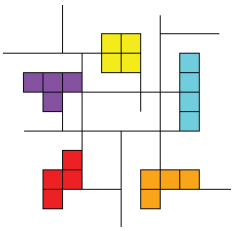
Projet programmation réseau

CONTAL Emile

12 janvier 2011



1 La spécification du jeu

1.1 Le déroulement d'une partie

Voici les différentes étapes que réalisent client et serveur durant l'établissement d'une partie.

1. Pour commencer, un client et le serveur initient une communication RSA, qui va permettre 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 va pouvoir accéder à la liste des parties en attente (*lobby*) et peut également fabriquer la sienne.
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 tout 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 leur message via DSA combiné à l'emploi de nombres aléatoires uniques (*nonce*) pour empêcher la réutilisation des messages par un fraudeur.
   (a) Pour chaque pair déjà connecté, le nouveau joueur va envoyer 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 au *chat* de manière sécurisée. Ils sont connectés en clique 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.

logo by David Montoya :)

5. Lorsque le créateur d'une partie le décide, il peut demander au serveur de commencer le jeu. Le serveur va alors dire aux clients de se préparer au lancement. Il leur fournit par la même occasion les premières pièces de la partie. Une fois que tout les clients sont prêts, le serveur lance le jeu.
   Les prochaines pièces seront envoyées par le serveur sur demande des clients.
6. Les pairs informent tout leurs adversaires de leurs action. A intervalle régulier (*round*), ils sont sensés envoyer un message signé comportant le descriptif des actions effectuées durant ce round. Les pairs reconstruisent ainsi le jeu des adversaires. Ils peuvent en vérifier la justesse 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 garantie 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 exemples importants :

- L'ordre des octets envoyés correspond au *network endianness* soit *big-endian*.
- Les entiers 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 choisi 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 aléatoires 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

2.1 Généralités

Afin de mettre au point notre protocole, nous nous réunissons régulièrement pour que chacun puisse apporter son point de vue. Nous avons utilisé un système de gestion de version pour autoriser tout le monde à récupérer le travail des autres. Il est ainsi aisé de tester la compatibilité des diverses versions.

J'ai choisi d'utiliser Java pour ce projet. Mon implémentation sépare systématiquement l'aspect graphique (vue et controleur) du modèle. Je me suis forcé à n'utiliser aucune librairie externe de façon à exécuter facilement mes binaires sous n'importe quelle plateforme.

J'ai fait usage sans modération des structures de Java pour faciliter le code.

2.2 Lecture et Ecriture des paquets

La communication est fondée sur la classe `Channel` qui gère les `Socket` et créer une instance des classes `InData` et `OutData`. Ces dernières héritent respectivement des classes de Java `DataInputStream` et `DataOutputStream`. Elles permettent la lecture et l'écriture de n'importe quel objet dont le codage est donné par les méthodes précisées dans l'interface `Encodable`. Je peux ainsi m'abstraire des détails du protocole dans l'implémentation du client et du serveur et léguer le codage aux classes particulières.

2.3 Reception des paquets

Les classes `Client` et `Server` disposent toutes deux d'une instance de `MsgHandler` ainsi qu'une implémentations de l'interface `Handler` par type de message. Le rôle du thread `MsgHandler` est d'écouter sur le `Channel` les messages entrants et de transmettre les données au `Handler` selon le type du message. Il est alors aisé d'ajouter de nouveaux types de paquets.