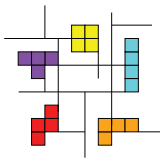
Projet programmation réseau
MindTris++

David Montoya

12 janvier 2011



1 Protocole réseau

Le protocole réseau est décomposé en deux : le protocole de communication client-serveur, et le protocole pair-à-pair. La liaison client-serveur permet aux joueurs de se rencontrer et de lancer une partie commune. Le serveur a le rôle de collecter l'information suffisante sur chaque client pour permettre leur authentification. Un client a le droit de créer une *partie* sur le serveur, que les clients intéressés pourront joindre. Le serveur transmet alors l'information propre a chaque client aux clients connectés sur une même partie, de telle sorte qu'ils puissent établir des liaisons pair-à-pair authentifiées. L'authentification entre les pairs repose largement sur le fait que la connexion avec le serveur est sûre, et un protocole d'authentification avec clé publique. Des détails sur l'implémentation du protocole sont fournis dans le document qu'on a prévu a cet effet, intitulé *MindTris Protocol Specification*.

Parmi les choix remarquables :

- Par soucis d'efficacité, l'unité de base pour transmettre l'information est l'octet. Ainsi, tout ce qui peut être codé sur moins de 256 valeurs, sera codé sur un octet. Les types des messages et des Tétrominos [1] sont un exemple. Un entier peut être encodé sur 1,2,4,8, ou un nombre variable d'octets, dans le cas des grands entiers, utiles lors de la transmission de clés publiques. Les chaînes de caractères sont codées en UTF-8 et sont préfixées d'un octet ou deux précisant la longueur en octets de la chaîne.
- Les entiers sont codés en *big-endian*. C'est la norme pour les communications réseau.
- L'utilisation du protocole TCP pour la couche Transport non seulement pour la liaison client-serveur, mais aussi pour les communications pair-à-pair. On aurait

1. pièces de Tétris

Nom	Encoding	Size (B)	Default	Comment
Protocole	STRING	4 ou 6	"DGMT" ou "DGMTP2P"	L'identifiant du protocole.
Taille	INTEGER	2		La taille du message en octets, en ajoutant la taille de l'entête (6 ou 8 octets).
Contenu	BYTE	variable		Le contenu du message. Celui-ci varie en fonction du type de message. Le premier octet est en général reservé pour indiquer le type du message.

pu choisir le protocole UDP pour les connexions pair-à-pair, qui a l'avantage de permettre le franchissement des *Network Address Translators*(NAT), mais cela nous aurait forcé à implémenter nous-même une couche de gestion de l'ordre et de la bonne réception des paquets.

– La transmission de l'information se fait par messages individuels. Le protocole spécifie chaque type de message, en général identifié par un octet placé presque au tout début. La structure des messages est donné dans le tableau 1
Bien que l'utilisation d'un identifiant du protocole ne soit pas vitale, elle permet aisément de filtrer des connexions parasites. L'utilisation d'un champ précisant la taille du message permet de récupérer individuellement chaque message, ceci n'étant pas géré par les couches réseaux inférieures.

1.1 client-serveur

Le protocole client-serveur est un protocole qui suit en grande partie le paradigme *request-response*. C'est au client de faire des requêtes au serveur, et au serveur d'y répondre. La seule différence est que le serveur peut être amené à répondre à une requête en envoyant des messages à plusieurs clients, qui ne sont pas du tout au courant de la requête en question.

La communication client-serveur est uniquement dédiée à l'identification des utilisateurs, à la création des parties, à la transmission des données utiles à l'établissement des liaisons pair-à-pair (adresses IP, ports, clés publiques, etc), et à l'envoi de nouvelles pièces de Tetris à jouer.

1.2 pair-à-pair

Les pairs établissent des connexions au moment où ils se connectent à une partie. Le serveur leur envoi de l'information sur de nouveaux joueurs au fur et à mesure que ces derniers se connectent, jusqu'au moment ou le créateur de la partie décide de la lancer. Au niveau logique, les pairs se connectent en formant une clique. L'établissement des connexions se fait par une poignée de main authentifiée en trois temps. C'est le rôle du client rentrant sur une partie d'initier une connexion avec les clients déjà présents. L'authentification repose sur l'envoi de *nonces* (variables aléatoires générées par chaque client) et l'utilisation de signatures (DSA), les clés publiques étant fournies par le serveur.

Tous les autres messages que les pairs échangent devront être signés. La communication pair-à-pair permet de gérer l'envoi des messages de texte entre les joueurs (chat) et des pièces jouées par chaque joueur.

2 Règles du jeu

Les règles que l'on a choisit pour le jeu de Tetris sont celles de la **Tetris Guideline**, que fait valoir *The Tetris Company* et qui est décrite sur `http://tetrisconcept.net/wiki/Tetris_Guideline`. Ainsi, les rotations s'effectuent en suivant l'algorithme *Super Rotation System* (`http://tetrisconcept.net/wiki/SRS`).

2.1 Le déroulement d'une partie

La partie débute une fois que tous les pairs ont établi des connexions entre eux, et que le créateur décide de lancer la partie. À ce moment, génère une liste aléatoire des premiers Tétrominos à être joués par tous les joueurs. Les joueurs vont au fur et à mesure demander de nouvelles pièces au serveur, lorsque ces derniers en auront besoin. Le serveur est censé envoyer l'information sur les nouvelles pièces à tous les joueurs. Les joueurs jouent donc tous les mêmes pièces.
Une fois la partie lancée, les joueurs sont censés informer leurs adversaires de leurs mouvements. Pour cela, le jeu est divisé en *rounds*, qui se jouent à intervalles réguliers de $100ms$. Sur chaque round, les joueurs placent leurs pièces, et à la fin du round, ils informent les autres des endroits où ils ont placé les pièces. Les joueurs doivent donc reconstruire le jeu de leurs adversaires en fonction des pièces que le serveur fourni et des positions où elles ont été placées. La cohérence entre les différentes reconstructions est due à une spécification rigoureuse des règles du jeu et du codage des mouvements. À chaque round, les joueurs envoient en plus des hash de l'information reçue des autres joueurs aux rounds précédents au fur et à mesure qu'ils disposent de cette information. Ainsi, les joueurs peuvent vérifier entre eux que tout le monde reçoit les mêmes mouvements de chaque joueur, ce qui permet de départager les tricheurs des non tricheurs.
Les pénalités sont simples : si un joueur fait n-lignes, le joueur suivant recevra n-1 lignes avec trou, tout en bas de son jeu. À noter que les joueurs ne reçoivent pas de message leur avertissant d'une pénalité, c'est à eux de les calculer en fonction des jeux des autres. Compte tenu du délai entre les joueurs à travers le réseau, les joueurs appliquent chaque pénalité avec 10 rounds de décalage. En général, un joueur à une visualisation des jeux de ses adversaires en retard avec la sienne, mais elle n'est pas sensé dépassé 5 rounds, ce qui correspond à un délai de 500ms. Autrement le jeu ne peut pas être joué.

3 Implémentation

Pour ce projet, j'ai choisi de faire une implémentation en C++. Bien que je n'avais jamais fait tout seul un vrai programme en C++ auparavant, j'avais un peu d'expérience, et je savais que le langage alliait performance et existence de bibliothèques robustes adaptées pour ce projet, que la programmation objet convenait tout à fait au développement d'un

jeu avec une interface graphique, et que la programmation procédurale pourrait simplifier les choix d'implémentation là où la réécriture du code n'était point vitale. Les bibliothèques que j'ai choisies ont été :

- **Standard Template Library (STL)** : pour sa classe string, l'utilisation des conteneurs et l'abstraction des pointers. J'ai en partie utilisé de aspects propre à la norme **C++1x** comme les *unique_ptr* et *move semantics*.
- **Simple and Fast Multimedia Library (SFML)** : pour les graphismes et l'audio.
- **Boost** : pour ses classes gérant le multi-threading, comme les classes `boost::thread` et `boost::mutex`.
- **Crypto++** : pour ses algorithmes cryptographiques, à savoir RSA pour l'encryption et DSA pour les signatures, mais aussi pour son générateur pseudo-aléatoire de nombres.

Bien que toutes ces bibliothèques soient multi-plateforme, leur configuration sur ma plateforme de développement a été suffisamment délicate pour ne pas me donner le temps de compiler la dernière version de mon programme sur plusieurs plateformes. Ma version finale n'a été testé que sur Windows. J'ai par contre compilé et testé avec succès ma version de Décembre, qui n'utilisait que la bibliothèque Crypto++ sur GNU/Linux.

Mon implémentation est divisé en trois parties :

- **mindtriscore** : Regroupe l'ensemble des classes et des méthodes communes au serveur et au client. On va remarquer l'existence d'une classe pour la gestion des sockets, et de classes pour la construction et la décomposition analytique des messages propres au protocole réseau.

- **MindTris++** : Le client. Il est décompose en trois parties, une première qui gère la communication avec le serveur, et la console utilisateur, une deuxième qui gère la communication avec les pairs, et une troisième qui gère la logique du jeu, et donne le rendu graphique. C'est la console de la première partie qui est lancée en premier, et elle communique avec la deuxième partie pour la gestion des pairs. L'aspect console et communication réseau se fait sur un seul thread. La partie jeu (troisième) est lancée sur un thread différent, et les entrées-sorties réseau sont gérées par les deux premières parties.

- **mt_server** : Le serveur. Il gère sur un seul thread les connexions clients entrantes, et communique de nouvelles informations à un ou plusieurs clients au fur et à mesure qu'il reçoit des requêtes de la part des différents clients.

3.1 mindtriscore

Voici les classes qui sont définies :
- `ByteArray` : construction, décomposition et manipulation des chaînes d'octets. Utile pour la construction des messages.
- `ByteBuffer` : lecture et écriture des flux des données. Utile pour la réception et

l'envoi de l'information sur les *sockets*.
- `MessageBuilder` et `MessageParser` : construction et décomposition analytique des messages, respectivement.
- `MessageStreamer` : construction et extraction de l'entête et du contenu du message. Manipule directement le flux d'octets.
- `OrderedAllocationVector` : manipulation efficace de listes indexées d'éléments.
- `DGMTP2PProtocol` : construction et extraction de l'information contenue dans les messages du protocole pair-à-pair.
- `DGMTProtocol` : construction et extraction de l'information contenue dans les messages du protocole client-serveur.
- `Packet` : spécification de la construction entête-contenu des messages.
- `CSocket`, `CTCPSocket`, `CTCPClient`, `CTCPServer` : manipulation *sockets*, en particulier des *sockets* TCP. Une classe pour chaque approche : celle du client et celle du serveur.
- `Tetromino` : codage et représentation des Tétrominos.

3.2 MindTris++

- `GfxManager` et `SfxManager` : gestion du son et des graphismes du jeu.
- `Block` : représentation graphique des carrés dans le jeu.
- `Board` : gestion du plateau de jeu, à savoir le rectangle 10x20 où se joue Tetris.
- `Mover` : gestion du Tetromino à placer, avec mouvement et rotation en fonction du temps et des entrées utilisateur.
- `MoverProvider` : obtention des nouvelles pièces. Interfacé avec MindTrisClient pour obtenir les nouvelles pièces du serveur.
- `Tetris` et `InGame` : gère le thread du jeu, les entrées utilisateurs, communique avec MindTrisClient pour l'envoi et l'obtention de l'information sur chaque round, et connecte les classes relatives à la gestion logique et graphique du jeu.
- `MindTrisClient` : la classe principale, point de d'entrée du programme. Gère la boucle principale, qui alterne réception de nouvelles connexions pair-à-pair, lecture des données sur les *sockets*, mis-à-jour de l'état du client en fonction des entrées utilisateur à travers la console et des messages reçus des autres pairs et du serveur, construction de nouveaux messages à envoyer aux autres pairs et au serveur en fonction du nouvel état du client, écriture des données sur les *sockets*, et rafraîchissement du rendu de la console.
- `Peer` et `PeerInfo` : interprète l'information reçue d'un pair, communique avec MindTrisClient et Tetris pour mettre à jour l'état du client et du jeu, et décide des nouveaux messages à envoyer aux autres pairs.

3.3 mt_server

- `ServerDatabase` : gestion de la base de données des utilisateurs. C'est une abstraction pour une base de données qui pourrait être stockée sur par exemple une serveur SQL.
- `Lobby` : gestion des *Lobby*, à savoir des parties créées sur le serveur.
- `MindTrisServer` : la classe principale, point de d'entrée du programme. Gère la

boucle principale, qui alterne entre réception de nouvelles connexions client, lecture des données sur les *sockets*, mis-à-jour de l'état du serveur en fonction des messages reçus des clients, construction de nouveaux messages à envoyer aux clients en fonction du nouvel état du serveur, et écriture des données sur les *sockets*.
– `User` : interprète l'information reçue d'un client, communique avec MindTrisServer et Lobby pour mettre à jour l'état du serveur, et décide des nouveaux messages à envoyer.

4 Conclusion

Ce projet m'a appris plusieurs choses. D'une part, à travailler en groupe avec beaucoup plus de communication. Tout cela parce que à la fin de la journée, nos programmes devaient communiquer entre eux, et cela ne pouvait se faire qu'avec une implémentation rigoureuse d'un protocole. Au final, les décisions ne sont prises que par une ou deux personnes, puis il faut les communiquer aux autres et chercher un consensus. Nous avons utilisé comme plateforme de travail un serveur de gestion des versions, un programme de chat vocal pour parler à plusieurs, et nos boîtes mail. D'autre part, j'ai appris à me servir de quelques bibliothèques en C++ assez riches (Crypto++ et boost), j'ai mis-à-jour mes connaissances en matière de modèles de conception pour la programmation orientée objet, et je me suis familiarisé avec quelques nouveaux outils de la norme de C++ à venir (C++1x).

Parmi les difficultés que j'ai rencontré en matière de programmation, cela a été dû à une connaissance pas très précise des bibliothèques utilisées, à des choix d'implémentation assez délicats, mais surtout aux difficultés et les problèmes qui ont survenu lors de la communications entre nos programmes. Souvent dûs à des interprétations différentes des consignes ou à des différences spécifiques aux langages qu'on a choisi, on a dû surmonter ces difficultés en analysant les messages qu'on échangeait et quelques fois en nous communiquant des différences entre nos choix d'implémentation.

Des choses qu'il resterait à faire, ça serait d'abord de rendre plus robustes le client et le serveur, qui peuvent toujours planter. Il faudrait aussi spécifier et implémenter le protocole de réaction face à des éventuels tricheurs. Cet aspect est déjà difficile à tester, vu qu'il faudrait implémenter un client qui essaye de tricher, pour des raisons évidentes. Des améliorations de l'expérience utilisateur et de l'interface seraient bienvenues. L'implémentation d'une vrai base de données est aussi à faire.