Rapport de projet Java – présentation de School M@n@ger

Avertissement d'utilisation

J'informe mon aimable correcteur que ce programme Java a été développé avec Eclipse sous Mac OS X. Ainsi, bien que testé en machine virtuelle Windows (via Oracle VM VirtualBox) sans encombre notoire, il se peut que quelques défauts apparaissent sur un PC sous Windows classique. En tout cas, ce programme fonctionne sans aucun problème sous Mac OS X (aucune erreur !). Dans l'ensemble, pour éviter au maximum les problèmes, le mieux est de mettre à jour votre JVM à **la version JAVA 6 SE** (en adoptant le dernier JRE en date sur le site d'Oracle pour Java). J'informe également que je ne suis pas parvenu, malgré le temps passé, à former un JAR correcte gérant les fichiers xml etc…

Dans tous les cas, les captures d'écrans sont disponibles sur ce rapport et vous avez eu l'occasion de visualiser mon application à maintes reprises lors des séances de Travaux Pratiques à l'EFREI.

Introduction

Cette année, le sujet du projet Java nous amène à concevoir une application client – serveur pour la gestion d'une école. Les consignes étant plutôt « vagues » ; nous nous sommes permis de faire des choix. Choix que nous avons fait selon plusieurs critères que nous expliciterons plus tard dans ce rapport. Nous verrons donc, dans un premier temps, les choix qui ont été fait puis dans un deuxième temps ; l'architecture qui a été choisie. Le projet ainsi élaboré a été baptisé « School Manager » (pour « le manageur d'école ») ;

Modèles de conception

La tâche la plus compliquée dans ce projet était sans aucun doute la conception. Nous avons fait des choix et pris des initiatives afin de faciliter la tenue globale et l'utilisation finale du programme.

Langage

Java

Le langage était une contrainte imposée. Le **Java** est un langage récent (lancé dans les années 90 par Sun puis racheté récemment par Oracle). C'est un langage **orienté objet**, c'est d'ailleurs son atout principal et ce qui en fait toute sa puissance. Le second point fort du Java est sa **portabilité**. En effet, hors mis certaines fonctions généralement futiles, toutes les fonctionnalités proposées au développeur sont disponibles sur tous les systèmes d'exploitation supportant la **machine virtuelle Java** (JVM). C'est donc muni d'un langage déjà très avancé et très efficace que nous pouvons commencer le travail de conception.

SQL

Etant donné que notre choix dans le stockage des données s'est porté sur une base de donnée Apache Derby (portable et légère car tout en Java), nous avons donc dû nous remettre au SQL avec des requêtes parfois assez lourdes avec des jointures externes, des likes etc.

Stockage

Globalement, nous avions trois choix quand au stockage des données sur le serveur.

- La **sérialisation** des objets de classe
- Le stockage en **base de données**
- Le stockage en **fichier de données brut**

Même si la sérialisation était envisageable, nous avons préféré choisir le stockage en base de données. Pourquoi ? Tout simplement car il est beaucoup plus simple de cibler des données insérées en base de données. De plus, les données dans une BDD sont réutilisables très facilement. Une base de données peut se copier n'importe où. Enfin, à titre pédagogique pour nous, nous voulions apprendre comment manipuler une base de données en Java.

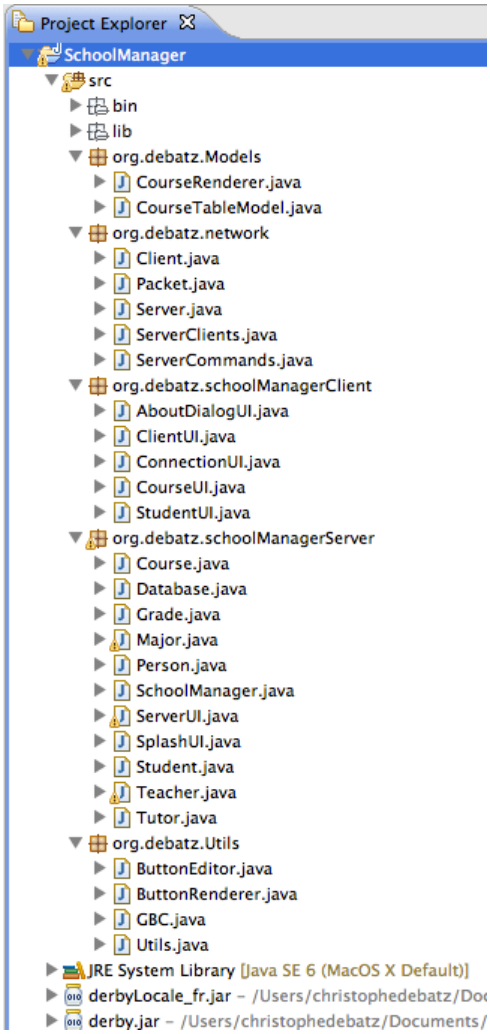
Cela dit, n'importe quel SGBD (Système de Gestion de Base de Données) n'était pas envisageable. En effet, des bases relationnelles connues telles que MySQL ou encore PostgreSQL s'avèrent un peu volumineuses pour une telle application. De plus, le point vraiment négatif est que ces bases requièrent l'installation de drivers pour pouvoir connecter JDBC avec la base en question.

Nous avons donc choisis une base de données intégralement développée en Java. Nous avons testé HSQLDB. Nos tests se sont révélés très lents d'autant plus que nous avons rencontré beaucoup de problèmes de compatibilité avec Mac OS X (système sur lequel je développe actuellement). Finalement, nous avons adopté **Derby DB** (aussi appelée Java DB) qui est proposée par l'équipe d'**Apache**. Cette base Java a pour avantage d'être suffisamment connue pour bénéficier d'un bon support que ce soit grâce aux forums communautaires (developpez.com, le site du zéro etc.) ou grâce à sa propre documentation sur le site d'Oracle ou d'Apache. Le point fort d'une base comme Derby est qu'elle est entièrement conçue en Java et est donc portable. De plus, elle ne nécessite aucune installation supplémentaire. La seule chose requise est de fournir l'archive Java avec le projet.

Architecture des fichiers

Le choix de la structure client – serveur était également imposé par le sujet. Cela dit, comment faire transiter des données entre deux postes ? Nous avons créé un ensemble de classes permettant de gérer cela. Nous avons notamment choisi de faire transiter un unique objet (Packet) au lieu de faire un test d'instance à chaque réception pour savoir quel objet on doit traiter.

Partition entre les fichiers



Le programme est fracturé en cinq paquetages :

- Le premier gère **le réseau** (network)
- Le second gère l'**activité client** (org.debatz.schoolManagerClient)
- Le troisième s'occupe de **l'activité serveur**
- Le quatrième contient les **classes communes** à tous les autres paquetages (les outils...)
- Le cinquième contient les classes communes pour les modèles de JTable (Models)

Comme vous pouvez le remarquer ; pour plus de confort d'utilisation, nous avons regroupé l'interface client et serveur dans un seul logiciel au lieu de deux. Comme ça, on a une seule archive JAR au final.

Les fichiers liés aux données stockées par Derby sont présents dans le dossier « DerbyDB » à la racine du projet. Ce dossier contient des fichiers de métadonnées indéxés par nom d' »environnement » (c'est-à-dire le nom de la base de données) et par nom de table (correspondante à chacune des catégories sur lesquelles nous travaillons).

Le fichier log de Derby se trouve à la racine du projet et contient en autre, les erreurs d'accès aux données (parfois nécessaire au débogage).

Fichiers de configuration

L'application nécessite trois fichiers de configuration au format texte et XML.

- Un fichier de **configuration pour le serveur** (identifiant de connexion Derby)
- Un fichier de **configuration pour le client** (préférences de connexion)
- Un fichier contenant les **instructions SQL de création des tables** de la base de données (afin de pouvoir vider la base puis la recréer sans problème)

Diagramme des classes

Des classes structurées

Le principal avantage à opter pour une solution tout-en-un (c'est-à-dire le client et le serveur dans le même programme) est que les classes permettant la manipulation des principaux objets (élèves, cours, professeur etc.) sont accessibles partout. Donc le code n'a à être révisé et éventuellement modifié qu'une seule fois pour que les modifications s'appliquent des deux côtés. Nous avons également fait un effort sur la modularité. L'ajout ou la suppression d'une table dans la base n'entraînent que des changements mineurs dans un fichier précis (ServerCommands.java a priori). De plus, le projet a été structuré autour de plusieurs fichiers importants. Ceux-ci ont été rangés dans des paquetages et des classes. Des techniques de

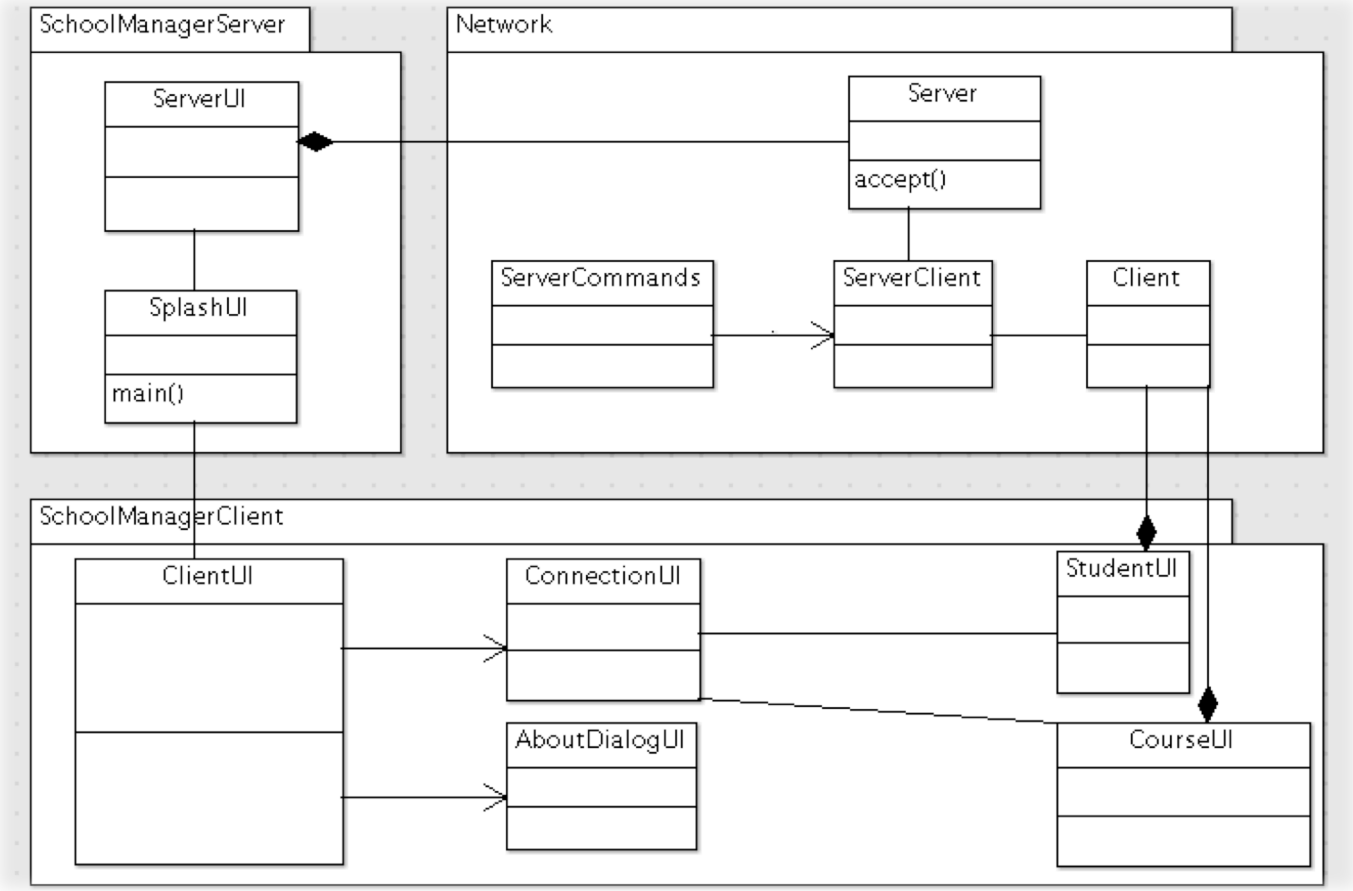
développement astucieuses comme le design pattern du singleton ou encore du polymorphisme ont également été de la partie pour rendre le code moins lourd, mieux structuré et davantage lisible.

Une interface utilisateur avec SWING et AWT

D'un autre côté, l'interface utilisateur a entièrement été conçu avec les bibliothèques destinées aux IHM intégrée naturellement à Java qui sont SWING et AWT. Nous vous avions envoyé un courriel pour savoir si nous pouvions utiliser une interface « by IBM » avec SWT mais nous avons finalement opté pour SWING car le projet allait nous demander suffisamment de travail.

L'avantage de SWING étant sa grande portabilité (indépendance des composants vis-à-vis du système d'exploitation) et sa vitesse d'exécution plus rapide que sa sœur lorsque beaucoup de données doivent être traitées. SWT, de son côté, est davantage préférée pour son intuitivité avec l'utilisateur (étant donné que cette bibliothèque adopte les composants graphiques natifs du système d'exploitation sur lequel elle s'exécute). Elle s'avère plus rapide pour de faibles lots de données. Malheureusement, elle n'est pas tout le temps portable (cela dit, pour les composants que nous utilisons, ce n'est pas le problème). Tout cela pour dire que l'interface utilisateur est présente dans les fichiers suffixés par le mot « UI » (UI pour « User Interface »).

Voici le diagramme (des classes) simplifié pour comprendre comment fonctionne le modèle :



Les classes de modélisation des données (mapping) sont situées dans le paquetage SchoolManagerServer mais j'ai préféré ne pas les inclure dans ce diagramme pour ne pas complexifié les choses.

La construction client – serveur

Principe utilisé

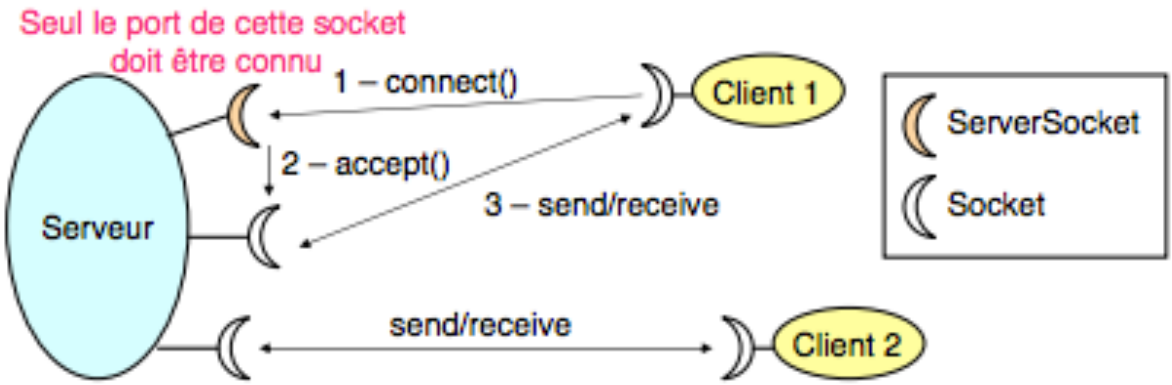
L'architecteur client – serveur est une notion très importante lorsque l'on a besoin de développer une application ou un serveur devra centraliser des données à un endroit et y laisser accéder des clients externes.

Il existe, en Java, trois types d'architectures connues :

- Les **paquets Datagram** (UDP)
- Les **paquets en flux** (TCP), ce que l'on utilisera ici
- Les **paquets multicast** (diffusion à de multiples clients)

Le protocole TCP nous garanti entre autre, l'ordres des paquets qui arrivent ainsi que l'intégrité des données. Il est un peu plus lent que son confrère UDP. A la vue du peu de données que nous avons à transférer, nous pouvons nous contenter de TCP concernant sa vitesse de transmission.

Voici le schéma de comment fonctionne le protocole TCP (en flux de données) en Java :



Au niveau du code, l'idée qui prédomine est le fait que plusieurs clients peuvent se connecter au serveur très facilement. Pour se connecter au serveur, seules deux informations sont utiles :

- **L'adresse IP** du serveur
- **Le port d'écoute** du serveur

Une fois ces deux informations connues du client, celui-ci peut tenter une connexion vers le serveur. C'est toujours le serveur qui décide si oui ou non un client peut se connecter.

Dans un premier temps, il faut savoir que toutes les classes nécessaires à la gestion du réseau sont regroupées dans le paquetage org.debatz.network.
Dans ce paquetage, la classe Server permet de gérer la connexion de chaque client via une table de hachage couplée à l'utilisation des Threads. Cette classe permet la gestion du serveur par l'administrateur (par exemple, fermer le serveur ou bannir un client connecté). C'est aussi cette classe qui accepte ou non les nouvelles connexions de clients.

La classe ServerClient est justement un objet créé par la classe Server dans un Thread afin de gérer chaque session cliente. Cette classe permet d'envoyer et de recevoir des données de la part du client courant. De ce fait, chaque client peut interargir comme il le souhaite avec le serveur sans que celui-ci ne confonde les différentes commandes des différents clients. Une

chose en entrainant une autre, la gestion des transactions (begin transaction et commit) en base de donnée étant plutôt complexe, nous avons plutôt décidé de synchronisé les méthodes permettant de modifier les données pour ne pas que deux clients puissent modifier une seule donnée en même temps.

La classe Client est l'équivalent de la classe précédente mais côté client. Elle permet au client de dialoguer avec ServerClient (côté serveur) afin de lire ou de modifier des données présentes dans la base de donnée du serveur. Elle permet également de se déconnecter du serveur.

La classe ServerCommands est une classe utilisée côté serveur pour lire et reconnaître les requêtes clients. Par exemple, si un client demande d'insérer un nouvel étudiant, c'est cette classe qui s'occupera d'interpréter puis d'exécuter la requête client.

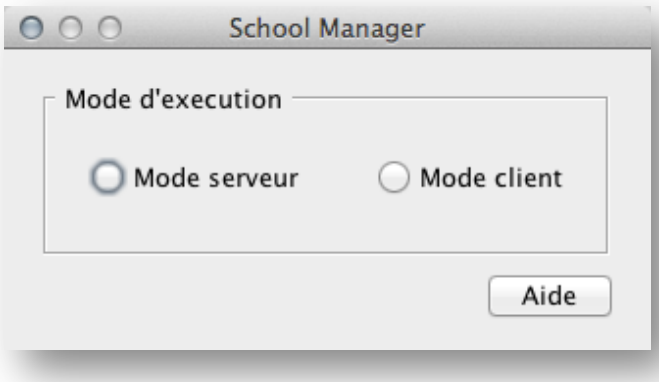
Enfin, la classe Packet est le noyau de toutes ces classes. C'est cet objet qui transitera entre les deux parts (client et serveur) afin d'apporter des informations etc. Il est constitué de plusieurs attributs permettant de faire transiter ce que l'on veut à travers le réseau (exemple : une liste de cours enseignés à l'école). Cette classe va notamment servir dans ServerCommands, dans Client et dans ServerClient.

Nous ne rentreront pas dans le détail de la lecture de données des sockets en Java car cela entrainerait de lourdes explications…

L'interface client – serveur

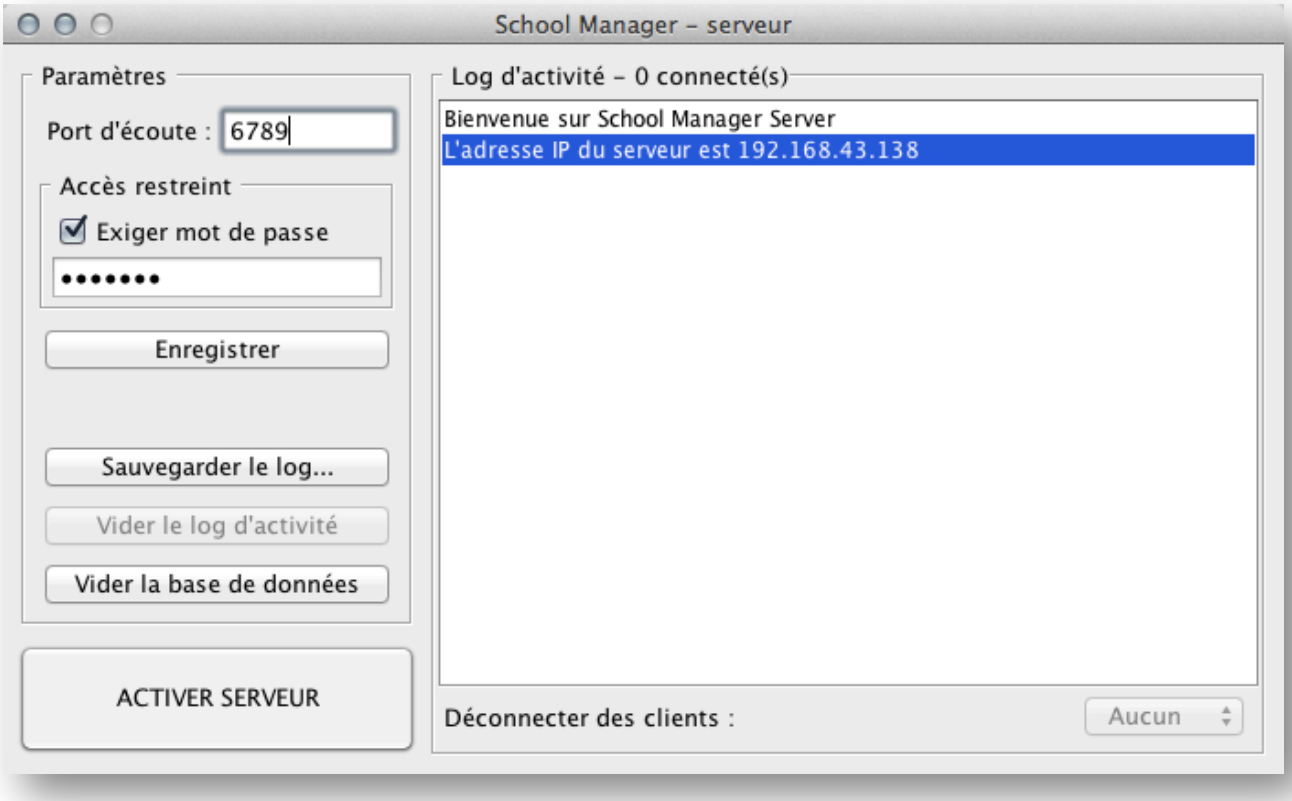
Lancement du programme

La particularité de notre programme est qu'il ne comporte pas deux programme séparés : un client et un serveur. Non. Il n'inclus qu'un seul programme Java qui lui même propose, via un menu « Splash Screen », d'accéder à la partie serveur ou à la partie client (un peu comme dans les jeux vidéos compatibles réseau).

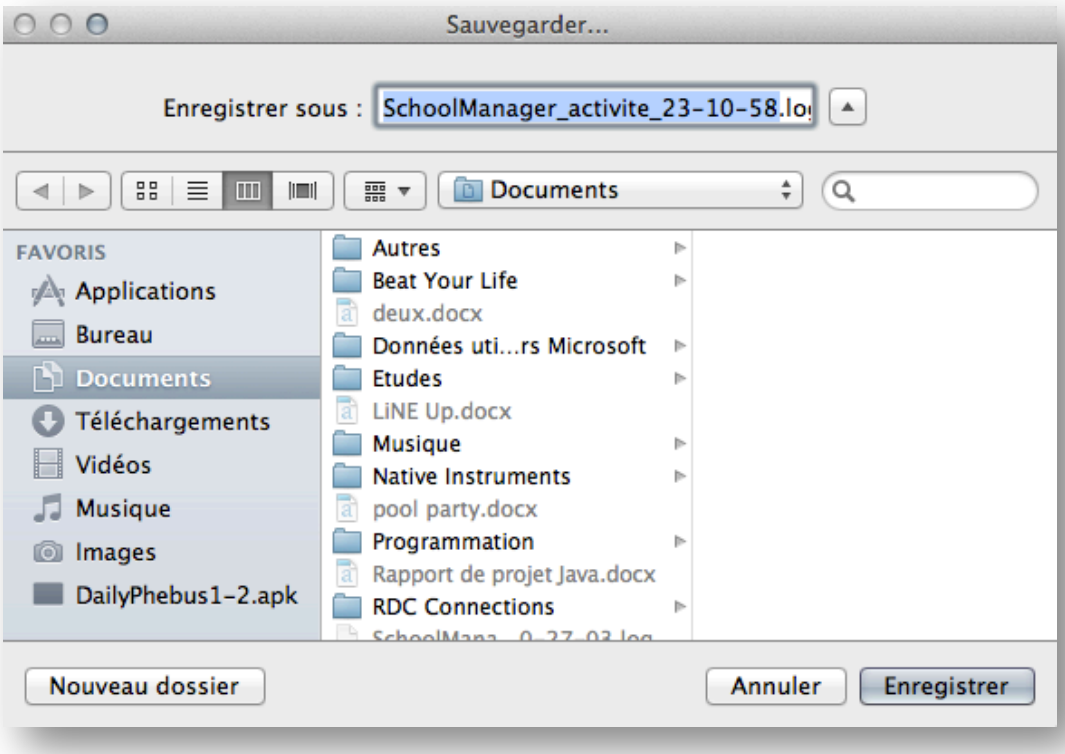


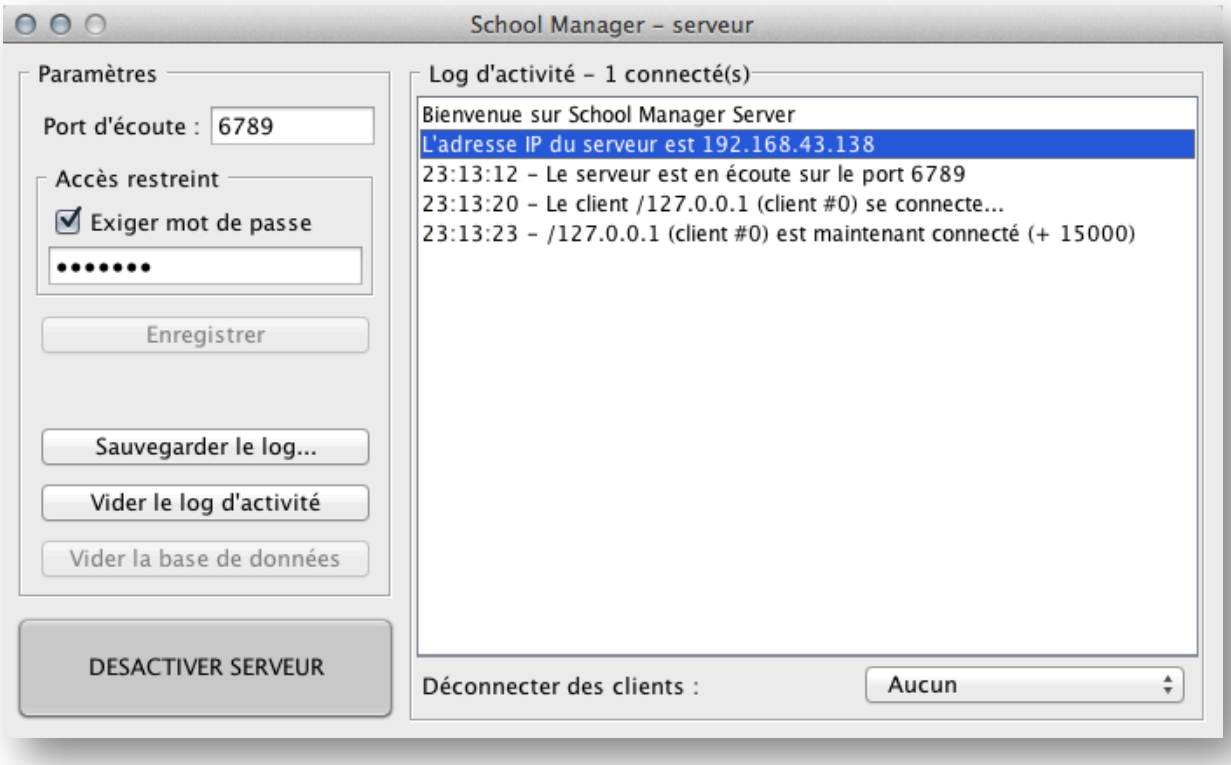
Côté serveur

Côté serveur, nous avons programmé une interface complète pour gérer l'intégralité des échanges entre le serveur et ses clients. En cela où l'administrateur serveur pourra voir ce que fait chaque client, il pourra imposer un mot de passe pour la connexion, vider la base de données (qui se régénèrera toute seule à la prochaine exécution côté serveur), voir les actions réseaux sous la forme d'un journal (ou « log ») et même enregistrer celui-ci sur l'ordinateur. Il pourra également décider de bannir un client : jugeant que celui-ci ne respecte pas les règles, pourquoi pas ?

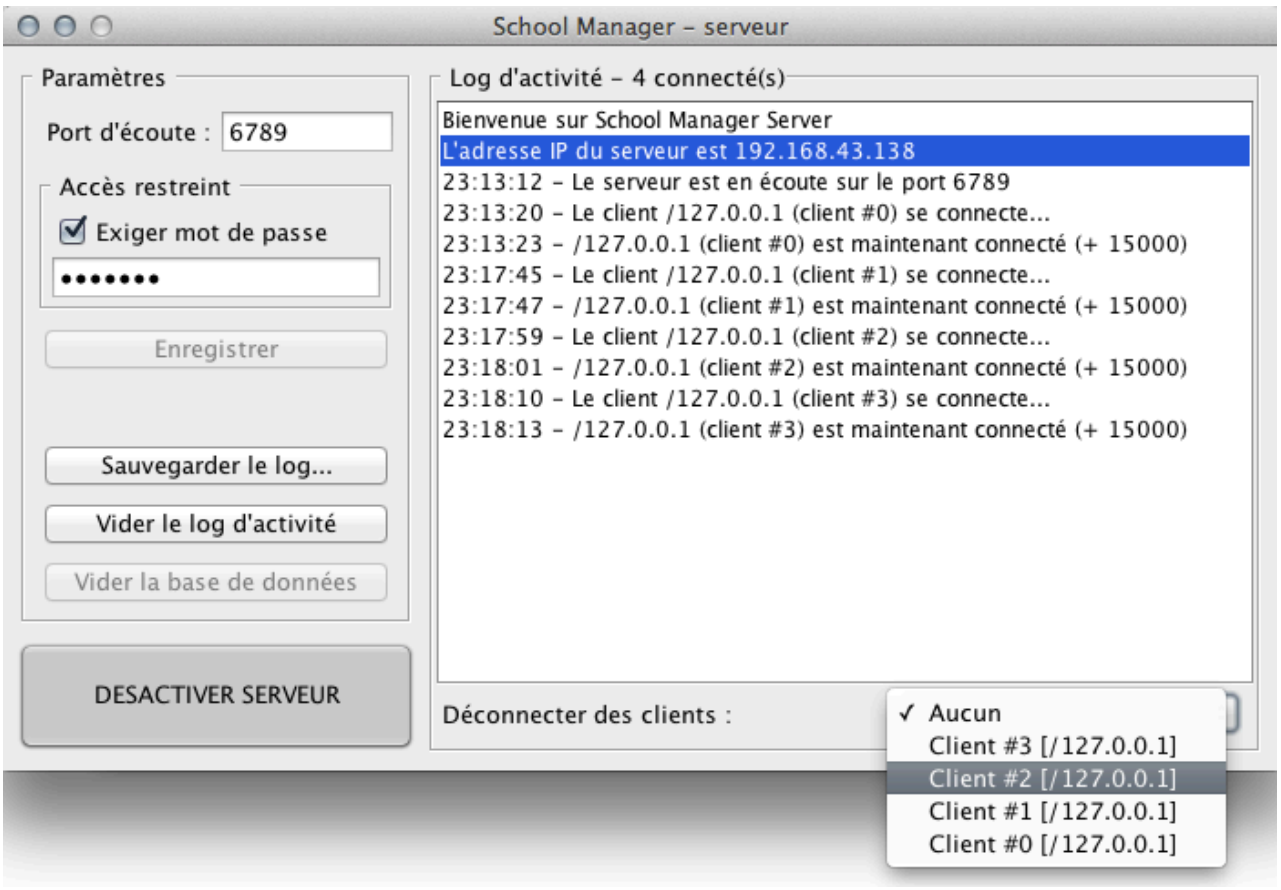


Sur cette dernière capture, le serveur n'est pas encore activé. Pour l'activer, il suffit de cliquer sur le bouton « Activer Serveur » en bas à gauche. Comme on le voit, l'administrateur peut fixer les paramètres comme le port d'écoute ou encore le mot de passe (il peut d'ailleurs ne pas en mettre du tout auquel cas la connexion client se fera toute seule).
Voici la boîte de dialogue pour enregistrer le log du serveur. Encore une fois, l'interface est claire et intuitive pour l'utilisateur.



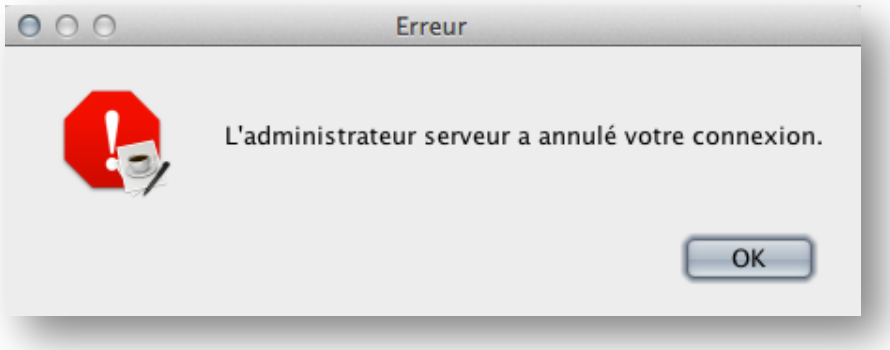


Sur cette capture d'écran, on voit l'interface serveur une fois l'écoute activée (bouton activé en bas à gauche). Ici, un client vient de se connecter. Il a pour IP 127.0.0.1 (adresse locale) et comme numéro 0 (client #0). Chaque client se voit attribuer un identifiant pour que l'administrateur puisse gérer ceux-ci plus facilement. L'interface du journal précise également les temps (ce qui est important dans ce genre de situation car en cas de problème, on peut savoir qui à fait quoi grâce à un moment donné).



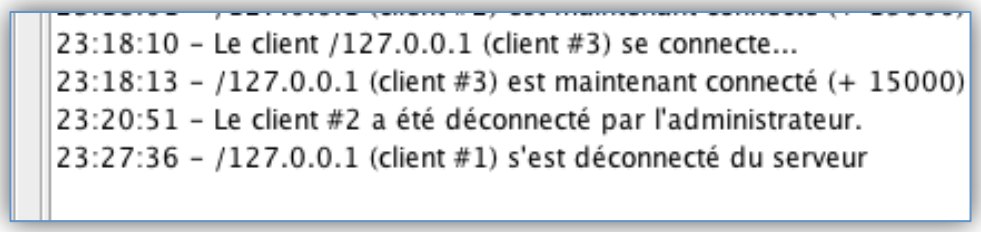
Ici, on peut voir que quatre clients sont connectés au serveur (du client #0 au client #3). En bas à droite, l'interface serveur permet de bannir les clients actuellement connectés. Par exemple, si on clique sur le client #2, celui-ci sera déconnecté et une alerte s'affichera sur son interface client. Essayons.

Les boutons de l'interface client se « grisent » et le client peut voir :



La synchronisation client – serveur a été un élément primordial. Elle nous a fait passé beaucoup de temps sur des exceptions et des erreurs non traitées. Ici, toutes les erreurs de connexion sont traitées pour que les incidents ou les déconnexions soient claires et bien formulées que ce soit chez le client ou au niveau du serveur.

Enfin, dans le cas d'une déconnexion du client (que ce soit avec la croix de fermeture, avec un raccourci de fermeture ou encore avec le bouton de déconnexion), le serveur en ait directement informé via le log :



Voilà, c'est à peu près tout concernant l'interface du serveur. A présent, intéressons-nous à l'interface du client :

Côté client

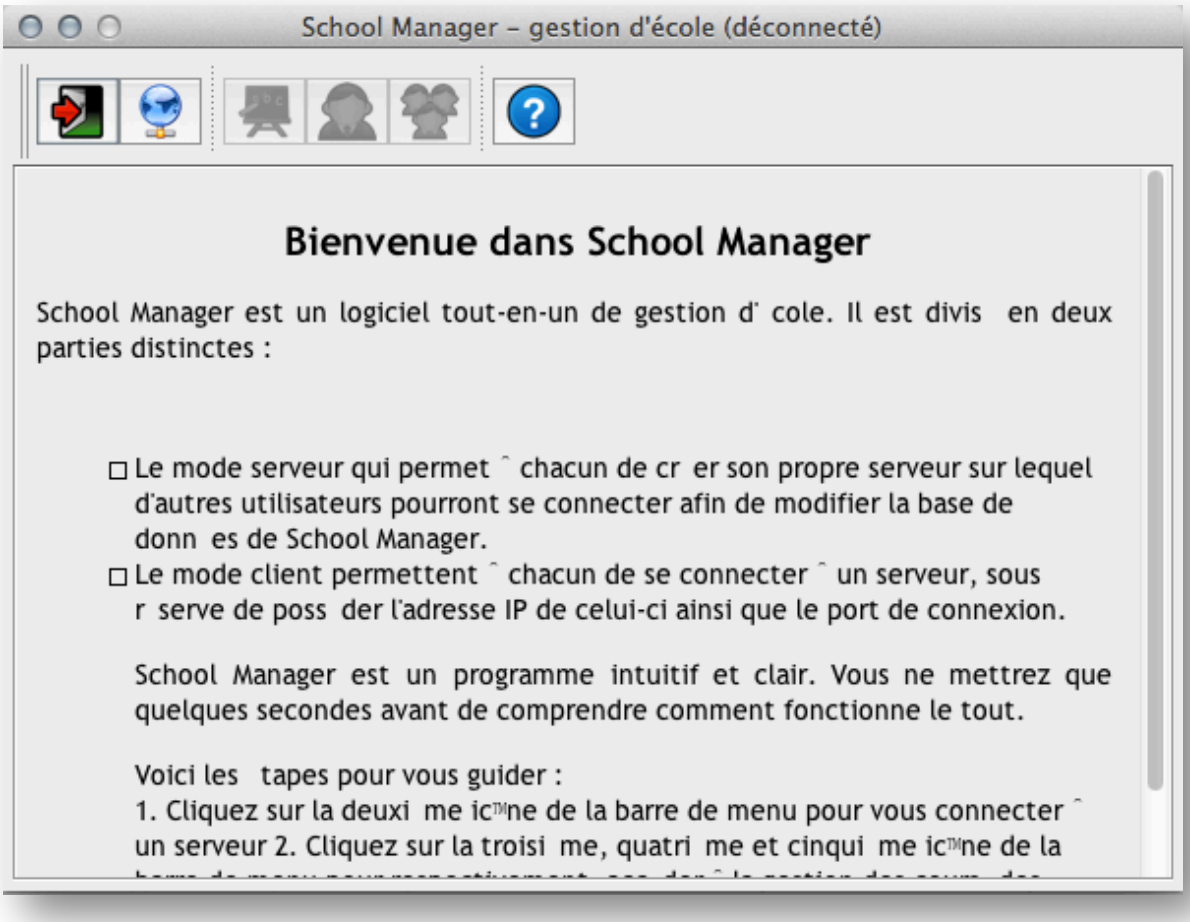
Le côté client est très important puisque, contrairement au côté serveur, ses utilisateurs seront non seulement beaucoup plus nombreux mais aussi beaucoup moins « expert en informatique ». L'interface doit donc être :

- Sobre et épurée
- Simple et intuitive
- Aidante

Ecran de base

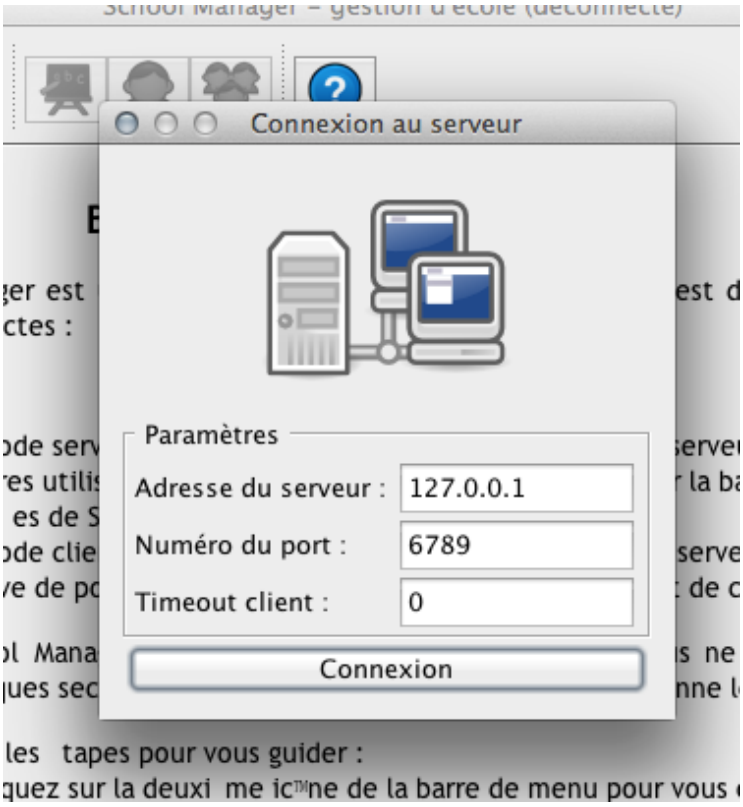
Ci-dessous l'arrivée à l'interface client. Elle est très simple : une barre d'outil en haut avec des icônes pour la clarté (et des bulles d'aide lorsque l'on passe la souris sur chaque icône). Au centre, nous avons décidé de placé un navigateur (très simplifié puisqu'il s'agit d'un `JEditorPane` qui ne prend en charge que le HTML 3.2 qui commence à dater, c'est le moins que l'on puisse dire !). Ce navigateur ne fait que lire une page web HTML située en locale sur l'ordinateur : on pourrait aussi lui faire charger une page de « news » sur le Net. En haut de la fenêtre, le nom du programme suivi par l'état actuel (déconnecté ici par exemple).

Le client étant déconnecté (voir état sur la barre de titre), les icônes de gestion des cours, des professeurs et des étudiants ne sont pour le moment pas disponibles (d'où le fait qu'elles soient grisées). La première icône du menu permet de quitter le logiciel, la deuxième permet de se connecter. C'est ce que nous allons voir maintenant.



Connexion au serveur

Pour se connecter, le client doit cliquer sur la deuxième icône. Apparaît alors une boite de connexion à trois paramètres. Comme déjà indiqué précédemment, il s'agit de l'adresse du serveur et du port d'écoute. Nous avons également ajouté un autre champ facultatif : le timeout. C'est le temps de non-réponse-serveur à partir duquel le client devra considérer la connexion avec le serveur perdue (si égal à 0, alors temps illimité).

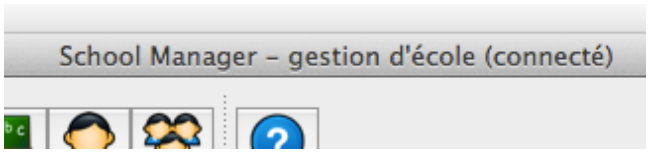


Lorsque l'on valide le formulaire, on demande à l'utilisateur s'il désire sauvegarder ces données en mémoire (pour une prochaine connexion). Puis, en fonction de si l'administrateur du serveur a demandé un mot de passe pour les connexions, une boîte de dialogue apparaît pour demander le mot de passe serveur au client. Si le client entre un mauvais mot de passe, la boîte continue d'être affichée. S'il annule, on ferme et on revient à l'écran de base.

Connecté au serveur

Si le mot de passe est correct, alors le client est connecté au serveur ! Du côté serveur, la nouvelle est affiché dans le journal. Du côté client, les icônes autrefois grisées, se voient à présent illuminées de couleurs car accessibles au clic !

On peut également constaté que le statut a changé sur la barre de titre. Il est désormais passé en mode « connecté ».



De plus, la deuxième icône a changée ! Etant donné que le client est connecté, il ne lui ai plus possible de se connecter à nouveau. Cependant, il est possible de se déconnecter du serveur en cliquant sur cette nouvelle icône :

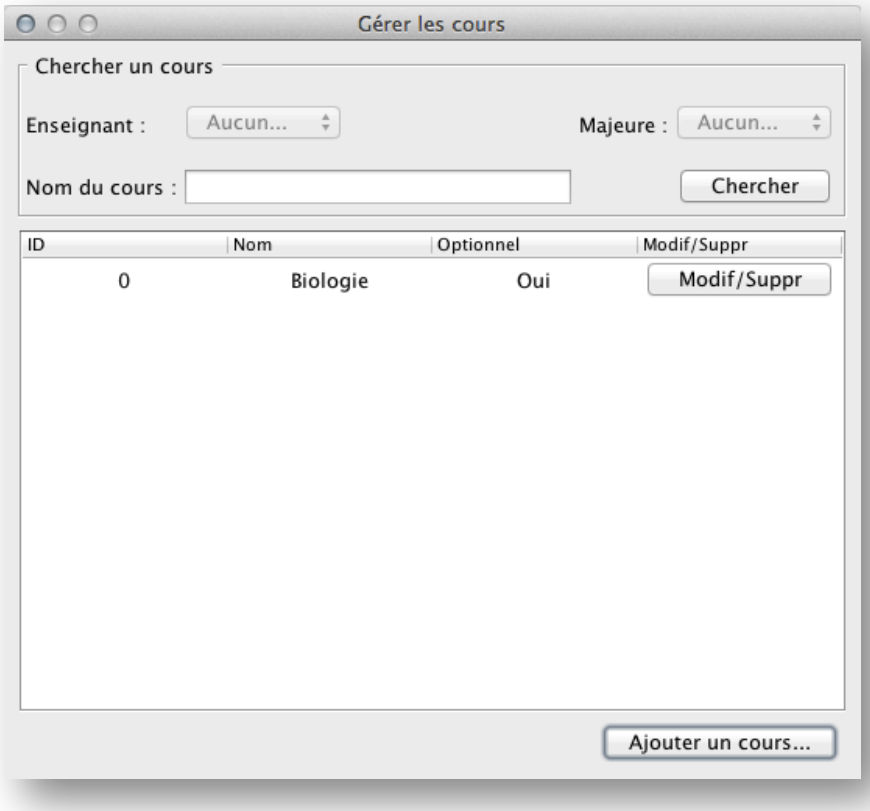
Selon si l'utilisateur choisi de se déconnecter, il est ramené à l'interface de base « non-connecté » ou la boîte de confirmation se ferme simplement.

Accéder aux cours disponibles

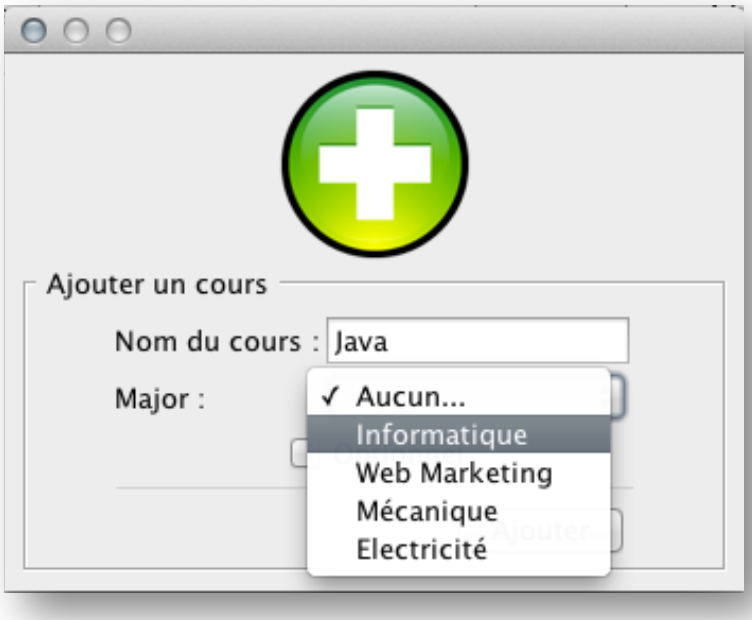
Pour accéder aux cours, encore une fois, c'est très simple. Il suffit de cliquer sur la troisième icône (celle avec un « tableau » vert dessiné dessus).

On accède alors à une fenêtre listant les cours disponibles. Ici par exemple on a un cours de Biologie.

Maintenant, insérons un cours de notre matière préférée : le Java !



Pour ce faire, le client doit cliquer sur le bouton en bas à droite de la fenêtre. Une fois ajouté, le cours est envoyé par Packet au serveur, lu, puis insérer en base de donnée. Pour éviter de gaspiller de la bande passante inutilement, le tableau de données (JTable) est mis à jour automatiquement sans avoir à rafraîchir toutes les données précédemment stockées.



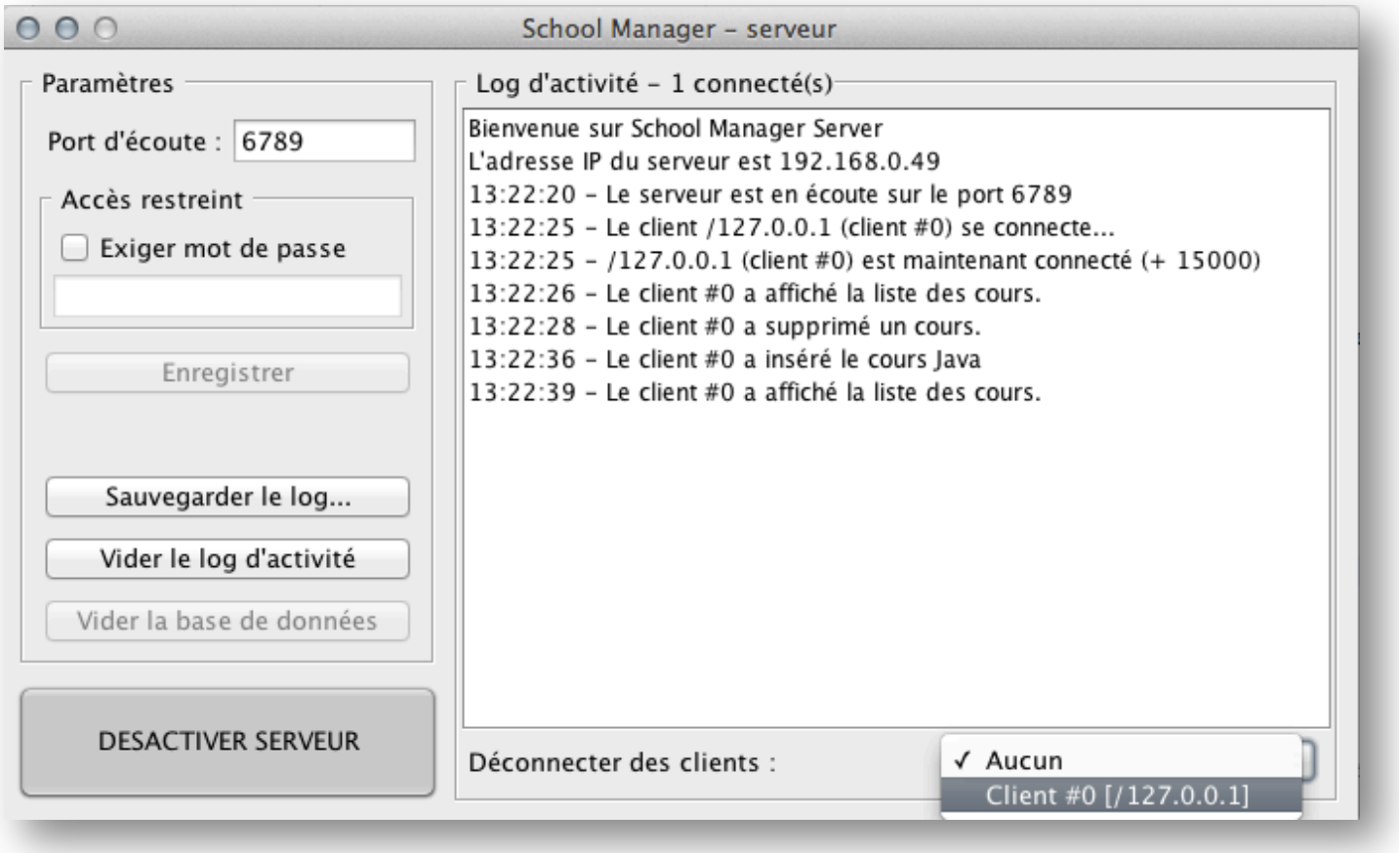
Un petit clic sur « Ajouter » pour voir ceci à l'écran :

ID	Nom	Optionnel	Supprimer
1	Anglais...	Oui	Supprimer
5	Allemand...	Oui	Supprimer
6	Sport...	Oui	Supprimer
7	Java...	Non	Supprimer

Ensuite. Vous l'aurez deviné, lorsque l'utilisateur clique sur les boutons « Supprimer», une fenêtre s'ouvre proposant le changement de nom d'une matière ou encore sa suppression. Rien de bien compliqué là-dedans.

Vous l'aurez également remarqué en voyant le formulaire dans l'entête de la fenêtre des cours, l'utilisateur peut aussi faire des recherche pour n'afficher que les cours qu'il souhaite (en fonction de plusieurs critères comme le nom (utilisation de LIKE avec « % » pour le joker SQL), l'enseignant ou encore la majeure associées. Ces deux derniers critères sont d'ailleurs chargés dans des listes de menu dès l'ouverture de la fenêtre.

Enfin il est désormais possible de supprimer un cours en cliquant sur le bouton « Supprimer » à côté de chaque cours. Le client envoi un ordre de surppression au serveur. Le cours disparaît de la liste et le serveur enregistre l'action dans le log.



Conclusion

Ce projet a été très intéressant pour nous car nous avons eu la chance d'apprendre un grand nombre de fonctionnalités de Java (gestion des fichiers, maintenance d'une session entre client – serveur, utilisation des Threads, utilisation des paquetages, utilisation d'une base de donnée et de requêtes SQL la mise en place de structure de classes (avec héritage, polymorphisme et surcharge de méthodes). Personnellement j'ai beaucoup aimé le projet car je suis moi-même très proche de Java, c'est un de mes langages préférés (peut-être à cause de sa simplicité et de sa portabilité je ne sais pas). Je l'avais déjà manié à de multiples reprises que ce soit pour faire mon application Android (http://camasoft.debatz.fr/applications.html) ou encore des jeux en Java étant plus jeune.

Cela dit, le projet s'est avéré très long à développé. C'est d'ailleurs pour cette raison que nous n'avons pas pu le terminer. J'espère que votre notation se basera davantage sur la connaissance de Java plutôt que sur l'exhaustivité du projet. Enfin, j'ai également essayé d'implémenter diverses fonctionnalités hors projet, notamment au niveau de l'IHM du serveur permettant de mettre l'accent sur le monitoring de l'application au niveau serveur… indispensable selon moi.