21/12/2017

Projet Snake

Design Pattern

Guillaume HUET | Hugues DUMONT | Rachid AIT SIDI HAMMOU | Mohamed Achraf HAROUACH SLASSI

Master 1 informatique – Université d’Angers

Table des matières

[1. Introduction 2](#_Toc501739989)

[2. Objectif 2](#_Toc501739990)

[3. Les spécifications du projet 2](#_Toc501739991)

[3.1. PARTIE RESEAU 2](#_Toc501739992)

[3.1.1. Contexte 2](#_Toc501739993)

[3.1.2. Architecture du programme 3](#_Toc501739994)

[3.1.3. Définition du protocole de transmission 3](#_Toc501739995)

[3.1.4. Gérer la communication réseau en JAVA 4](#_Toc501739996)

[3.1.5. Sécurisation des échanges d’informations 5](#_Toc501739997)

[3.2. PARTIE DESIGN PATTERN 6](#_Toc501739998)

[3.2.1. Contexte 6](#_Toc501739999)

[3.2.2. Architecture globale : un premier pattern, le MVC 6](#_Toc501740000)

# Introduction

Dans le cadre du cours sur les Design Pattern, il nous a été demandé de travailler sur la conception d’un jeu de type Snake à la façon RPG et jouable en réseau. Il ne s’agit aucunement ici d’implémenter tout le jeu dans son intégralité mais de réfléchir uniquement à l’architecture de notre programme en faisant ressortir certains Design Pattern abordés durant les cours. Pour ce premier semestre, ce projet consiste en la rédaction d’un rapport renseignant les spécifications de notre jeu. Deux grandes parties sont décrites :

* Une partie réseau décrivant les échanges entre des hôtes distants.
* Une partie Design Pattern décrivant l’architecture logicielle de notre programme ainsi que tous les Design Pattern utilisés permettant de concevoir au mieux notre jeu.

Pour réaliser ce projet, la classe a été divisée en plusieurs petits groupes. Notre groupe quant à lui est composé des personnes suivantes :

* Guillaume HUET
* Hugues DUMONT
* Mohamed Achraf HAROUACH SLASSI
* Rachid AIT SIDI HAMMOU

# Objectif

Créer un jeu Snake en réseau multiplateforme (Windows, Linux, MacOs).

# Les spécifications du projet

## PARTIE RESEAU

### Contexte

Dans cette partie, nous allons nous intéresser à la partie réseau du projet Snake. En effet, il ne s’agit pas d’un simple jeu à jouer seul sur son ordinateur mais d’un jeu en réseau où plusieurs personnes peuvent venir jouer. Il s’agit donc d’un Snake en réseau et multijoueur.

Concernant le côté réseau du jeu, nous nous limitons ici au réseau local, le Local Area Network (LAN). En effet, pas d’objectif ici, en tout cas pour le moment, d’étendre le jeu au-delà du LAN. Cependant, l’idée n’est pas exclue.

Pour le côté multijoueur, il faut être au minimum 2 joueurs. Pour plus de facilité et de simplicité, nous allons restreindre à 2 le nombre de joueur pour notre jeu Snake. Chaque joueur jouera sur son propre ordinateur et chaque joueur contactera le même serveur par un protocole réseau qui sera définit dans la suite.

### Architecture du programme

Dans le cadre de notre jeu en réseau, l’architecture qui nous parait la plus naturelle et logique est l’architecture de type Client/Serveur, un modèle bien connu des utilisateurs d’internet. Dans le cas d’un exemple basique, un programme qui n’est autre qu’un client va demander un service ou une ressource à un autre programme qui lui est en fait le serveur. Le client effectue ce que l’on appelle des requêtes et le serveur va quant à lui envoyer des réponses en fournissant s’il le peut les ressources demandées. Ce modèle est représenté de la manière suivante (figure 1) :

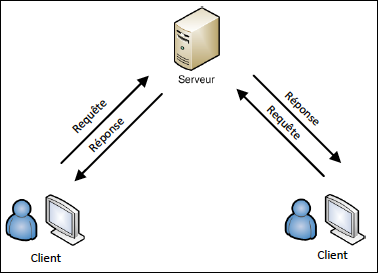


Figure - Modèle Client/Serveur

Dans notre jeu, nous aurons donc au minimum deux clients et un seul serveur.

### Définition du protocole de transmission

Nous allons nous intéresser ici à la couche transport du modèle TCP/IP et plus précisément aux protocoles de cette fameuse couche qui vont nous permettre de transporter les données de notre jeu. Il y a donc deux protocoles capables de transporter nos données entre nos clients et notre serveur :

* **Le protocole TCP** :
  + Avantages :
    - Orienté connexion.
    - Fiable.
    - Pas de contrôle à faire.
  + Inconvénients :
    - Lent si traitement de beaucoup de données volumineuses.
* **Le protocole UDP** :
  + Avantages :
    - Rapide dans un réseau non saturé.
    - Plus adapté généralement pour le multimédia (jeux, streaming…)
  + Inconvénients :
    - Non orienté connexion.
    - Non fiable.
    - Pas de contrôle d’erreurs ni de gestion de la congestion et donc il faudra mettre en place ce contrôle.

Suite à cette petite comparaison, nous avons décidé de nous orienter dans un premier vers l’utilisation du protocole TCP. En effet, ce dernier nous permettra de bénéficier d’une transmission fiable des données avec gestion des erreurs ce qui nous facilite grandement l’implémentation. De plus, nous ne prévoyons pas de transmettre une grande quantité de données volumineuse donc nous supposons que notre jeu sera fluide. Dans le cas où l’on constaterait d’énormes ralentissements alors nous envisagerons d’utiliser le protocole UDP. Cependant, il faudra pour ce dernier mettre en place la gestion d’erreurs à moins que la perte de certains paquets ne soit pas si grave dans le cas de notre jeu ?

### Gérer la communication réseau en JAVA

Cette partie consiste en une rapide description des outils que nous allons utiliser pour gérer la communication entre nos clients et notre serveur en JAVA. En effet, chaque action des clients sera transformée en un ou plusieurs objets qui seront traités par des classes spécifiques JAVA afin de les transmettre sur le réseau vers le serveur. De même, une fois que le serveur a effectué tous les traitements nécessaires, les réponses à envoyer aux clients seront également traitées par des classes spécifiques.

L’API java.net

Concrètement, pour gérer les communications entre les clients et le serveur, nous allons utiliser l’API réseau de JAVA qui est java.net. Cette API fournit un ensemble de classes permettant l’implémentation d’applications en réseau.

L’API peut être divisée en deux sections, une de bas niveau et une de haut niveau. Nous allons nous intéresser uniquement à la partie bas niveau. En effet, celle-ci traite des abstractions suivantes :

* Les adresses, qui sont les identifiants de réseau telles que les adresses IP.
* Les Sockets, qui sont des mécanismes de communication de données bidirectionnels de base.
* Les interfaces, qui décrivent les interfaces réseau.

Les Sockets

Pour faire communiquer nos clients avec le serveur et vice-versa, nous allons donc utiliser des adresses IP, des ports et des protocoles et pour échanger les différentes informations à travers le réseau, les clients et le serveur utiliseront des sockets.

Dans notre architecture, nous n’auront pas le choix, il faudra utiliser deux types de sockets :

* **Socket côté client** : permet la connexion à une machine distante afin de communiquer avec elle. Dans ce cas, nos clients établiront une connexion avec le serveur, enverront et recevront des données vers et depuis ce dernier et une fois que tout est terminé, la connexion se terminera.
* **Socket côté serveur** : connexion qui attend qu’un client vienne se connecter afin de communiquer avec lui. Ici, notre serveur sera en attente de connexion des clients. Pour le reste du fonctionnement il y aura également des échanges et une fois le tout terminé, on ferme toutes les connexions.

Concernant les numéros de port à utiliser, nous utiliserons ceux présents dans la plage des ports disponibles.

### Sécurisation des échanges d’informations

Un point important concerné la sécurité au niveau des communications qui s’effectuent entre les clients et le serveur. En effet, la question qui se pose actuellement est : doit-on chiffrer les communications clients <-> serveur ? Cela représenterait-il un risque dans notre architecture si tous les messages s’affichent en clair dans les trames Ethernet ?

Si vraiment on souhaite crypter toutes les communications alors on pourra s’orienter vers les SSLSockets et SSLServerSockets qui nous permettent d’avoir des Sockets sécurisées crypter en SSL ou TLS.

Cependant, accordant une importance sur la fluidité et la rapidité du jeu et donc des échanges d’informations, le fait déjà d’avoir choisi le protocole TCP fait naître en nous un petit doute quant à la validité de ces paramètres. Ajouter en plus du chiffrage sur toutes les communications augmenterait considérablement les temps de traitement de chaque paquet étant donné qu’il faut crypter et décrypter.

En revanche, il serait tout à fait envisageable de chiffrer uniquement certaines communications dans lesquelles passent des données sensibles. Par exemple, durant l’inscription d’un joueur ou de sa connexion au serveur où là il y a utilisation de login et de mot de passe, on cryptera les échanges à ce niveau. Une fois inscrit/connecté et donc In-Game, les communications pourront s’effectuer normalement sans que celles-ci soient chiffrées.

## PARTIE DESIGN PATTERN

### Contexte

Dans cette section, nous allons plus nous tourner vers l’architecture logicielle de notre jeu en nous aidant et en utilisant notamment les Design Pattern. Cette partie se composera de manière générale de diagramme de classes représentant la modélisation de notre jeu.

L’objectif est donc de réaliser un jeu de type Snake en mode RPG c’est-à-dire qu’il ne s’agit pas ici de contrôler un serpent qui ne fait que grossir en mangeant de la nourriture mais d’ajouter des fonctionnalités supplémentaires tels que des bonus qui font changer nos serpents de comportement, des obstacles qu’il faudra éviter, des retours dans le temps etc.

### Architecture globale : un premier pattern, le MVC

Ici, nous aurons une vue très généraliste et globale de notre jeu. Par ailleurs, un premier pattern vient ici faire son entrée. Il s’agit du pattern MVC (Modèle, Vue & Contrôleur). Le schéma représentant notre programme est le suivant :

CLIENT

SERVEUR

Vue 1

Vue N

Vue 2

Modèle

Contrôleur

Figure - Représentation sous forme MVC

Comme on peut le voir sur la figure numéro 2, nous avons une architecture MVC permettant de distinguer 3 grandes parties distinctes de notre logiciel. Par ailleurs, nous pouvons également constater une séparation entre les vues qui est en fait côté client et donc des joueurs et le contrôleur ainsi que le modèle qui sont côté serveur donc sur une machine distante accessible dans le réseau local.

Pour des raisons de sécurité, de maintenabilité et d’extension, nous avons décidé qu’il n’y aurait pas de communication du modèle vers les vues. En effet, nous avons préféré dans notre cas, une fois le modèle mis à jour, de notifier le contrôleur qui lui transmettra toutes les informations nécessaires aux vues afin que ces dernières se mettent également à jour. Obligation donc de passer par le contrôleur. Ce dernier peut nous faire penser à un mandataire. Enfin, tous ces éléments sont en fait représentés sous forme d’objets et dans notre cas d’objets JAVA.

Qui dit MVC dit également pattern composé et dans notre cas, nous verrons plus loin les différents patterns qui composent donc notre architecture.

Dans la suite de ce rapport, nous allons parler de chaque grande partie de notre architecture globale à savoir en premier lieu des vues puis viendra ensuite la partie contrôleur et enfin nous terminerons avec le modèle.

### Les vues du jeu (côté client)

### Le contrôleur (côté serveur)

Nous allons ici parler d’un objet permettant de faire le lien entre nos vues et notre modèle lorsqu’une ou plusieurs actions utilisateur surviennent sur les vues ou sur la vue courante. De plus, c’est cet objet qui aura pour rôle de contrôler les données. En effet, il s’agit de notre contrôleur.

Nous ne décrirons pas de manière exhaustive dans cette partie notre contrôleur mais plutôt de manière générale. Nous décrirons quelques méthodes que contiennent notre objet mais pas toutes. Il s’agit ici de comprendre facilement et rapidement le rôle qu’à le contrôleur.

Globalement, les actions effectuées par les utilisateurs seront captées par le contrôleur qui vérifiera la cohérence des données et si nécessaire les transformer afin que le modèle les comprenne. De plus, c’est notre contrôleur également qui se chargera de demander à la vue ou aux vues de changer après un ou plusieurs changements d’états du modèle. Par ailleurs, il est tout à fait possible que le contrôleur demande à la vue de changer directement après une action utilisateur sans attendre le changement d’états du modèle. Par exemple, on désactive immédiatement un bouton quand un utilisateur clique sur celui-ci afin d’éviter une anomalie de fonctionnement quelconque du programme et en attendant que les traitements au sein du modèle soient effectués.

### Le modèle (côte serveur)

Pour la vue :

* Définir les actions de la vue et donc ce que le contrôleur devra traiter, (déplacement via les touches directionnelles, barre d’espace pour utiliser un bonus etc.)

Bonus :

* Doit-on stocker un bonus et laisser libre à l’utilisateur de l’utiliser quand il le souhaite ? Dans ce cas, un seul bonus peut être stocké et passer sur un nouveau bonus ne fera rien. Sinon le bonus est activé dès que l’utilisateur passe sur celui-ci.