Projet programmation réseau

Martin Gleize

12 janvier 2011

Equipe : Charles-Pierre Astolfi, Raphaël Bonaque, Emile Contal, Martin Gleize, David Montoya.

1 Spécification du jeu

1.1 Le déroulement d'une partie

1. Pour commencer, un client et le serveur initient une communication RSA, permettant à l'utilisateur de se connecter à son compte ou d'en créer un nouveau de manière sécurisée.
2. Une fois le client authentifié, il peut accéder à la liste des parties en attente (*lobby*) et également décider de créer sa propre partie.
3. Quand un client décide de rejoindre une partie, il fournit au serveur les données qui vont permettre la connexion avec les pairs : adresse ip globale, port d'écoute et clé publique. Le serveur accepte alors la requête en envoyant en retour les données de connexion de tous les pairs déjà connectés. Il devra prévenir ces pairs de l'arrivée d'un nouveau joueur en leur fournissant les mêmes informations.
4. C'est alors que débute la véritable connexion *peer-to-peer*. Les clients procèdent à une étape d'introduction en trois étapes pour sécuriser la connexion. Ils vont signer leurs messages, rendus uniques par des nombres aléatoires (*nonce*), via DSA. Ainsi, sans la clé privée DSA, un faussaire ne peut fabriquer des paquets d'introduction et se faire passer pour un autre client.
   (a) Pour chaque pair déjà connecté, le nouveau joueur envoie une demande portant les identifiants nécessaires ainsi qu'un *nonce*.
   (b) La réponse à cette demande est constituée d'une répétition des identifiants, du *nonce* et d'un deuxième *nonce* fraichement généré. Le pair signe l'ensemble du message pour prouver son identité.
   (c) Enfin, le client ayant fait la demande vérifie que toutes les informations concordent et envoie un acquittement signé contenant les identifiants et les deux *nonces*.

   Les pairs en attente de partie ont maintenant accès à un *chat* de manière sécurisée. Ils sont connectés en clique (graphe complet) et peuvent envoyer et recevoir des messages signés. Comme pour les autres messages signés, les pairs sécurisent leur signature, ici en incorporant un *nonce* propre à la session.
5. Lorsque le créateur d'une partie le décide, il peut demander au serveur de commencer le jeu. Le serveur va alors prévenir les clients de se préparer à lancer la partie. Il leur fournit par la même occasion les premières pièces de Tetris. Quand tous les clients sont prêts, le serveur lance le jeu.
   Les prochaines pièces seront envoyées par le serveur à la demande des joueurs.
6. Les pairs informent tous leurs adversaires de leurs actions. On entend par action le fait de fixer un tetromino sur le plateau, ce qui est naturellement décrit par l'orientation de la pièce, son numéro dans l'ordre de distribution du serveur et ses coordonnées en abscisse et en ordonnée. A intervalle régulier (appelé *round*), ils sont censés envoyer un message signé comportant le

descriptif des actions effectuées durant ce round, le cas échéant. Les pairs reconstruisent ainsi le jeu des adversaires. Ils peuvent en vérifier la correction et doivent en inférer les pénalités qu'ils subissent. Pour assurer qu'un pair n'envoie pas des informations différentes aux autres, ce message contient également un hash SHA-1 des messages reçus au tour précédent. Ceci garantit la cohérence d'une seule et même partie valide malgré l'éventuelle présence de client frauduleux.

1.2 Le protocole réseau

L'ensemble du protocole est décrit dans un pdf que nous tenons à jour à chaque modification. Clients et serveur communiquent via TCP. La cohérence des parties n'étant assurée que si les messages arrivent tous et dans le bon ordre, nous n'avons pas pu utiliser UDP. Par soucis d'efficacité et de rigueur, nous avons choisi de spécifier le contenu des messages octet par octet. La structure des paquets envoyés est la suivante :

```
ProtocoleId + messageLength (2 bytes) + messageType (1 byte) + payload
```

Ce choix nous a amené à définir très précisément le codage des données, chaque langage ayant une manière différente de les représenter. Voici en particulier quelques points sur lesquels des précisions ont dû être données :

- L'ordre des octets envoyés correspond sur le réseau (*endianness*) Dans notre cas, nous avons choisi *big-endian*.
- Les entiers (8, 16, 32 et 64 bits) sont non signés.
- Les clés publiques et les signatures sont envoyées en concaténant les grands entiers qui les compose précédés de leurs tailles.

1.3 Les règles du jeu

Les règles que nous avons choisies suivent la spécification officielle [1] de *Tetris*® par *The Tetris Company*. Parmi ces règles on notera les plus intéressantes :

- La génération des nouvelles pièces se fait par séquence de permutations aléatoire des sept pièces [2].
- Les rotations s'effectuent en suivant l'algorithme *Super Rotation System* [3]. En particulier, les rotations doivent implémenter les *wall kick* : lorsqu'une rotation bloque, il faut tenter de décaler légèrement la pièce vers une position alternative. Ceci autorise les joueurs à effectuer des mouvements de pivots très intéressants lorsqu'ils sont bien contrôlés.

1. http://tetrisconcept.net/wiki/Tetris_Guideline
2. http://tetrisconcept.net/wiki/Random_Generator
3. http://tetrisconcept.net/wiki/SRS

2 Implémentation

MindTrisSharp est l'implémentation en C# du protocole de MindTris.

2.1 Côté serveur

Le serveur (classe **Server**) suit un modèle mono-thread adapté à la gestion d'une très grande masse d'utilisateurs : le nombre de threads n'augmente en effet pas avec les connexions ou les requêtes. La boucle principale (méthode **Listening**) réalise les actions suivantes :

1. Accueil de nouveaux clients (Action LISTEN sur le socket serveur)
2. Sélection des sockets à traiter (Action SELECT)
3. Réception des données (READ)
4. Traitement des données
5. Envoi des paquets serveur, souvent les réponses au requêtes des clients (SEND)

La possibilité au serveur de pouvoir effectuer ces actions successivement sans interruption est donnée par le mode non bloquant choisi pour les sockets. Si il n'y aucun client à accueillir ou aucune donnée à lire, l'action concernée est simplement ignorée et le thread ne "bloque" pas. Cela signifie aussi que la lecture des paquets peut ne pas compléter (le paquet n'est pas entièrement lu immédiatement après une itération de la boucle) ; l'implémentation utilise donc des buffers (circulaires : voir classe **BufferWindow**) pour stocker les données en attendant qu'un paquet complet soit formé.

La phase de traitement des données réalise la sémantique du serveur : selon les paquets, elle crée un nouvel utilisateur, le laisse se connecter à son compte, à un lobby, en créer un nouveau, demander des nouvelles pièces pour une partie en cours. La logique sous-jacente est très simple et le serveur retient grossièrement une collection des utilisateurs connectés et des lobbys en cours, avec les variables d'état qui les décrivent.

2.2 Côté client

Le jeu suivant un protocole *peer-to-peer* à la connexion des clients entre eux, le client (classe **Client**) intègre nécessairement la même logique serveur que le serveur lui-même. Il doit en effet accueillir de nouveaux pairs, recevoir leurs requêtes, les traiter et leur répondre. Pour ce qui est de la communication avec le serveur, le client implémente les méthodes complémentaires (il peut recevoir ce que le serveur envoie et inversement).
Le seul mécanisme à noter est l'utilisation d'évènements (*events*), levés pour signaler à l'interface utilisateur faisant usage du client que certains paquets ont été reçus ou envoyés. Par exemple, lorsque le client réussit à se connecter à un lobby et que le serveur lui a confirmé, l'évènement **LobbyJoined** est levé, fournissant les paramètres nécessaires à l'utilisateur pour procéder à l'étape suivante du protocole. Les events utilisent une fonctionnalité spécifique au C#, les **delegate**, qu'on peut voir comme des fonctions d'ordre supérieur.

2.3 Interface graphique

L'interface graphique du jeu utilise XNA, un framework conçu par Microsoft pour faciliter le développement de jeux vidéos sous Windows et Xbox 360. Ce framework gère beaucoup des composants du jeu : les entrées comme les frappes clavier, l'audio, le rafraichissement de l'image et le chargement de texture ou de formes.

L'implémentation est guidée par une logique "frame-by-frame", où la grille de jeu est redessinée par la carte graphique à intervalle de temps régulier (ici, 60 images par seconde), de même que le clavier, interrogé sur son état à intervalle régulier. Il a suffi de dériver la classe **Microsoft.Xna.Framework.Game** et de ré-implémenter en override les méthodes **Update** (chargée de la mise à jour de l'état interne du jeu et du traitement des frappes claviers) et **Draw** (chargée de dessinée la grille de jeu et les blocs qui y sont placés).
Les pièces (*tetrominos* dans la terminologie) et leurs rotations ont été codées "en dur", ainsi que les décalages offerts par le système de rotation SRS (produisant les fameux *wall-kicks*). A chaque fois que l'interface détecte une pièce posée sur le plateau (classe **Board**), un évènement est levé et le prochain paquet de round contiendra l'action effectuée. De même, lorsqu'un paquet de round non vide est reçu en provenance d'autres joueurs, un évènement est levé à destination du plateau d'observation (classe **BoardObserver**, qui est affiché de la même façon que **Board**, mais qui ne gère aucun mouvement des pièces et commandes du joueur) pour l'affichage de la pièce concernée, la suppression de lignes éventuelles et l'ajout de pénalités.
A ce sujet, dans le cas d'adversaires multiples, le choix a été de conserver l'affichage d'une seule grille supplémentaire de taille normale, et d'alterner régulièrement entre les plateaux des différents pairs pour aider le joueur à vérifier la progression et le score de ses concurrents.
Des sons (la musique originale de Tetris et des bruitages "faits maison") ont été ajoutés pour enrichir l'ambiance du jeu.

2.4 Plateformes ciblées

Le jeu est fonctionnel sous les systèmes Windows, ainsi que Mac OS et Linux par l'intermédiaire de *Mono*, une implémentation libre du framework .NET. La partie du jeu utilisant XNA est portable sous Linux et Mac OS grâce à *monoxna*.

3 Résultats

Le jeu est finalement très jouable, et l'état cohérent entre les différents joueurs est assuré. Les phases de test ont permis de repérer puis corriger les erreurs d'implémentation de la spécification, voire d'enrichir cette dernière si le besoin s'en faisait sentir. On pourrait améliorer la prise en charge des comptes d'utilisateurs en incluant la gestion d'une base de données, ce qui n'est pour l'instant pas fait. L'ajout d'un système de compétition permanente entre les joueurs d'un même serveur est essentielle pour dissuader les tricheurs, car si leur compte est rendu unique, ils ne veulent pas se le voir bannir ou pénaliser si leurs méfaits sont détectés par la sécurité du jeu, assez élaborée.