflits

attendre(nbMsgConflictsResolved == nombre de joueurs en conflits avec j)

Si il y a encore un conflit **Alors**

```
                    Pour tous les joueurs en conflits j2 avec j Faire
                        envoyer MsgResolveConflict à j2
                    Fait
                Sinon
                    i.id = nouvelle ID valide
                    nbMsgConflictsResolved++ ;

                    Pour tous les joueurs adverse de j Faire
                        envoyer MsgIdChoice(i.id)
                    Fait
                Fsi
fin

Règle 4 : i reçoit d'un message  MsgSyncConflict de j
début
        nbMsgConflicts++ ;
fin
```

Propriétés :

propriété de vivacité :

un message de type MsgIdChoice sera toujours reçu (Règle 2)
un message de type MsgResolveConflict sera toujours reçu (Règle 3)
un message de type MsgSyncConflict sera toujours reçu (Règle 4)

propriété de terminaison :

3) Élection

Cet algorithme s'occupe de déterminer le leader.

L'algorithme que nous avons implémenté pour l'élection est l'algorithme LeLann-Chang-Roberts (LCR). Pour cela, nous avons utilisé une structure en anneau où chaque joueur connaît celui qui le succède grâce à son numéro attribué pendant la phase de numérotation. Ainsi, un joueur j à pour suivant le joueur j1 si l'id du joueur j1 est immédiatement supérieur à l'id de j.

Algorithme en pseudo code :

```
Variables utilisées par chaque joueur :
participant : booléen initialisé à FAUX
leader_id : l'id du leader initialisé à null
suivant(j) : joueur suivant de j

Messages :
```

MsgElection(id) :message d'élection
MsgLeader(id) : message contenant l'id du leader

Règle 1 : *le joueur j déclenchement d'une élection*
début
    envoyer MsgElection(j.id) à suivant(j)
    participant = VRAI
fin

Règle 2 : *le joueur j1 reçoit un message MsgElection(j.id)*
début
    Si (j1.id < j.id) **Alors envoyer** MsgElection(j.id) à suivant(j1) **Fsi**
    Si (j1.id = j.id) **Alors envoyer** MsgLeader(j1.id) à suivant(j1) **Fsi**
    Si ((j1.id > j.id) & (!participant)) **Alors envoyer** MsgElection(j1.id) à suivant(j1) **Fsi**
    participant = VRAI
fin

Règle 3 : *le joueur j1 reçoit un message MsgLeader(j.id)*
début
    leader_id = j.id
    Si (j1.id != j.id) **Alors envoyer** MsgLeader(j.id) au joueur suivant j1 **Fsi**
fin

Cet algorithme a une complexité en terme de messages de O(nlog(n)) .

Propriétés :

propriété de vivacité :
    un message de type MsgElection sera toujours reçu (Règle 2)
    un message de type MsgLeader sera toujours reçu (Règle 3)

propriété de terminaison :

    1) Lorsqu'une élection est déclenché, un message d'élection est émis(MsgElection).

    2) Lors de la réception de ce message, 2 cas se présente :
        _le message est relayé au joueur suivant avec un message MsgElection
        _on indique au joueur suivant qu'on est le leader avec un MsgLeader

Si on reçoit un message MsgElection, on retourne à l'étape 2, sinon on passe à l'étape 3.

    3) Lors de la réception d'un MsgLeader, 2 cas se présentent :
        _l'id reçu dans ce message est le notre, dans ce cas l'algorithme se termine.
        _on fait suivre le message MsgLeader au joueur suivant (= on recommence cette étape).

Pour que l'algorithme se termine, il faut s'assurer qu'on ne boucle pas à l'infini sur l'étape 2

et sur l'étape 3.

Pour ne pas boucler sur l'étape 2, il faut qu'au moins un message MsgLeader soit émis, en d'autres termes, cela signifie qu'au moins un joueur se déclare comme leader.

Preuve : l'élection étant basé sur l'id du joueur qui est un entier unique, on a obligatoirement un id qui est plus grand que tous les autres et donc il existe forcément un leader.

Pour ne pas boucler sur l'étape 3, il faut qu'il existe un joueur dont l'id est celui qui est contenu dans le message MsgLeader

Preuve :
Le message MsgLeader contient l'id d'un joueur. Comme on utilise une structure en anneau et qu'on fait suivre le message MsgLeader au joueur suivant, tous les joueurs recevront ce message, en particulier le joueur dont son id est égal à l'id du message.

4) Distribution des cartes

Le leader définit précédemment récupère le paquet de cartes et en distribue 5 une par une à tous les joueurs. Pour cela on envoie un message MsgCard contenant la carte.

Algorithme en pseudo code :

Variables utilisées par chaque joueur :

Messages :
MsgCard(c) : message contenant une carte c

Règle 1 : Le leader distribue les cartes
début
    Pour i de 0 à 5 **Faire**
        Pour tous les adversaires j **Faire**
            envoyer MsgCard(deck.nvlleCarte()) à j
        Fait
    Fait

    envoyer MsgCard(deck.nvlleCarte()) à soi-même
fin

Règle 2 : Le joueur j recoit un message MsgCard(c)
début
    j.ajouterCarte(c)

    Si j a reçu 5 carte **Alors**
        Notifier la fin de l'état

```
        Fsi
fin
```

La complexité en nombre de messages est de O(5n) = O(n)

Propriétés :

propriété de vivacité:
un message de type MsgCard sera toujours reçu (Règle 2)

propriété de terminaison :
on envoie 5 cartes à tous les joueurs et l'algorithme se termine quand on reçoit 5 cartes.

5) Échange de cartes

Cette algorithme implémente une section critique.
Cette section critique est prise lorsqu'un joueur donne des cartes
et est relâché lorsque le joueur a effectivement reçu toutes ces nouvelles cartes.

Protocole :

Coté joueur non leader :

- Un joueur commence par demander la section critique en envoyant un message.
- Une fois cette section critique obtenue, il envoie un certain nombre de cartes au leader.
- Il attend de recevoir toutes ces nouvelles cartes.
- Une fois qu'il a toutes ces cartes, il envoie un message pour relâcher la SC.
- Il peut recommencer l'opération encore une fois.

Messages :

- réception de MsgObtainCriticalSection : On relâche l'attente de la SC coté joueur. Cela entraîne l'envoi des cartes à échanger (MsgTradeCards).
- réception de MsgCard : On ajoute la carte à la main du joueur et on le notifie au thread principale

Coté leader :

- Lorsqu'il commence, il lance un second thread. Le premier aura le même rôle que les autres joueurs. Le second gérera la section critique.

  (Le protocole qui suit décrit le second thread)
- Il commence par prendre la SC interne.
- Il prend la première demande de SC par un joueur ou attend une demande si il n'y en a pas.
- Il envoie la SC à ce joueur (MsgObtainCriticalSection)
- boucle à la prise de SC

Messages :

- la réception de "MsgReleaseCriticalSection" : provoque le relâchement de la SC
- la réception de "MsgGetCriticalSection" : ajoute le joueur a la liste d'attente pour la SC
- la réception de "MsgTradeCards" : provoque l'échange de cartes : Ajout des carte reçu au jeu de carte et envoi de nouvelles cartes.

Variables utilisées par chaque joueur :
nbTradeMade : nombre d'échanges déjà effectués. Init à 0
NbTradeMadeFromLastReceiveToken : nombre d'échange fait par le joueur depuis la dernière réception du jeton. Init à 0

Messages :

MsgEndingToken : message contenant le jeton enregistrant les informations d'échange.

MsgGetCriticalSection : message de demande de section critique.

MsgObtainCriticalSection : message spécifiant qu'il a obtenu la section critique.

MsgReleaseCriticalSection : message spécifiant qu'on relâche la section critique.

MsgCard : message contenant la nouvelle carte pour le joueur

MsgTradeCards : message contenant les cartes à échanger.

MsgTradeEnd : message signifiant la fin des échanges.

Règle 1 : faire un échange avec leader

début

envoyer MsgGetCriticalSection() à leader
attendre MsgObtainCriticalSection() de leader
envoyer MsgTradeCards(liste de cartes à échanger) à leader
tant-que (nombre de carte reçu < nombre de carte envoyer)
attendre MsgCard(carte) de leader
ajouter la carte à la main
nombre de carte reçu = 0
nbTradeMade + 1
NbTradeMadeFromLastReceiveToken + 1
envoyer MsgReleaseCriticalSection() à leader

fin

Règle 2 : i reçoit un message MsgCard() de j

début

nombre de carte reçu + 1

fin

Règle 3 : leader reçoit un message MsgGetCriticalSection de j

début

Si section critique est libre
envoyer MsgObtainCriticalSection() à j
Sinon
mettre j en liste d'attente

fin

Règle 4 : leader reçoit d'un message MsgTradeCards<liste de carte> de j

début

ajouter les cartes reçus au jeu de carte
Tant-que nombre de carte envoyé < nombre de cartes reçus
envoyer MsgCard(première carte du jeu de carte) à j
nombre de carte envoyé + 1

fin

Règle 5 : leader reçoit un message MsgReleaseCriticalSection de j

```
        Si liste d'attente non vide
                envoyer  MsgGetCriticalSection() à j
Règle 6 : i reçoit un message MsgEndingToken de j
début
        Si on a fait des échanges depuis le dernier passage du token
                invalider le token
                enregistrer le nombre d'échange fait
        Si on a pas déjà atteint le nombre d'échange max
                on peut faire règle 1 si on le souhaite
fin

Règle 7 : i reçoit un message MsgTradeEnd de j
début
        Notifier la fin de l'état
fin
```

6) Affichage des cartes

Cet algorithme permet à chaque joueur d'afficher ses cartes.

Protocole :

- un joueur j1 montre une de ses cartes (à l'initialisation, c'est le leader)
- on attend que tous les autres joueurs reçoivent la carte en faisant passer un jeton
- Lorsque le jeton a fait un tour, le joueur suivant au joueur j1 montre une carte etc.
- On s'arrête lorsque le nombre de cartes envoyé est égal au nombre de carte par joueur multiplié par le nombre de joueur

Algorithme en pseudo code :

```
Variables utilisées par chaque joueur :
countCards : nombre de cartes initialisé à 0
Messages :
MsgCardWithNextPlayer(c, p) : message contenant une carte et le joueur suivant
MsgReceiveToken(id) : message de synchronisation

Règle 1 : init : le leader montre une carte
début
        Pour tous les adversaires j Faire
                envoyer MsgCardWithNextPlayer(getCard(), suivant()) à j
```

```
        Fait
        countCards++;
        Si (countCards == 5 * nombre d'aversaire) Alors
                Notifier fin de l'état
        Fsi
fin

Règle 2 : i reçoit un message MsgCardWithNextPlayer(cardWithNextPlayer,player) de j
debut
        countCards++ ;
        Si (countCards == 5 * nombre d'adversaire) Alors
                notifier fin de l'état
        Sinon si(i == cardWithNextPlayer.getNextPlayer())
                envoyer MsgReceiveToken(i.getID()) à suivant(i)
        Fsi
fin

Règle 3 : i reçoit un message MsgReceiveToken(id) de j
debut
        Si (i.getId == id) Alors
                Pour tous les adversaires j Faire
                        envoyer MsgCardWithNextPlayer(getCard(), suivant(i)) à j
                Fait
        Sinon
                envoyer MsgReceiveToken(id) à suivant(i)
        Fsi
fin
```

On envoie 2n-1 messages par carte. L'envoi de la carte à tous les autres joueurs demande n-1 messages, et le jeton passe par tous les joueurs soit n messages.