« Coinch'UTC» Application de coinche multi supports

Rapport final IA04-P13

LANCELOT Simon

WANG Yiou

DEVAUX Bruno

SOMMAIRE

INTRODUCTION

Dans le cadre de l'UV IA04 enseignée à l'UTC, nous devions réaliser un projet de développement informatique intégrant les connaissances acquises en cours et en séance de TP sur la gestion des systèmes multi-agents.

Nous étions tous les trois déjà d'un projet parallèle dans l'UV « NF28 – Initiation aux Interfaces Homme-Machine » et avons remarqué que le but de ce projet cadrait parfaitement avec le développement d'un environnement multi-agent : la création d'une application mobile Android de jeu de Coinche en réseau. Au cours du développement, nous avons étendu le support mobile initial afin de développer également une interface PC.

Ce rapport présentera la conception retenue pour réaliser cette application ainsi que les choix d'implémentations : Agents et Behaviours Jade.

CAHIER DES CHARGES

Dans le cadre du projet NF28, nous devions réaliser un cahier des charges en amont avant de se lancer dans la conception de l'application. Après l'enquête réalisée auprès des utilisateurs, nous avions conclu que le besoin le plus important pour notre application était la restitution de l'ambiance présente lors d'une partie de cartes réelles. La partie la plus importante de ce projet était donc de faire en sorte que l'utilisateur s'adapte facilement et apprécie de jouer une partie via notre application Android. Cela rentrait donc bien dans le cadre de l'UV NF28 puisque nous devions optimiser l'interface dans ce but.

Un autre objectif de notre application était la gestion du tournoi organisé chaque semestre par l'association de Coinche à l'UTC.

Les fonctionnalités que nous avions prévu de développer étaient :

- Visualisation des règles du jeu
- Création d'une partie
- Rejoindre une partie
- Sélectionner les paramètres de jeu
- Jouer une partie
- Espace Coinch'UTC
- Statistiques et fiche profil des joueurs
- Chat
- Visualisation des joueurs connectés
- Création d'une application
- Connexion à l'application

Vis-à-vis de notre cahier des charges initial et de notre maquette d'avant conception, notre projet final est sensiblement différent, et ce pour plusieurs raisons : nous avons fait le choix d'étendre la porté de l'application avec une interface PC, et nous nous sommes concentrés sur la réalisation d'une partie fonctionnelle au détriment des autres objectifs, principalement par manque de ressource et de temps de développement (la partie Chat ayant néanmoins également été implémentée).

LA BASE DE CONNAISSANCES

Dans notre application, nous utilisons une base de connaissance RDF afin de pouvoir gérer les comptes des utilisateurs. Chaque utilisateur est défini par un nom (qui peut être un surnom) et il possède plusieurs attributs, dont au moins deux sont obligatoires : le **login** et le **mot de passe**.

Dans la base de connaissance, chaque utilisateur est donc défini de la même manière que l'exemple ci-dessous :

```
projet:Simon
      a foaf:Person ;
      foaf:name "Simon Lancelot" ;
    foaf:mbox <mailto:simon.lancelot@gmail.com> ;
      foaf:identifiant "slancelo";
      foaf:mdp "curitibaA13" .
```

FIGURE 1 : EXEMPLE UTILISATEUR

Les deux attributs essentiels sont l'identifiant et le mot de passe, car ce sont ces attributs qui sont vérifiés lors de la connexion à l'application. De plus, pour le moment, ce sont les deux seuls attributs que le joueur remplit lors de la création de son compte.

Cette base de connaissance est donc utilisée en lecture et en écriture, lorsque l'on veut vérifier si un utilisateur possède un compte (lecture) ou lorsque l'on veut créer un compte (écriture)

LE *LOGAGENT*

Comme son nom peut le laisser deviner, le **LogAgent** permet d'accéder à la base de connaissances, que ce soit pour trouver un compte existant ou pour en créer un nouveau.

On crée donc le modèle en allant récupérer les assertions présentes dans notre fichier **baseDonnees.txt** et on y accède ensuite lors de l'interaction avec les boutons de connexion ou de création de compte.

CONNEXION A L'APPLICATION

Pour vérifier si un utilisateur est bien inscrit dans la base de connaissances, on a utilisé une requête de type *ask*, qui vérifie si, pour un même utilisateur, le login et le mot de passe sont bien les mêmes dans la base de connaissance que ceux entrés dans les zones de texte prévues à cet effet.

La requête utilisée est la suivante :

```
String queryString =
        "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>"+
                "PREFIX foaf: <http://xmlns.com/foaf/0.1/>" +
                "PREFIX projet:  <http://projet/>"+
                "ASK" +
                "WHERE { ?pers foaf:identifiant '"+login+"'. ?pers foaf:mdp '"+mdp+"'}";
```

FIGURE 2 : EXEMPLE REQUETE SPARQL

Si la requête renvoie *true*, ce qui signifie que le joueur est bien présent dans la base de données, le LogAgent va créer un nouvel agent de type **Joueur** (voir plus loin), qui aura comme *LocalName* le login de l'utilisateur.

CREATION D'UN COMPTE

Lors de la création d'un compte, l'utilisateur renseigne trois informations : son nom (surnom), son login et son mot de passe.

Une fois que le joueur clique sur le bouton de création du compte, le **LogAgent** va écrire dans le fichier les informations concernant ce nouveau compte en respectant la syntaxe correspondant à un utilisateur, de la manière suivante :

```
String nomFichier = "baseDonnees.txt";
FileWriter f = new FileWriter(nomFichier,true);
BufferedWriter output = new BufferedWriter(f);
output.write("\nprojet:"+arg0.getParameter(0) +
        "\n a foaf:Person ; "+
"\n foaf:identifiant \""+arg0.getParameter(1)+"\";"+
"\n foaf:mdp \""+arg0.getParameter(2)+"\".");

output.flush();
output.close();
```

FIGURE 3 : CODE DE CREATION D'UN COMPTE

LOGWINDOW

Durant toute la partie de l'application qui concerne les connexions et créations de compte, l'interface est gérée par une fenêtre appelée **LogWindow**, qui est implémentée par le LogAgent et qui interagit avec lui lors de clics sur les boutons, par exemple.

Le LogAgent étend donc la classe des ***GuiAgent***, ce qui lui permet de répondre à des *guiEvent* lors de l'interaction avec la fenêtre.

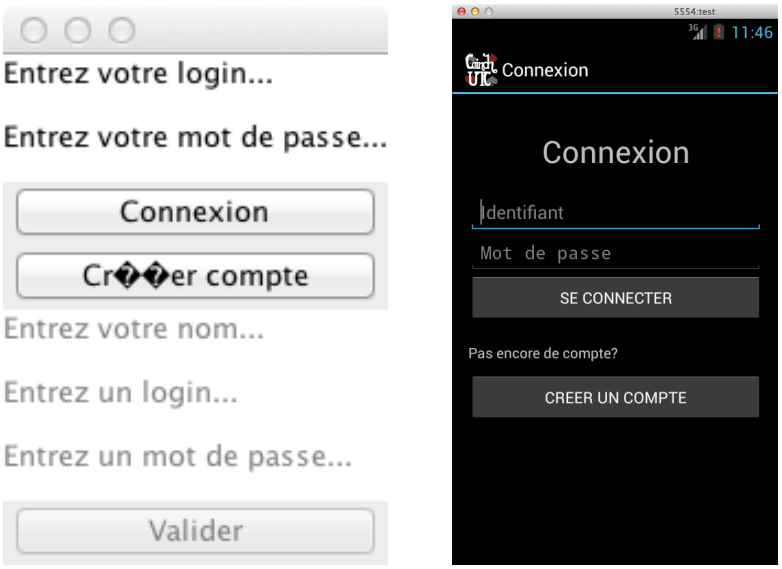

FIGURE 5 : INTERFACES DE CONNEXION PC ET ANDROID

L'AGENT *JOUEUR*

Lorsqu'un utilisateur se connecte à l'application, un nouvel agent de type **Joueur** est donc créé, comme expliqué précédemment. Ce joueur est également un GuiAgent et, à sa création.

Ce joueur implémente, comme LogAgent, une fenêtre appelée **MainWindow** dans notre projet, qui va lui permettre de réaliser les actions au fur et à mesure de la partie.

Lors de sa création, on lui ajoute plusieurs Behaviours différents et ce joueur s'enregistre dans les « pages jaunes » comme étant capable de chater avec d'autres joueurs.

Une partie de coinche est décomposée en plusieurs étapes, et chaque étape ne peut commencer que lorsque l'étape précédente se termine. Ainsi, les joueurs suivent un comportement qui est séquentiel, ce que nous avons utilisé dans notre application. En effet, le comportement général est séquentiel et nous avons ainsi ajouté plusieurs comportements correspondants aux différentes étapes d'une partie de coinche.

LE CHATBEHAVIOUR

Lorsque le joueur se connecte à l'application, il a deux possibilités : soit rejoindre une partie soit « chatter » avec les autres joueurs connectés.

Le ChatBehaviour est donc le premier comportement implémenté pour un joueur, et il étend la classe **Behaviour**. En effet, c'est un comportement cyclique mais on souhaite pouvoir l'arrêter pour passer à l'étape suivante.

Comme nous l'avons vu dans l'architecture générale du projet, nous avons trois types d'agents différents : le LogAgent, les Joueurs et l'agent Partie. C'est important de garder cela à l'esprit car nous allons avoir sans arrêt des échanges de messages entre chaque joueur et l'agent Partie. Ici, la communication entre ces deux types d'agent va consister tout d'abord en un message du joueur vers **Partie** lorsqu'il clique sur *Rejoindre Partie*. Ce message va permettre à l'agent Partie d'enregistrer le joueur dans un tableau d'AID qui a une taille de 4 (car il y a forcément 4 joueurs sur une partie). L'agent Partie va pouvoir utiliser ces AID pour envoyer, plus tard, les messages précisément aux participants de la partie. Ensuite, l'agent Partie va envoyer un message aux joueurs pour leur signifier que la partie va commencer et que le chat est terminé.

Ce message va être envoyé lorsque quatre joueurs auront cliqué sur *Rejoindre Partie*, ce qui signifie que tous les joueurs sont réunis pour commencer la partie. Comme expliqué dans le cahier des charges, lorsque la partie commence, les joueurs ne sont plus autorisés à discuter entre eux.

Avant cela, les joueurs vont pouvoir communiquer entre eux, grâce à l'interface composée de deux zones de texte et d'un bouton *Envoyer*. On va retrouver ici l'intérêt des « pages jaunes », et on peut rappeler que les joueurs sont tous enregistrés comme étant capables de chatter.

En effet, la première chose nécessaire avant d'envoyer un message est de savoir à qui il faut l'envoyer, donc ajouter un destinataire. Lorsqu'un joueur va envoyer un message, la méthode **setReceiver** suivante va être appelée :

```
private void setReceiver() {
    int ind=0;
    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("Chat");
    template.addServices(sd);
    try {
        DFAgentDescription[] result = DFService.search(this, template);
        if (result.length > 0) {
            for (int i=0;i<result.length;i++)
            {
                if (!result[i].getName().equals(this.getAID()))
                {
                    receivers[ind] = result[i].getName();
                    ind++;
                }
            }

        }
        else System.out.println("Erreur lors de la creation des receivers");
    }
    catch(FIPAException fe) {  }
    ind = 0;
}
```

FIGURE 6 : CODE DE LA METHODE SETRECEIVER()

Cette méthode permet de récupérer les autres joueurs, en faisant appel à leur capacité de chatter, à l'aide des pages jaunes. Il faut bien sur différencier l'émetteur du récepteur, et on traite au niveau de l'interface graphique afin d'obtenir le résultat suivant :

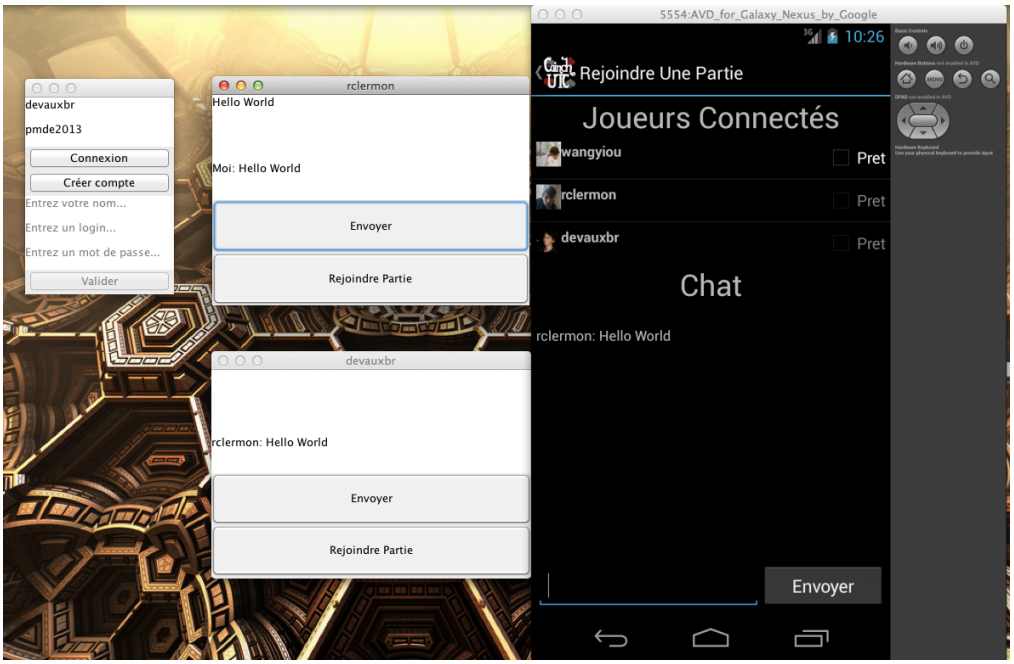

FIGURE 7 : INTERFACES DE CHAT PC ET ANDROID

L'implémentation du chat est très importante pour notre application car elle répondait à un besoin récurrent de la part des futurs utilisateurs, lors de la diffusion d'un sondage, c'est-à-dire de recréer l'ambiance autour d'une table de jeu.

A leur arrivée dans l'application, les utilisateurs ne sont donc pas encore affectés à la partie. Ils le sont en cliquant sur *Rejoindre Partie*, et lorsque les quatre joueurs ont cliqué sur ce bouton, le chat s'arrête (la méthode *done* du Behaviour retourne *true*) et on passe alors au Behaviour suivant.

LE RECUPBEHAVIOUR

Une fois que tous les joueurs ont rejoint la partie, le joueur entre dans son *RecupBehaviour*. Ce comportement consiste en une communication avec l'agent **Partie** et étend également la classe Behaviour.

En effet, ce dernier va envoyer à chaque joueur 8 messages contenant chacun une carte. Les cartes sont des instances de la classe **Carte**, et contiennent leur valeur (de 7 à 14, ce qui correspond aux cartes entre le 7 et l'As), leur couleur (Pique, Cœur, Carreau et Trèfle), une chaîne de caractère qui est le chemin d'accès à l'image de la carte et une chaîne de caractère correspondant au nom du joueur qui a joué la carte (cela est utilisé dans **JouerBehaviour**). Elles sont transmises sous le format JSON, grâce à la sérialisation.

A chaque message reçu (en ayant vérifié évidemment le performatif pour être sur que l'on reçoit le bon type de message), la chaîne est dé-sérialisée depuis la chaîne de caractères en carte. Cette carte est ensuite ajoutée dans la main du joueur, qui n'est autre qu'un tableau de cartes.

Étant donné que dans une partie de coinche, chaque joueur possède au début 8 cartes dans sa main, lorsque le joueur a reçu 8 messages contenant une carte, le behaviour se termine.

La liaison entre l'agent et l'interface qu'il implémente, **MainWindow**, permet d'afficher les images dans la fenêtre de chaque agent des cartes qu'il reçoit. On réalise cela à l'aide d'un ***PropertyChangeListener*** sur la fenêtre du joueur.

A la fin de ce comportement, chaque joueur possède donc une main et y a accès graphiquement grâce à sa fenêtre.

LE ANNONCEBEHAVIOUR

L'étape suivante correspond au(x) tour(s) d'annonces. Chaque joueur, après avoir pris conscience de sa main, va pouvoir annoncer le nombre de points qu'il pense réaliser et avec quel atour pour couper. On a donc ajouté un *AnnonceBehaviour*, qui étend là encore Behaviour.

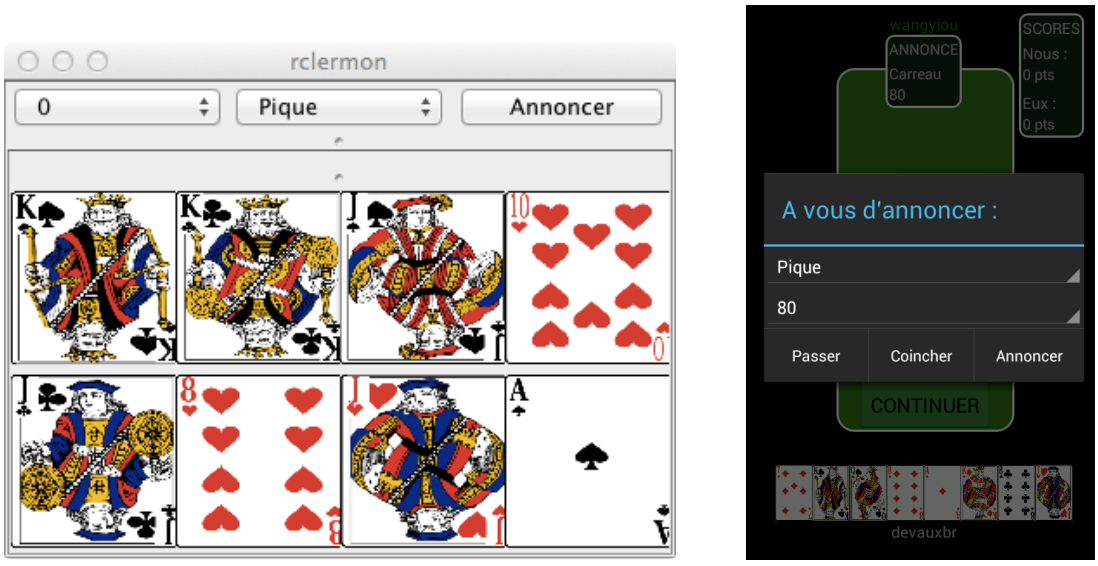

FIGURE 8 : INTERFACES D'ANNONCE PC ET ANDROID

Là encore, ce comportement met en jeu la communication entre l'agent **Partie** et les joueurs. En effet, les annonces doivent se faire un par un et dans l'ordre dans lequel ils vont commencer la partie. Ainsi, l'agent Partie va envoyer un seul message au premier joueur (on a un compteur qui bloque la possibilité d'envoyer plusieurs messages) lui indiquant que c'est à son tour de commencer à annoncer.

Lorsque le joueur reçoit le message, on va modifier son interface afin qu'il puisse annoncer, puis, lorsqu'il envoie le message à Partie, il revient dans l'état initial, c'est-à-dire qu'il ne peut plus annoncer. Les annonces sont également sérialisées sous format JSON, ce sont des instances de la classe *Annonce*, qui a pour attributs la valeur de l'annonce, la couleur de celle-ci ainsi qu'un tableau d'AID correspondant à l'équipe qui est maître de l'annonce.

Une fois que le(s) tour(s) d'annonce sont terminés, Partie va envoyer un message à chaque joueur, en utilisant un autre performatif, afin de leur signifier que cette étape est terminée. Le comportement correspondant aux annonces va donc se terminer également.

LE JOUERBEHAVIOUR

L'étape qui suit correspond au jeu en lui-même. On va là encore étendre la classe Behaviour. Comme pour le comportement précédent, l'agent Partie va envoyer des messages à chaque joueur dans l'ordre pour leur signifier que c'est à eux de jouer.

Une fois le message reçu, là encore comme avant, on « écoute » ce que le joueur joue à l'aide de l'interface graphique, puis le joueur va envoyer un message sous forme JSON à Partie pour lui signifier la carte qu'il a joué. Lorsque la carte est jouée, elle disparaît de l'interface graphique.

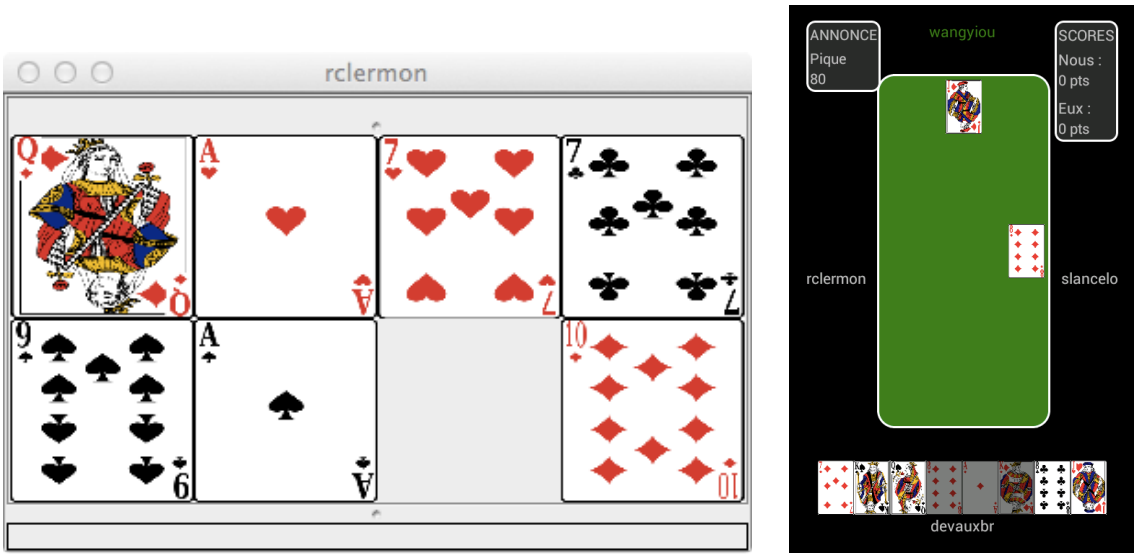

FIGURE 9 : INTERFACES DE JEU PC ET ANDROID

Étant donné que les joueurs possèdent leur propre interface sur différents matériels, il est nécessaire qu'ils aient accès aux cartes qui sont jouées. Ils ont donc chacun, dans leur interface graphique, un tapis de jeu avec la photo du joueur et de la carte qu'il vient de jouer. Pour mettre à jour ce tapis, une fois que Partie reçoit la carte jouée, il va envoyer un message à tous les joueurs contenant la carte jouée. A cette carte, on a affecté dans son attribut *login* le nom du joueur qui l'a envoyé, en récupérant le *sender* du message reçu par l'agent Partie. Ainsi, grâce au ***PropertyChangeSupport***, le joueur va pouvoir mettre à jour son tapis de jeu.

Un autre type de message (avec un autre performatif) peut être reçu de la part de Partie par un joueur. Ce message indique qu'une manche est terminée mais que la partie continue, car aucune des deux équipes n'a atteint le score de 1000. Ainsi, si un message de ce type est reçu, (performatif CONFIRM ici), l'agent joueur recrée un comportement séquentiel similaire à celui que nous venons d'expliquer, à l'exception du fait que l'on ne va pas retourner dans le chat. On va donc ajouter un RecupBehaviour, un AnnonceBehaviour et un *JouerBehaviour*.

Une fonctionnalité que nous avons commencée à implémenté mais que nous n'avons pas eu le temps de mener à son terme pourrait être par la suite ajoutée à ce comportement. Cette fonctionnalité serait de pouvoir restreindre la carte que les joueurs peuvent jouer, en fonction de ce qu'il y a déjà sur le tapis.

L'AGENT *PARTIE*

Cet agent, dont on a déjà parlé précédemment, est, comme le *LogAgent*, créé dès le lancement de l'application. De la même manière que les agents de type *Joueur*, Partie va avoir un comportement séquentiel, cohérent avec les différentes étapes d'une partie de coinche.

Lors de sa création, il va également créer un jeu de 32 cartes complet, avec les différentes valeurs et couleurs, sous forme d'une *ArrayList<Carte>*.

LE SUBSCRIBEBEHAVIOUR

Initialement, cet agent possède un sous comportement appelé *SubscribeBehaviour*. Dans ce comportement Partie attend de recevoir des messages provenant des joueurs, afin de les enregistrer comme participants. Ce comportement étend aussi la classe Behaviour.

Le nombre maximal de joueurs est de 4. Partie possède, comme on l'a déjà vu, un tableau d'AID de taille 4 donc, permettant d'enregistrer les AID des 4 joueurs s'inscrivant auprès de lui. Ensuite, ce qu'il faut noter est que la coinche se joue en deux équipes de deux joueurs. Pour définir les équipes, l'agent Partie prend en compte l'ordre d'inscription auprès de lui, et va enregistrer dans deux tableaux d'AID, *equipe1* et *equipe2* les AID correspondants. L'ordre choisi est le suivant :

Joueurs 1 et 3 → équipe 1 et joueurs 2 et 4 → équipe 2

Cet ordre respecte les règles de la coinche, qui veut que l'on se positionne en face de son partenaire, donc de façon ½.

Une fois que 4 joueurs ont envoyés un message (lors de leur clic sur *Rejoindre Partie* pour rappel) à l'agent Partie et qu'ils sont enregistrés, ce behaviour se termine, et on peut passer à la distribution des cartes. Juste avant de se terminer, Partie envoie un message aux joueurs pour leur signifier que le chat n'est plus autorisé.

LE DISTRIBUTIONBEHAVIOUR

Ce behaviour, qui permet de distribuer des cartes aux 4 joueurs, étend cette fois la classe *OneShotBehaviour*. En effet, on ne va distribuer, pour une manche, qu'une seule fois les cartes.

Ce behaviour n'attend pas de message pour agir, puisqu'il va envoyer à chaque joueur 8 cartes, par message sous forme de JSON là encore. Il récupère les destinataires des messages à partir du tableau de 4 AID dans lequel on a enregistré précédemment les joueurs participants à la partie.

Pour être sûrs que les cartes ne seront pas envoyées en double, on utilise le jeu créé précédemment et on va récupérer pour chaque carte distribuée, un indice aléatoirement entre 0 et la taille de l'ArrayList correspondant au jeu. Une fois cet indice généré, à l'aide de *Random*, on récupère la carte puis on utilise la méthode *remove* de l'ArrayList. Ainsi, au début on cherchera l'indice entre 0 et 32, puis 0 et 31, etc. tout en étant sûrs que la carte choisie ne l'a pas déjà été puisqu'on les retire ensuite.

Une fois les 32 messages envoyés (un message = 1 carte et 8 messages par joueurs, en comptant 4 joueurs), ce behaviour se termine et on peut passer à l'étape des annonces.

LE ANNONCEBEHAVIOUR

Lorsque l'on arrive dans ce comportement, l'agent Partie va envoyer un message tout d'abord au premier joueur. Pour éviter des problèmes au niveau de l'interface du joueur (qui change avec le PropertyChangeSupport), on utilise un compteur pour empêcher Partie d'envoyer plus d'un message à la fois.

Une fois le message envoyé, on attend un message de la part du joueur en question, en vérifiant bien que le *msg.getSender* corresponde au joueur auquel on a envoyé le message. Pour cela, on utilise encore une fois le tableau d'AID (appelé **joueurs**). Ainsi, on va vérifier si *msg.getSender* correspond, dans le premier cas, à **joueurs[0]**, 0 étant en fait un compteur également, incrémenté par la suite.

Lorsque l'on reçoit le bon message, on le dé-sérialise (on rappelle que les annonces sont envoyées sous format JSON) et on étudie l'annonce reçue.

L'agent Partie contient un attribut correspondant à l'annonce en cours. Pour qu'une annonce reçue remplace l'annonce en cours, il faut qu'elle soit plus élevée, sinon elle n'est pas traitée. Si c'est le cas, l'annonce en cours est remplacée par la nouvelle annonce, et on répète ensuite l'opération pour le deuxième joueur, et ainsi de suite.

Dans les règles que nous avons implémentés, nous considérons que les annonces se terminent lorsque 3 joueurs passent (annoncent un score de 0) ou qu'un joueur annonce un Capot (qui signifie que l'équipe va remporter tous les plis de la partie).

Une annonce a comme attribut, outre sa valeur et sa couleur, l'AID du joueur qui en est maître, afin de pouvoir déterminer quelle équipe doit réaliser le score annoncé.

Lorsque ce comportement se termine, on peut passer au jeu en lui-même.

LE TOURBEHAVIOUR

Le jeu en lui-même est implémenté en utilisant un behaviour appelé *TourBehaviour* qui étend Behaviour lui aussi. Comme son nom l'indique, ce comportement permet de jouer un tour, ce qui signifie que chaque joueur, à la fin de ce comportement, aura joué une carte.

L'agent Partie va donc, dans son comportement séquentiel, avoir 8 sous comportements qui seront des TourBehaviour.

De la même façon que pour les annonces, on va envoyer un message à chaque joueur un par un, en attendant sa réponse avant d'envoyer au joueur suivant.

La réponse en question sera un message contenant une chaîne JSON correspondant à la carte jouée par un agent Joueur. Lorsque Partie reçoit un message contenant une carte, il affecte à cette carte le nom du joueur qui l'a envoyée (à l'aide de *msg.getSender*), puis envoie cette carte en message aux 4 joueurs, afin que ceux-ci puissent mettre à jour leur interface graphique, en faisant apparaître sur le tapis de jeu la carte jouée à côté de la photo du joueur.

Une fois que les 4 joueurs ont envoyés leur carte à Partie, ce dernier détermine qui est le gagnant et le score réalisé sur le plis, grâce à une fonction récupérant les points de chaque carte, et la priorité des cartes les unes par rapport aux autres (il a fallu faire attention aux cas où les cartes sont « coupées » par des atouts). Le score de l'équipe dont le joueur a remporté le plis est modifié en conséquence.

Enfin, il change l'ordre des AID dans le tableau des AID, car le tour suivant doit repartir du gagnant de ce tour-ci. Par exemple, si le joueur 3 gagne le tour, il passera pour le tour suivant, dans ce tableau, en qualité de joueur 1, et il faut décaler les autres également, de façon à respecter l'ordre dans lequel les joueurs jouent.

Nous avons donc 8 tours qui se déroulent de la même façon, puis nous passons enfin au dernier behaviour, qui est le calcul des points de la manche.

LE CALCULBEHAVIOUR

Le dernier comportement de l'agent Partie correspond donc au calcul de fin de manche. Nous appelons *manche* les 8 tours précédents. La fin d'une manche correspond au moment où les joueurs n'ont donc plus de carte en main. Ce comportement étend, comme pour la distribution, la classe *OneShotBehaviour*.

La première chose à faire est donc de récupérer l'annonce en cours. Cela va nous permettre de voir si l'équipe qui est maître de cette annonce a un score supérieur ou égal à la valeur annoncée ou pas. Dans le cas où ce score est supérieur ou égal, l'équipe a remporté la manche mais son score est dans tous les cas fixé au score annoncé. Par exemple, si une équipe annonce **80** mais réalise **120** points, on redéfinit son score comme étant égal à **80** (plus le score des autres manches évidemment).

A l'inverse, si l'équipe qui est maître de l'annonce n'a pas réalisée ce qu'il devait, son score est mis à 0 (pour cette manche) et le score de l'équipe adverse est augmenté de 160.

La deuxième étape est de vérifier si un des deux scores est supérieur ou égal à 1000. En général, à l'UTC, les parties de coinche se jouent en 1000 points, c'est pourquoi nous avons fixé ce score là. Si aucune des deux équipes n'a encore atteint 1000 points, alors on va envoyer un message avec le performatif CONFIRM aux joueurs, pour que ceux-ci repartent de l'étape de récupération des cartes, puis Partie va recréer le même comportement séquentiel, à l'exception bien sur du ***SubscribeBehaviour*** dont nous n'avons plus besoin puisque l'on garde bien sur les mêmes joueurs.

Le *CalculBehaviour* va ensuite se terminer, et nous allons donc repartir du *DistributionBehaviour*, pour distribuer de nouvelles cartes aux joueurs, qui seront eux dans leur *RecupBehaviour*.

MODIFICATIONS APPORTEES

Nous avons dû modifier notre cahier des charges initial qui a été estimé trop conséquent pour un temps de réalisation trop court. Nous avons donc concentré notre travail sur les principales fonctionnalités qui peuvent intéresser les intervenants de IA04 et NF28.

Le joueur a donc toujours la possibilité de se connecter. Ensuite, la possibilité de créer une partie a été supprimée de part l'architecture réseau Jade utilisé. Nous avons juste maintenu la possibilité de « rejoindre une partie » qui n'est en faites qu'une partie unique gérée grâce à un serveur sur un PC. En effet, nous avons dû intégrer un ordinateur pour pouvoir utiliser JADE, afin d'avoir le système de partie centralisée. Il n'était pas possible de créer un *main conteneur* sur un smartphone Android.

Enfin, nous n'avons pas développé la partie Coinch'UTC qui ne rentrait pas dans le contexte de IA04 et NF28 (mais se rapprochait plus d'une gestion base de donnée de type NF17)

LIMITES DU PROTOTYPE ET AMELIORATIONS POSSIBLES

Il y a plusieurs points qui pourraient être améliorés :

- Modifier l'architecture réseau afin de ne pas avoir besoin de pc pour faire fonctionner l'application
- Implémenter la gestion multi parties
- Mise en place d'une intelligence artificielle pour s'entrainer
- Mise en place d'un système d'aide à chaque tour de jeu pour les débutants

CONCLUSION

En conclusion, ce projet nous a donc permis de mettre en application les conaissance acquise dans le cadre de l'UV IA04 en réalisant notre premier environnement multi-agent, et également de réaliser notre première application Android. Nous avons découvert l'efficacité de l'interface de message entre Agent avec leurs Behaviour associés et le potentiel système d'exploitation Android, membre éminent de la famille des systèmes mobile qui seront de plus en plus utilisés dans notre futur proche.

Nous avons énormément appris en réalisant ce projet, tant au niveau de la gestion de projets que de la programmation sous Java, Android et dans l'exploitation des librairie Jade et du parsing JSON. De plus, nous avons ainsi pu étudier la mise en correspondance de Jade/ Android.

Nous tenons donc à remercier les intervenants des UVs A04 et NF28 qui nous ont fait découvrir un fort intérêt pour cette dernière technologie.