CENTRE SCOLAIRE SAINTE-JULIENNE

Programmation et langages – Systèmes d'exploitation et logiciels

Consignes du Projet Jeu réseau

Le sujet

Dans le cadre de ce projet vous devez développer un jeu multi-joueurs en réseau.

Créer une telle application représente un lot de défis pour le développeur:

- Mettre en place une architecture « clients / serveur »;
- Gérer les entrées et sorties réseau;
- Paralléliser plusieurs threads;
- Implémenter la logique du jeu.

De plus, afin de permettre aux clients de communiquer avec le serveur, il sera nécessaire d'établir un **protocole de communication**. Il conviendra donc d'expliquer précisément la forme et la syntaxe des messages échangés.

Ce travail sera réalisé par groupe de 3 ou 4 élèves.

L'idée est de développer un jeu du type « Capture du drapeau », au tour par tour, sur un plateau en deux dimensions (représenté par une matrice par exemple).

Le plateau de jeux est divisé en différentes cases, positionnées aléatoirement:

- **vide**: pas d'implication particulière
- **drapeau**: lorsqu'atteinte par un joueur, celui-ci remporte la victoire
- **mur**: infranchissable
- **piscine**: le joueur ne joue pas le prochain tour
- **bar**: le joueur ne joue pas les 1, 2 ou 3 prochain(s) tour(s).
- **piège**: la quête du joueur s'arrête ici.

Les joueurs sont positionnés aux extrémités du plateau de jeu au démarrage, et chaque tour, un joueur peut se déplacer d'une seule case.

La visibilité de chaque joueur est réduite par un « brouillard de guerre » : seules les cases à proximité immédiate et les cases déjà parcourues sont visibles.

Remarque: toute la logique du jeu se trouve sur le serveur ! Le client va uniquement afficher le plateau de jeu et permettre les interactions avec l'utilisateur.

Vous trouverez sur l'illustration suivante un exemple de plateau de jeu.

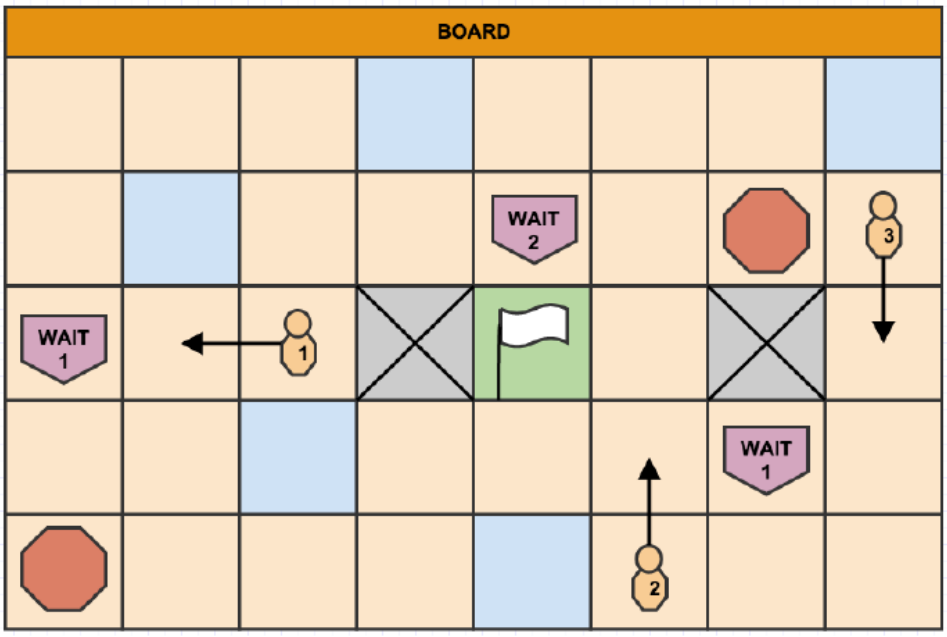

Figure 1 – Exemple de plateau de jeu

Le déroulement d'une partie

En démarrant son client, le joueur va se connecter au serveur, et s'identifier auprès de lui (via un nom d'utilisateur).

Il aura alors la possibilité de créer une nouvelle partie, ou d'en rejoindre une existante n'ayant pas encore démarré. Les parties non démarrées sont « en attente ». Il pourra également quitter l'application, et donc se déconnecter du serveur.

Lorsque le joueur hôte décide de démarrer la partie, tous les autres joueurs de la partie sont informés, et la partie peut commencer. C'est le serveur qui indiquera aux joueurs quand c'est leur tour de jouer.

Ces interactions sont représentées sur la figure 2.

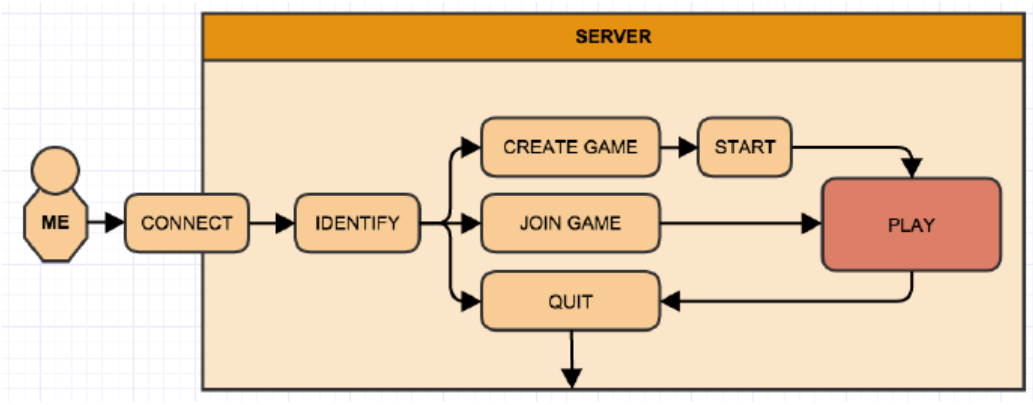

Figure 2 - Player story

L’architecture Clients - Serveur

Le serveur doit gérer une seule partie (Figure 3 à gauche).

Il y a la possibilité d’obtenir un bonus pour l’implémentation d’une solution multi-partie (Figure 3 à droite).

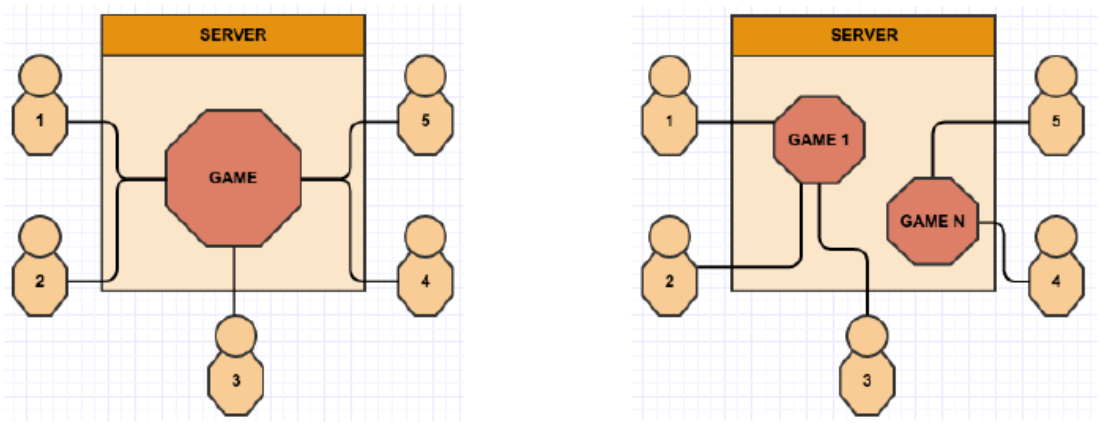

Figure 3 - Clients - Serveur

Le protocole de communication

Définitions générales

Pour préciser la syntaxe des messages échangés entre le serveur et ses clients, il faut convenir d’un format précis. Ce format, relativement intuitif, est la grammaire ABNF1.

```
space = %x20 ; le caractère espace (code ASCII 20 en hexadécimal)
crlf = %x0D %x0A ; caractères \r\n de Java & C
digit = "0"/"1"/"2"/"3"/"4"/"5"/"6"/"7"/"8"/"9" ; un chiffre
letter = %x41-5A / %x61-7A ; lettre majuscule A-Z ou minuscule a-z
passchar = %x21-ff ; les caractères imprimables sauf espace
character = %x20-ff ; les caractères imprimables (espace compris)
```

Définitions du protocole

```
player = 4*8(letter|digit) ; 4 à 8 lettres ou chiffres
receiver = player ; définition identique (pour la lisibilité)
sender = player ; définition identique (pour la lisibilité)
id = 1*4(digit) ; 1 à 4 chiffres
game_id = id ; définition identique (pour la lisibilité)
cell_id = id ; définition identique (pour la lisibilité)
game_xml = 1*50000character ; 1 à 50.000 caractères
error_code = 1*50(letter|digit) ; 1 à 20 lettres ou chiffres
error_message = 1*500character ; 1 à 500 caractères imprimables
hello = "HELLO" space player crlf
welcome = "WELCOME" space player crlf
game_create = "GAME_CREATE" crlf
game_join = "GAME_JOIN" space game_id crlf
game_start = "GAME_START" space game_id crlf
game_ended = "GAME_ENDED" space player crlf
```

```
game_updated = "GAME_UPDATED" space game_id space game_xml
game_action_move = "GAME_ACTION_MOVE" space game_id space cell_id
quit = "QUIT" crlf
error = "ERROR" space error_code space error_message crlf
```

Les explications

La grammaire ABNF définit exactement la syntaxe attendue pour les messages. Ainsi, tous les messages doivent se terminer par la séquence CR LF sinon l'implémentation n'est pas conforme. A titre d'exemple, voici quelques messages valides:

- **HELLO cobolforever**

Le message est valide puisqu'il commence par HELLO, qu'il est suivi

- **GAME_UPDATED 23 <game><players>…<players><board>…</board></game>**

Le message est valide puisqu'il commence par GAME_UPDATED, qu'il est suivi d'un espace, de l'identifiant de la partie, d'un espace et des informations sur l'état actuel de la partie stockées au format XML. Il faut bien sûr qu'il se termine par CR LF.

La syntaxe est extrêmement précise, ainsi envoyer un message commençant par hello (en minuscules donc) ne serait pas valide.

La séquence des messages

Après avoir défini exactement la forme des messages, il convient de préciser comment ces messages peuvent être échangés. Pour ce faire, nous allons utiliser une machine à états finis montrant les dialogues possibles entre un client et le serveur.

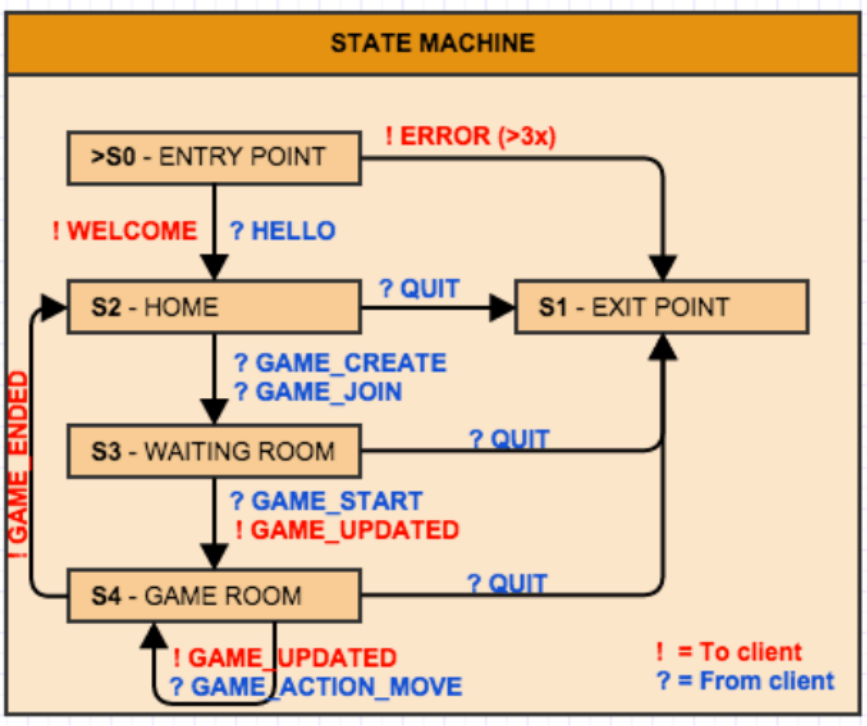

Figure 4 - Machine à états

S0	**ENTRY POINT** Etat initial lorsque la connexion réseau entre un client et le serveur est établie. Seul le message « hello » (voir grammaire ABNF) est attendu. Si un message « hello » est reçu, mais que le serveur détecte qu'il n'est pas conforme (3 tentatives), transmission d'un message « erreur », et passage à l'état S1.
S1	**EXIT POINT** Etat final représentant la terminaison de la connexion réseau. Plus aucune interaction n'est possible avec ce client qui est, maintenant, déconnecté.
S2	**HOME** Etat où l'utilisateur est connecté et identifié. Plusieurs actions peuvent être demandées par le client au serveur à partir d'ici : • Créer une partie (« GAME_CREATE ») : transition vers S3. • Rejoindre une partie existante (« GAME_JOIN ») : transition vers S3. • Quitter l'application (« QUIT ») : transition vers S1.
S3	**WAITING ROOM** Etat où l'utilisateur est en attente du démarrage d'une partie. Seul le joueur hôte peut démarrer la partie via le message « GAME_START ». Dans ce cas, tous les joueurs de la partie passent dans l'état S4.
S4	**GAME ROOM** Etat dans lequel l'utilisateur a entamé une partie. Dans cet état, à chaque action de l'utilisateur, lors de son tour de jeu, et autorisée par le serveur (« GAME_ACTION_MOVE »), le serveur transmettra à tous les joueurs un message « GAME_UPDATED », contenant l'état du jeu après l'action (mise à jour du plateau, des joueurs, à qui le tour, ...). Seul le serveur peut quitter cet état, lorsque la partie se termine (tous les joueurs sont exclus, ou la partie a été gagnée), avec le message « GAME_ENDED ». Retour à l'état S2.

L'implémentation du protocole

Les messages envoyés et reçus doivent être traités par une classe particulière (par exemple une classe nommée « Parser »). Le « Parser » est responsable de construire et d'analyser les messages et d'en identifier les différents éléments. Afin de vérifier l'exactitude des messages échangés, nous exigeons l'utilisation d'expressions régulières. Ainsi, vous ferez un usage intéressant de la classe Pattern de Java (cf. API Java).

Il est, sauf cas très particulier, interdit d'utiliser dans l'analyse des messages reçus, des méthodes comme String.contains, String.split ou encore String.substring. Des classes comme StringTokenizer ou StreamTokenizer sont également à éviter.

L'architecture applicative du serveur

Le langage

La partie serveur du laboratoire sera réalisée en Java.

Le système

La figure 5 suggère une architecture globale pour votre serveur, comportant plusieurs éléments distincts, que nous allons détailler dans la suite de ce document : Task Dispatcher, NET Server et Core.

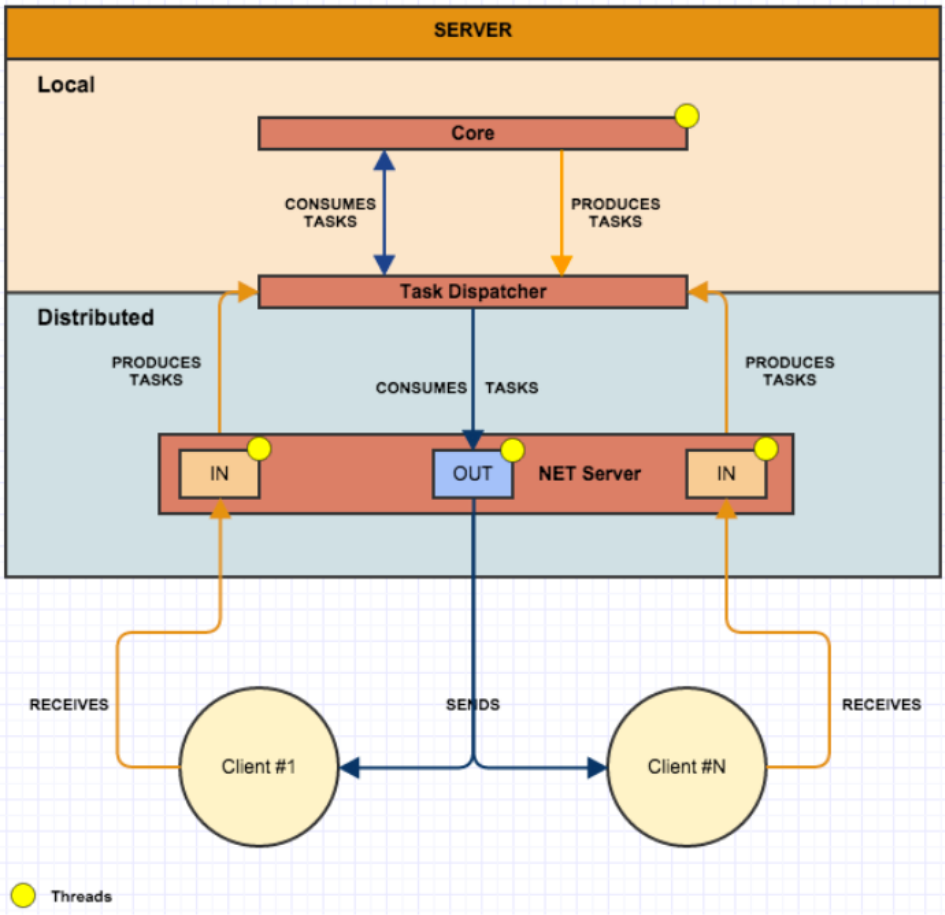

Figure 5 - Architecture globale du serveur

Le task dispatcher

Les éléments marqués d'une pastille jaune représentent des **threads**. Afin de synchroniser ceux-ci, et de leur permettre d'échanger des informations entre eux, vous utiliserez un mécanisme de propagation de tâches. C'est à cela que sert cette couche.

D'une part, lorsqu'un « thread » devra réaliser une action qu'il ne peut faire directement, ou dont la responsabilité incombe à un autre « thread », il créera un nouvel objet « Task » spécifiquement typé et contenant l'information suffisante, et demandera au composant « Task Dispatcher » (respectant idéalement le design pattern « singleton ») de la stocker. C'est le cas notamment pour les « threads » IN.

D'autre part, certains « threads » comme Core ou OUT, devront constamment demander les « Tasks » les concernant au composant « Task Dispatcher » afin de pouvoir les traiter. Ils pourront ensuite éventuellement ajouter de nouvelles tâches à destination d'autres « threads ».

La figure 6 illustre la gestion interne du « Task Dispatcher ».

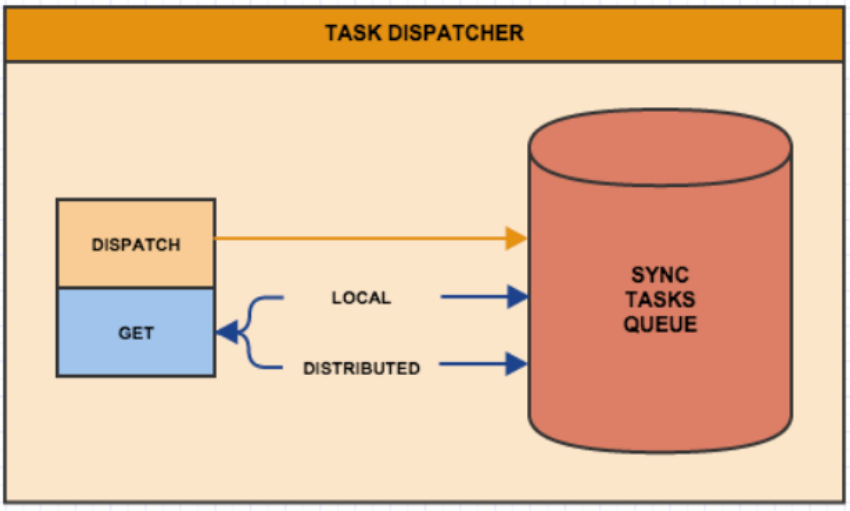

Figure 6 - Task Dispatcher

Le NET Server

C'est la couche de l'application qui gère les interactions et échanges réseaux, et qui met à jour la machine à états de chaque client. La figure 7 illustre le fonctionnement interne de cette couche :

- ☞ Les threads IN vont lire ce que leur envoient leurs clients, le décoder grâce au « Parser », et créer les tâches correspondantes à destination du « Core » du serveur.
- ☞ Le thread OUT va récupérer les tâches qui lui sont destinées, les analyser, et les encoder afin de les écrire sur le flux.
- ☞ Cette couche se charge également de mettre à jour la machine à états par client.

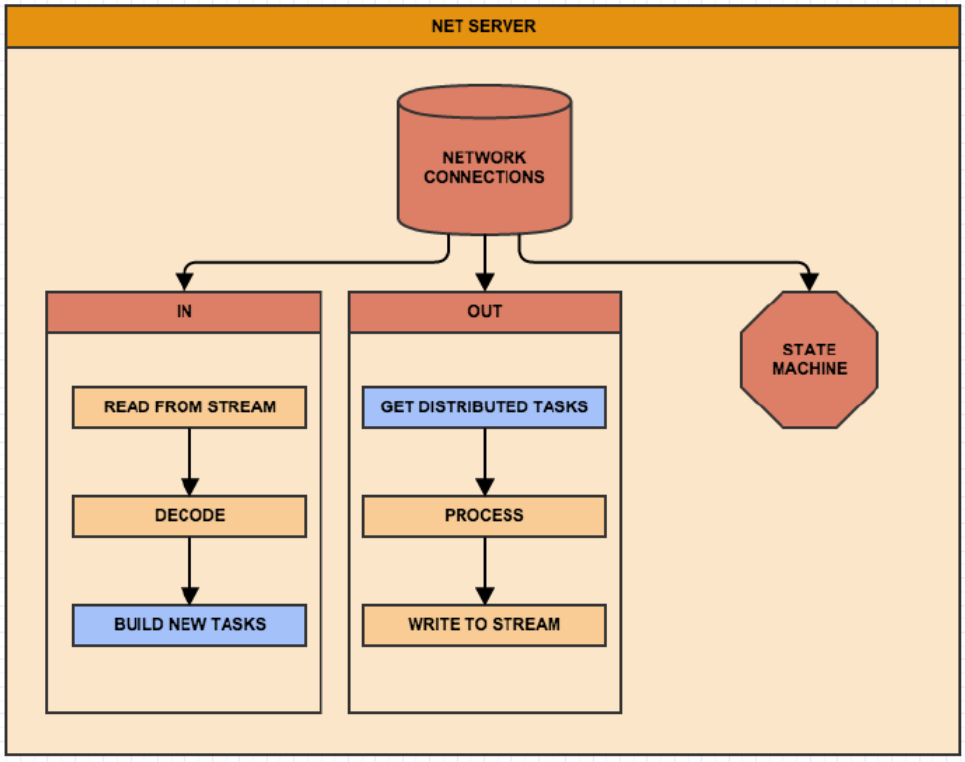

Figure 7 - NET Server

Le Core

C'est la couche de l'application qui se charge de gérer le modèle de données « métier » du serveur (joueurs, mécaniques de jeu, plateau, …). Celle-ci aussi sera gérée grâce au mécanisme des tâches, et n'agira que sur réception de celles-ci, ainsi qu'illustré sur la figure 8.

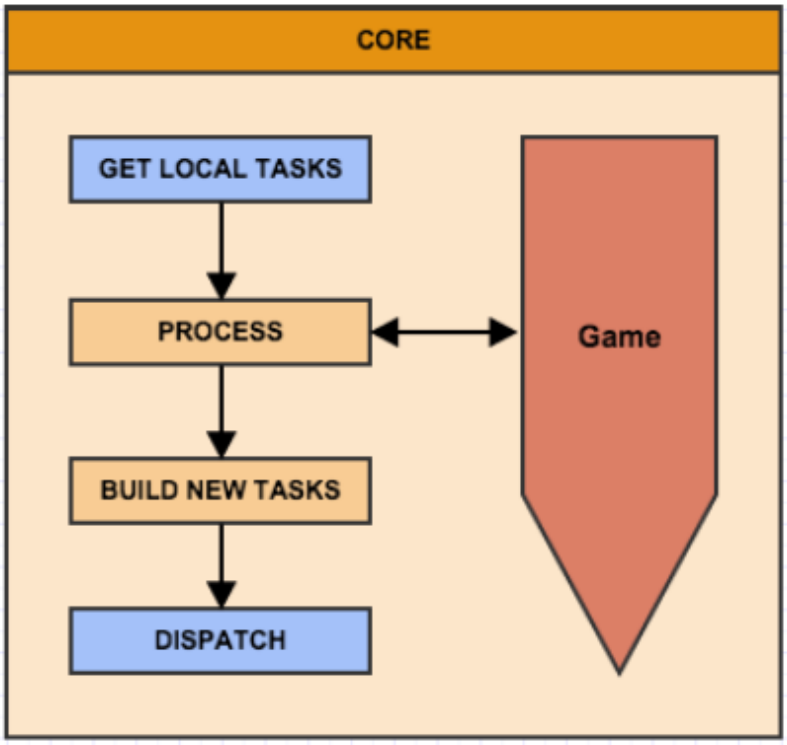

Figure 8 - Core

Les tasks

Vous trouverez plus d'informations au sujet de mode de fonctionnement des tâches elles-mêmes dans l'annexe.

L'architecture applicative du client

Le Langage

La partie cliente doit être réalisée en Java.

L'implémentation

L'architecture utilisée pour réaliser le serveur peut vous inspirer pour implémenter une partie de l'application cliente, notamment au niveau de la gestion du réseau voir du mécanisme de tâches.

Cependant, celle-ci devra idéalement respecter le design pattern MVC.

L'organisation du projet

Développer une telle application doit être fait rigoureusement, afin d'éviter les surprises au cours de l'implémentation. Nous vous conseillons donc de bien répartir les tâches dans votre groupe, et de subdiviser votre travail en plusieurs phases:

1. Implémentation et tests de la couche NET Server.
2. Implémentation et tests du protocole et de la couche Task Dispatcher.
3. Implémentation et tests de la couche Core (gestion des tâches, génération du plateau, mécanique de jeu…).
4. Implémentation d'un client basique (sans interface utilisateur) testant complètement le protocole.
5. Ajout de l'interface utilisateur au client.

Les phases 1 et 2 sont à rendre avant les vacances de Pâques au plus tard, et feront l'objet d'une évaluation formative.

Ce projet représente 100% des points des examens de Juin et devra être remis une semaine avant la période d'examens de Juin (version finale). Nous vous signalons que le **non respect des échéances sera sanctionné**.

La présentation orale portera, le jour de l'examen, sur la matière vue au cours et sur la forme de la solution apportée par chacun des membres du groupe, tant pour la partie Systèmes d'exploitation et logiciels que pour la partie Programmation et langages.

Les critères d'évaluation

Ils sont repris sur deux grilles d'évaluation, l'une pour l'écrit et l'autre pour l'oral.

Vous serez évalués sur les compétences suivantes:

C1 Exploiter les savoirs et les procédures en montrant qu'on en a compris le sens;

C2 Résoudre les problèmes par application des savoirs, des modèles et des concepts appris;

C3 Choisir parmi des concepts, des modèles, des procédures pour mener à bien la résolution d'un problème;

C4 Savoir travailler en autonomie, en groupe.

Nous vous souhaitons un bon travail!

Annexes

Les tâches

Nous avons besoin de deux types de tâches :

- ☞ des tâches locales (càd : gérées uniquement par le composant Core du serveur);
- ☞ et des tâches distribuées (qui seront converties en messages à envoyer sur le réseau).

Exemple:

Le serveur reçoit (via le thread IN de son client #1) le message « HELLO Louis ». Le « Parser » va vérifier la syntaxe du message. Si tout est correct, la prochaine étape consiste
à créer un utilisateur (ou joueur), portant le nom Louis.

Cette action, ce n'est pas le thread IN du client qui en a la charge, mais le « Core » du serveur. Afin de notifier celui-ci, le thread IN va créer une tâche locale, par exemple nommée « HelloTask », contenant les informations de l'utilisateur (ou du joueur), et va demander au « Task Dispatcher » de la dispatcher.

Ainsi, lorsque le « Core » du serveur va demander au « Task Dispatcher » les tâches le concernant, il trouvera une « HelloTask », et pourra alors créer l'utilisateur (ou joueur) en question. Une fois sa tâche accomplie, il dispatchera à son tour une tâche « WelcomeTask », à destination du client concerné. Lorsque le thread OUT trouvera cette tâche, il se chargera de la convertir en message et de l'envoyer à ses clients.

Le XML

Comme vous l'aurez remarqué, le format d'échange choisi pour partager l'état d'une partie est le format XML. Java propose diverses méthodes pour analyser et/ou construire des fichiers XML. La plus efficace dans notre cas est d'utiliser JAXB (Java-XML Binding) qui permet de faire correspondre une classe Java à un fichier XML. Ainsi, le programmeur ne s'occupe pas de traiter le fichier mais, simplement, demande à Java de le faire et obtient un objet d'une classe donnée.

Les interfaces

Afin de vous aider à structurer votre application, nous vous recommandons de définir les interfaces suivantes:

- ☞ ITaskProducer: représente un object producteur de tâches (comme IN ou Core par exemple).
- ☞ ITaskConsumer: représente un objet consommateur de tâches (comme OUT ou Core par exemple).
- ☞ IProtocolCodec: représente un objet permettant d'encoder et de décoder un message.
- ☞ IStoppable: représente un objet pouvant être « arrêté » (cela a plus de sens pour objet typé « Thread » ou « Runnable »).
- ☞ ISingleton: représente un objet implémentant le design pattern « Singleton3 » (comme le Tasks Dispatcher

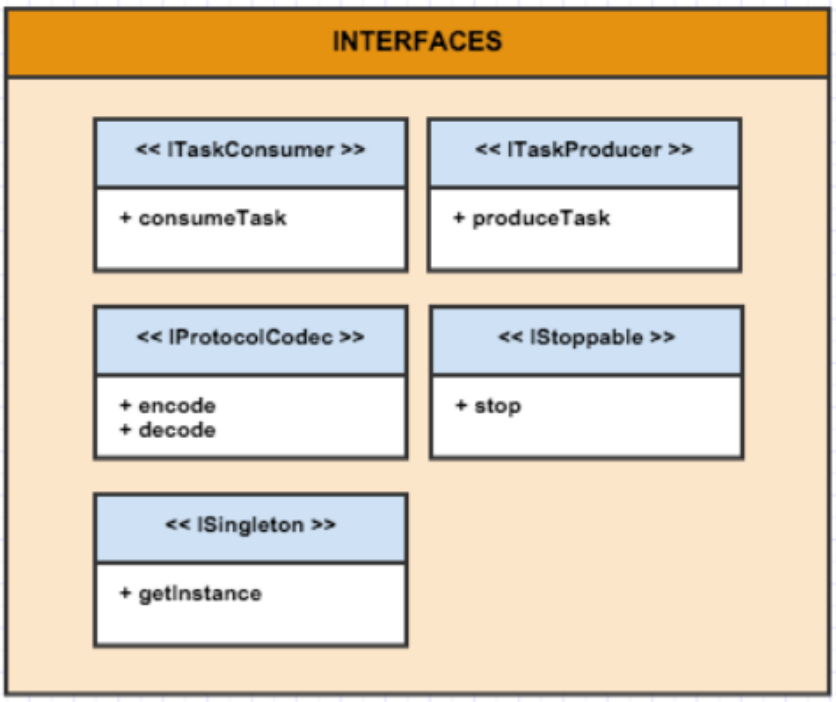

Figure 9 - Interfaces conseillées