Hamed Brahane KY
Yann MAHE
Julien GUERY

s2-p22-membres@forge-pp.telecom-bretagne.eu

Développement d'un clone libre de Dropbox®

Rapport Technique V1

18/05/2011

Projet Développement - Groupe 22

Année scolaire 2010 - 2011

À destination du groupe de pilotage

Résumé

Dans l'environnement virtuel qui s'offre à nous, de plus en plus d'informations sont amenées à être partagées entre différents acteurs. Comment partager ces données ? Ceci est une problématique du monde dans lequel nous habitons. Les premières solutions sont maintenant loin derrière nous, avec par exemple l'utilisation du fax. D'autres méthodes sont devenues plus courantes : envoyer un e-mail, faire passer ses données de main en main via un support USB, ou encore stocker ses données sur un site web de stockage pour que d'autres puissent les télécharger. C'est ainsi que le logiciel que nous proposons apporte un autre moyen de partage. Il permet à plusieurs utilisateurs d'un réseau de synchroniser des fichiers en temps réel, assurant un partage instantané de l'information.

Bien qu'existant déjà sous forme de logiciel propriétaire, notre solution, sous licence libre quant à elle, a l'avantage de pouvoir se déployer sur n'importe quel réseau. Il suffit d'y installer un serveur, par lequel passent toutes les données synchronisées entre les différents clients qui y sont connectés. Si le réseau est local, alors cette solution assure une sécurité et une confidentialité des données beaucoup plus importante que si les données étaient envoyées sur Internet. En outre, une interface web déployée avec le serveur permet une administration aisée de ce dernier. C'est en effet par elle que de nouveaux utilisateurs peuvent s'ajouter à une synchronisation, ou que sont choisis les administrateurs des différents répertoires synchronisés.

Le produit final est satisfaisant, mais souffre de bogues encore non résolus quand le trafic de données avec le serveur devient important.

Abstract

In the environment we live in, more and more information are being shared between different participants. How to share those data? This is a question seeking an answer nowadays. The first solutions are now far behind us, for example we used to send information by fax. New methods have become more common: sending e-mails, handing data to someone using USB devices, or storing data on a web site so that others will be able to download it afterwards. Thus, the software we are introducing brings another means to share. It allows many users of an network to synchronize files in real time, so that data is being shared instantaneously.

Although such a product already exists as proprietary software, our solution, this one open-source, has the advantage that it can work on any network. One just has to install a server on it. Clients then log onto the server, and every piece of information synchronized between connected clients go by the server. If the network is a local one, then this product guarantees a safety and privacy about data being shared way more important than if data were send into the Internet. Furthermore, a web interface makes the server easy to manage. Indeed, thanks to this latter, new users can be added on a synchronized directory, and administrators of the different shared directories can be determined.

The final product is satisfying, but is still suffers bugs when the data traffic with the server becomes important.

Rédacteur : Julien GUERY
Relecteur : Yann MAHE

Sommaire

1. INTRODUCTION

Dropbox[1] est un service de stockage et de partage de fichiers en ligne proposé par l'entreprise *DropBox Inc.* Il permet de synchroniser automatiquement avec un serveur distant des répertoires choisis sur plusieurs machines utilisateurs. Le serveur distant, qui stocke les fichiers, est un serveur appartenant à DropBox Inc. Les fichiers deviennent donc accessibles par le web via n'importe quel navigateur internet, mais aussi en utilisant une application cliente multi-système d'exploitation (Linux, Mac OS, Microsoft Windows, iOS …).

Les intérêts d'une telle application sont évidents : la synchronisation automatique de ses répertoires avec un serveur distant protège les données quasiment en temps réel, et elle permet de travailler à partir de plusieurs machines sur les mêmes fichiers de manière complètement transparente (sans avoir besoin de se connecter à un serveur pour accéder à des documents).

Un des gros inconvénients de Dropbox est qu'il impose d'avoir une totale confiance dans le gestionnaire du serveur central qui stocke les fichiers. Ce serveur, hébergé par Dropbox Inc, pourrait à tout moment accéder à vos données confidentielles. De plus, le logiciel est propriétaire et devient payant au-delà de 2 Go d'espace de stockage. Le logiciel souffre d'autres défauts, en particulier le fait qu'il ne soit pas possible de configurer et personnaliser très précisément les répertoires à partager.

C'est dans le soucis de remédier à ces quelques défauts que s'inscrit ce projet : concevoir et réaliser un logiciel capable de mimer les fonctionnalités de Dropbox à ceci près que qu'il utilisera un serveur local, et aura donc l'avantage de la confidentialité des données synchronisées et d'un déploiement sur un réseau local. Il pourrait donc devenir très intéressant, pour des collectivités telles que des universités, d'utiliser notre produit. Un groupe d'étudiants travaillant sur un même projet pourrait par exemple créer un nouveau dépôt pour partager des fichiers sur différentes machines du réseau de l'université.

Le document qui suit revient sur le travail réalisé au 18/05 pour répondre à la problématique de notre client Matthieu Arzel.

Ce rapport se décompose globalement en trois grandes parties : la première aborde la solution retenue, son architecture et ses cas d'utilisation. La seconde partie approche en détail son fonctionnement et sa dynamique à travers l'analyse des classes implantées dans le code source. Puis enfin dans la troisième partie, nous montrons l'adéquation du produit final au besoin évoqué par le client.

Rédacteur : Yann MAHE Relecteur : Hamed Brahane KY

2. PRÉSENTATION GÉNÉRALE DE LA SOLUTION RETENUE

Le projet est intitulé *Développement d'un clone libre de DropBox*, et consiste en la mise en place d'un système de stockage, de partage et de synchronisation automatique de fichiers, tel que *DropBox*, mais avec le gros avantage de la confidentialité des données assurée grâce au déploiement du serveur de stockage dans un réseau local.

2.1 Retour sur le besoin et les contraintes du client

Lors de la première phase du projet, le client nous a exposé clairement son besoin et a imposé certaines contraintes dans le produit à développer :

- Le produit se constituera de deux applications distinctes : une application cliente et une application serveur.
- Le principe consiste en ce que le serveur interagisse avec les clients pour synchroniser leurs fichiers. Ces fichiers sont stockés sur un serveur SVN qui n'est accessible que par le serveur. Dès qu'un client détecte une modification sur un répertoire synchronisé, il en fait part au serveur. Ce dernier la sauvegarde sur le serveur SVN, et prévient à son tour tous les autres clients pour qu'ils soient à jour.
- Le logiciel devra fournir les moyens nécessaires pour que l'accès aux fichiers se fasse de manière sécurisée. Si besoin est, des protocoles sécurisés seront utilisés pour la communication entre le serveur et les clients.
- Des interfaces graphiques de configuration devront permettre la manipulation ergonomique du client et du serveur.
- Le système (client et serveur) devra être portable et fonctionnera à priori sur les trois systèmes d'exploitations les plus répandus à savoir : Microsoft Windows, Linux, et Mac OS.
- Le projet s'inscrit dans un cadre de logiciel libre. Aussi, tous les outils utilisés lors du développement devront être de licence libre.

A ces contraintes, s'ajoutent les contraintes de délais et de réalisations disponibles dans le cahier des charges.

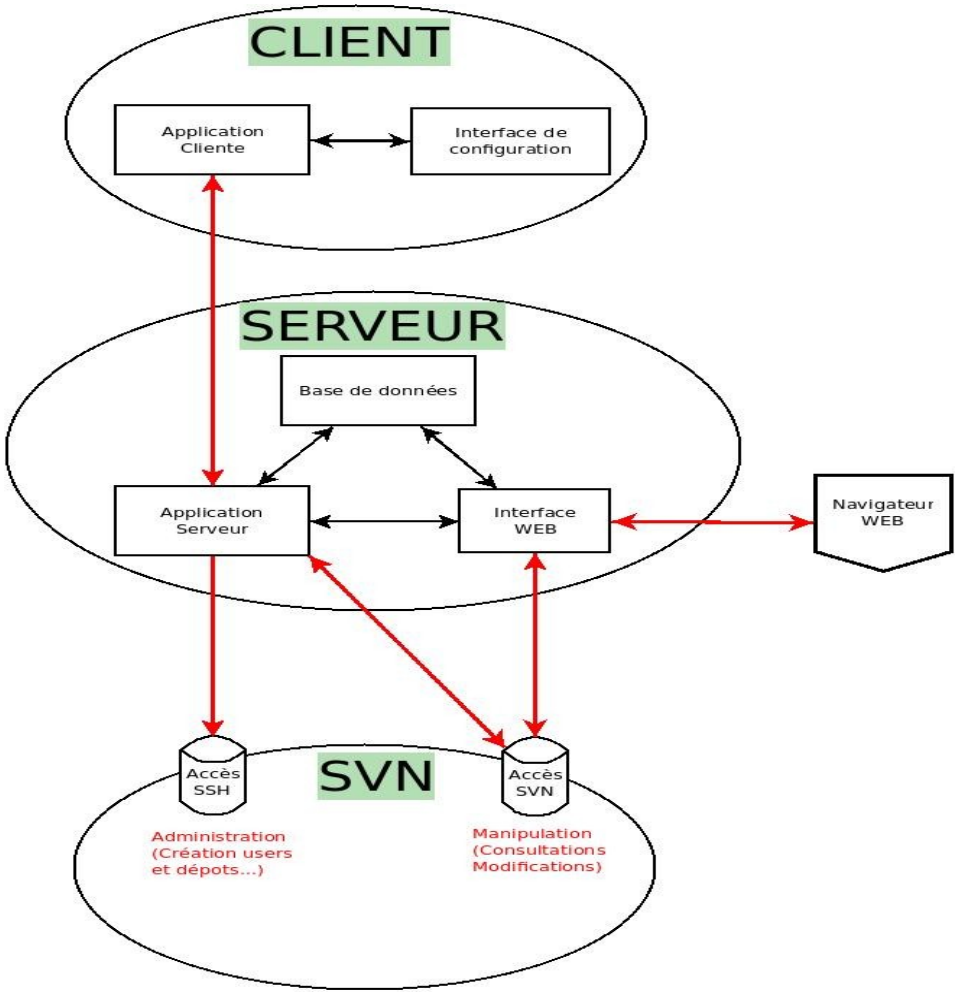
2.2 Architecture externe du système

Compte tenu des besoins et des contraintes évoqués par le client, nous avons décidé des différents acteurs qui constitueront le produit final à fournir :

- Une application serveur : elle sert de centre de synchronisation pour tous les clients.
- Une application cliente : elle reste connectée au serveur pour lui envoyer les détections et recevoir les changements.
- Un serveur SVN : il sert à stocker les fichiers synchronisés, et par la même occasion de gestionnaire de versions.
- Une base de données : elle permet de stocker les informations sur les utilisateurs du système et les dépôts synchronisés.
- Une interface web : elle permet de consulter l'historique des synchronisations et sert à ajouter des nouveaux utilisateurs au système.

Tous ces modules ont été développés séparément tout en gardant les transitions et interactions qui existent entre eux.

Le schéma suivant récapitule tout cela, en précisant en rouge les transitions entre différents acteurs et en noir les transitions au sein d'un même acteur.

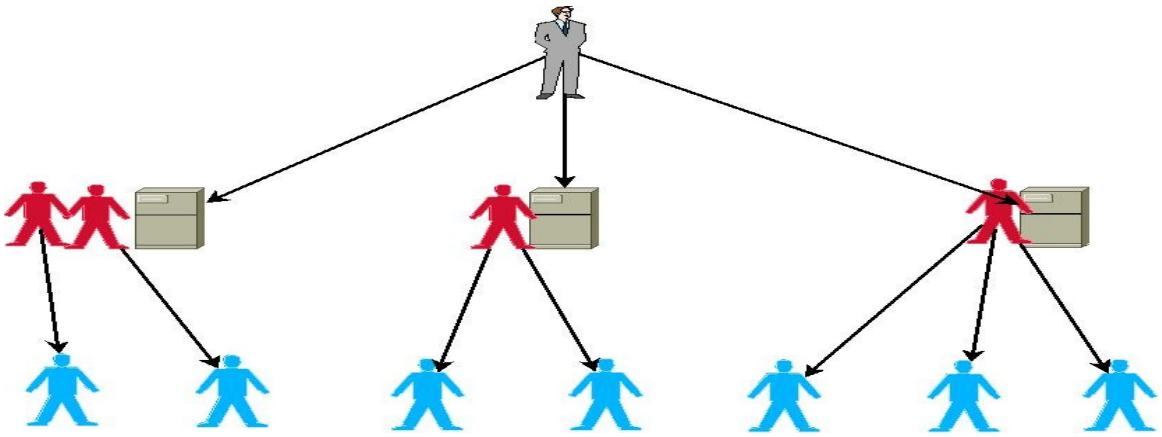


Acteurs mis en jeu dans la solution retenue

Aussi, pour assurer une certaine organisation, nous avons établi une hiérarchie dans les utilisateurs du système. Ainsi, il existe trois types d'utilisateurs :

- Les super-administrateurs : ces utilisateurs sont ceux qui disposent de tous les droits sur le système. Ils peuvent créer de nouveaux dépôts sur le serveur SVN, en supprimer, créer des utilisateurs, et attribuer des droits de super-administrateurs à d'autres utilisateurs.
- Les administrateurs de dépôts : ces utilisateurs ont les droits seulement sur les dépôts qu'ils administrent. Ils peuvent ajouter/enlever des utilisateurs à leur dépôt, gérer les droits d'accès à leur dépôt …
- Les utilisateurs : ce sont ceux qui ne possèdent que les droits de synchronisation sur certains dépôts.

Le schéma suivant récapitule tout cela. Tout en haut, se place un super-administrateur (on aurait pu en avoir plusieurs). Celui-ci a le droit d'administrer tous les dépôts. Chaque dépôt est ensuite administré par un ou plusieurs administrateurs (en rouge). Tout en bas de la hiérarchie, se trouvent les utilisateurs, gérés par les administrateurs des dépôts.



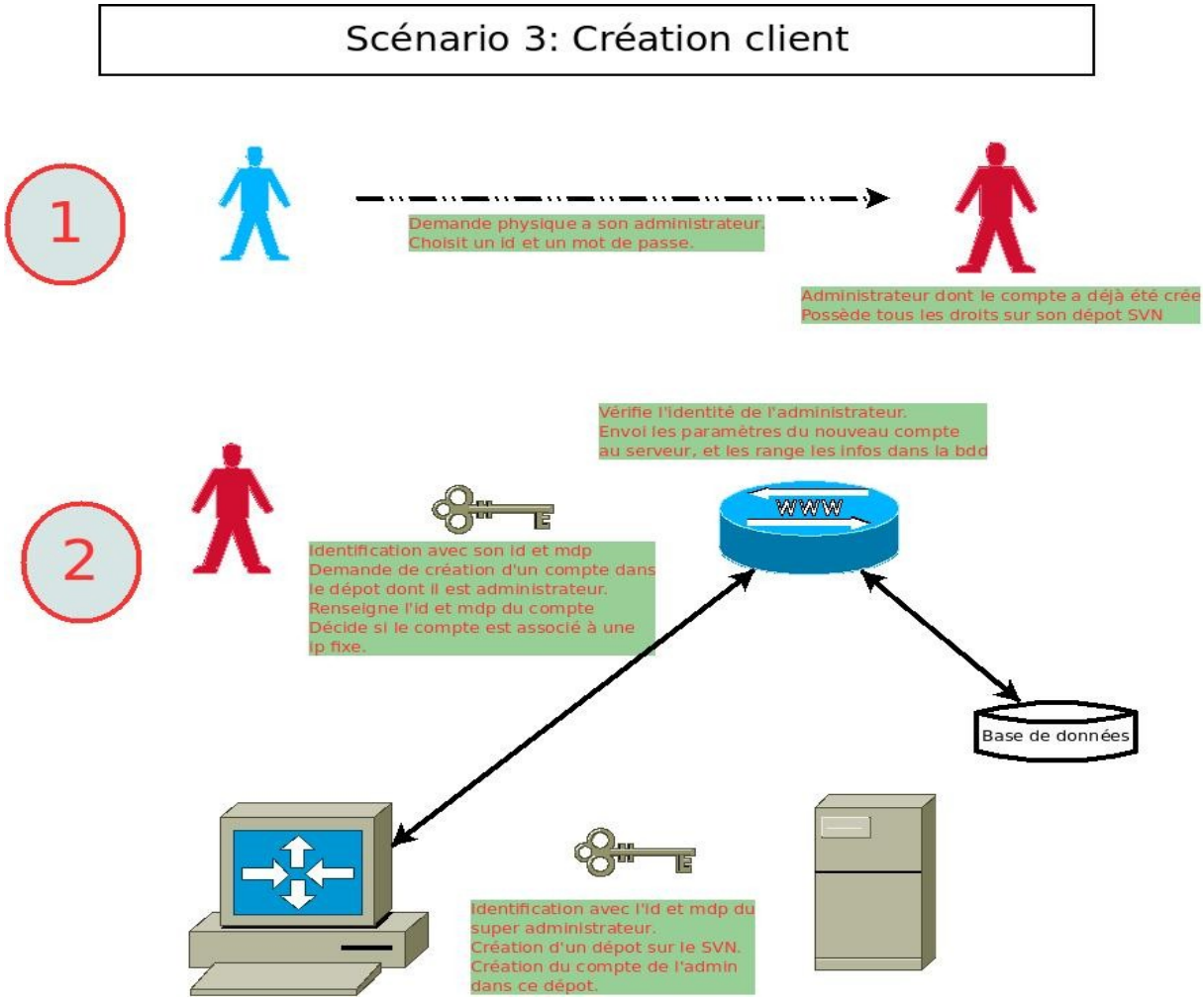
Hiérarchie des utilisateurs

2.3 Scénarios d'utilisations

Pour mettre en place un protocole de communication entre le client et le serveur, il a fallu définir les différents cas d'utilisations qui peuvent exister entre le client et le serveur.

Étant donné les contraintes de taille du rapport, nous ne présentons ici que les scénarios les plus importants :

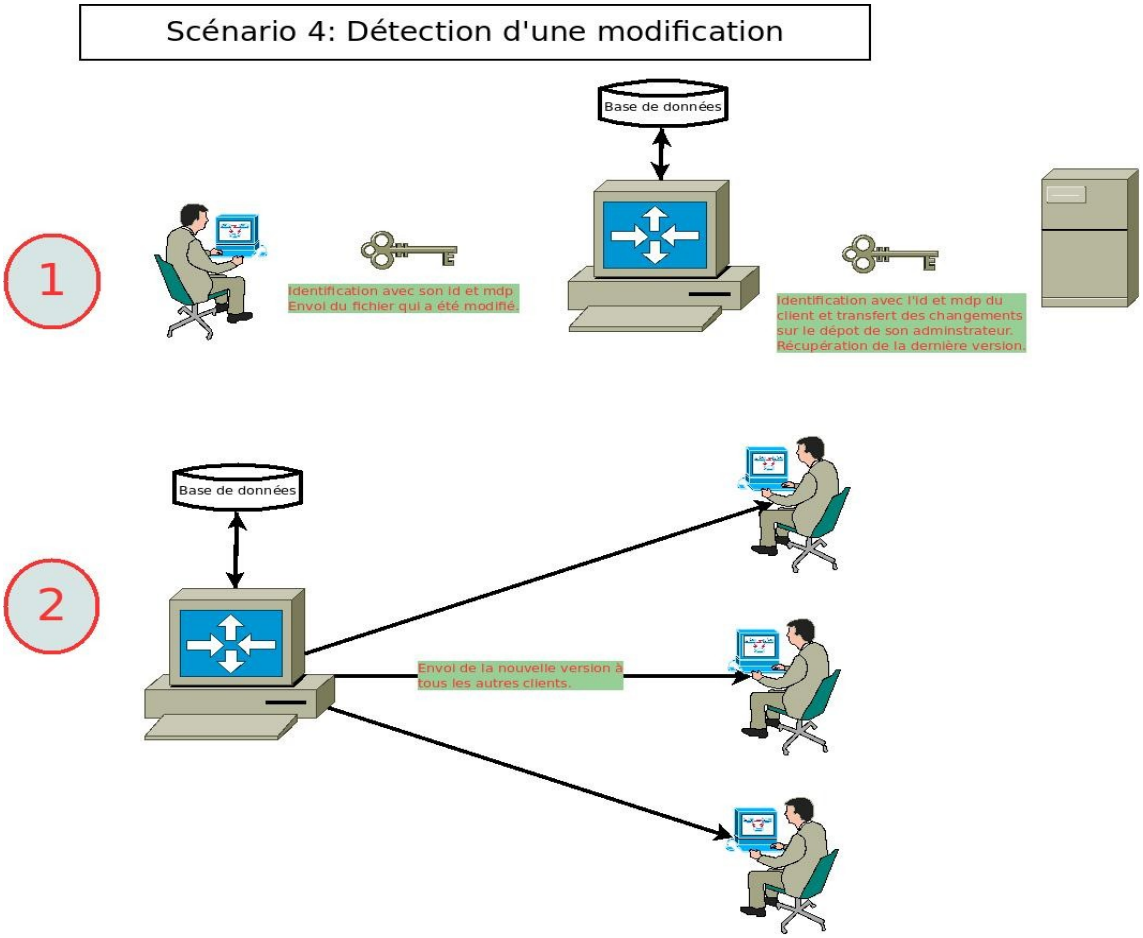
2.3.1 Le scénario de création d'un client



Création d'un client

Ce scénario intervient lorsqu'un nouveau client souhaite obtenir un compte sur le système. Pour cela, il fait une demande physique à l'administrateur du dépôt. Cet administrateur se connecte sur l'interface web, fournit les informations du compte à créer. L'interface web, stocke ces informations dans la base de données, et crée le compte sur le serveur SVN par l'intermédiaire des scripts Perl du serveur.

2.3.2 Scénario de détection d'une modification

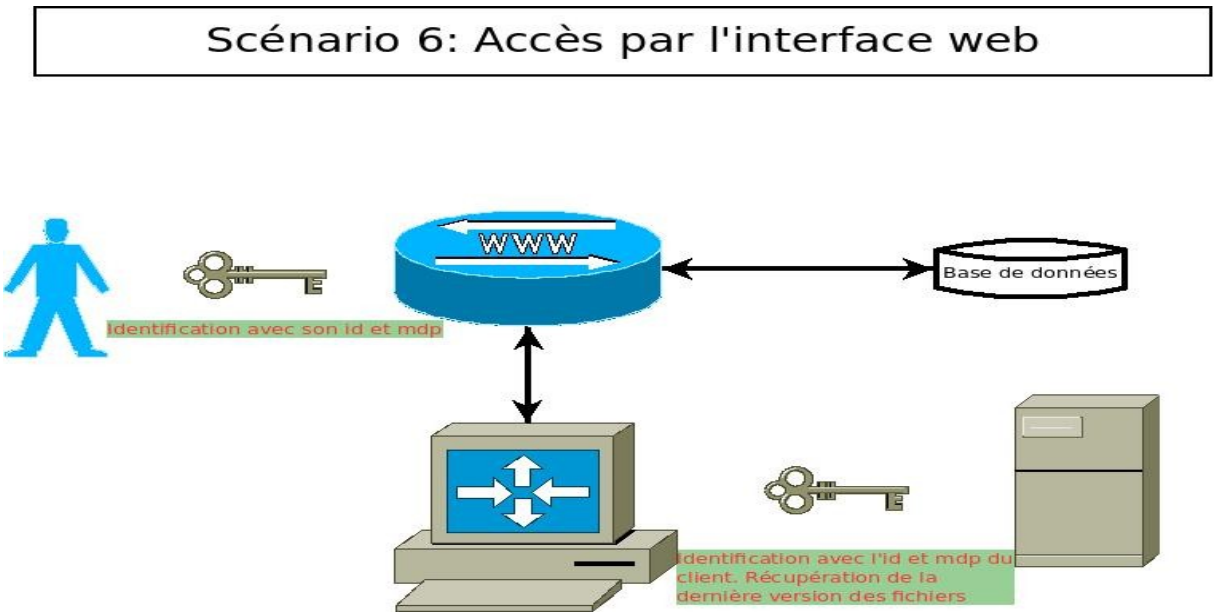


Scénario de détection d'une modification

Ce scénario intervient lorsqu'une application cliente détecte une modification dans un répertoire à synchronisé. Il envoie une requête au serveur. Le serveur vérifie d'abord l'identité du client, et vérifie que ce dernier a les droits d'écriture dans le dépôt. Il effectue ensuite la modification sur le serveur SVN, et envoie une notification aux autres clients afin qu'ils obtiennent la dernière version à jour.

2.3.3 Scénario d'accès par l'interface WEB

Dans ce scénario l'utilisateur souhaite accéder à des fonctionnalités à partir de l'interface web. Il s'identifie donc sur le serveur web, ce dernier vérifie l'identité, et en fonction de ses droits, présente les fonctionnalités auxquelles l'utilisateur a accès. Lorsqu'une requête nécessite l'accès au SVN, l'interface web se sert des scripts Perl du serveur pour télécharger la dernière révision des fichiers sur le serveur SVN.



Accès par interface web

2.4 Choix de développement

Nous avons choisi comme langage de programmation des applications serveur et cliente, le C++. Plus précisément, nous utilisons le Framework Qt[4] qui offre les possibilités suivantes :

- facilité de création d'interfaces graphiques
- un module permettant de communiquer en réseau grâce à des sockets
- créer facilement des applications multithreads
- la possibilité d'utiliser QCA[5], une API s'ajoutant à Qt permettant des fonctionnalités de cryptage avancées.

La base de données utilisée est de type SQLite[2] . Elle présente l'intérêt de s'intégrer à l'application serveur sous forme d'un unique fichier et facilite ainsi la portabilité du produit.

L'interface Web est écrite en PHP, et des scripts Perl[3] permettent de faire la communication entre l'interface web et le serveur SVN.

Rédacteur : Hamed Brahane KY Relecteur : Yann MAHE

3. ARCHITECTURE INTERNE DE LA SOLUTION

Dans cette partie, nous détaillerons le fonctionnement interne de chaque acteur du logiciel, à savoir le client, le serveur, la base de données, l'interface web, et les scripts Perl.

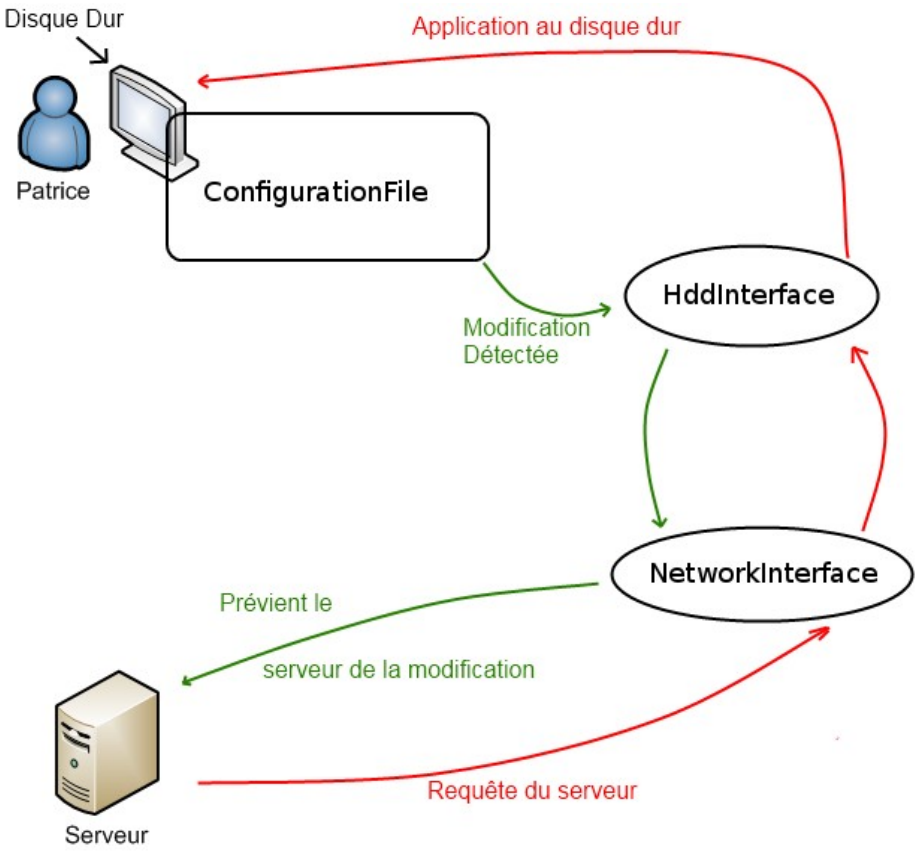
3.1 Fonctionnement et Architecture interne du client

Le tableau ci-dessous présente les fonctionnalités offertes par le client :

Nom de la fonctionnalité	Description
S'identifier auprès du serveur.	Le client est capable d'établir une connexion sécurisée auprès du serveur afin que celui-ci connaisse son identité Le protocole SSL est utilisé. Le client possède donc ses clés privés et ses certificats.
Détecter les changements dans les répertoires partagés.	Le client possède la liste des répertoires partagés sur l'ordinateur, et est capable de détecter les changements dans ces répertoires.
Prévenir le serveur d'une modification dans le(s) répertoire(s) surveillé(s) et lui envoyer la liste du/des fichier(s) modifié(s).	Lorsqu'il y a détection d'un(e) ajout/mise à jour/suppression d'un fichier, le client est capable de prévenir le serveur et de lui transférer le(s) fichier(s) concerné(s).
Recevoir une liste de fichiers modifiés correspondant à une mise à jour.	Le client est aussi capable de recevoir une mise à jour du serveur et de correctement actualiser les répertoires synchronisés.
Permettre la configuration et la personnalisation des répertoires à synchroniser	L'interface de configuration du client permet l'accès graphique à plusieurs options tel que le paramétrage de adresse serveur et la personnalisation des répertoires synchronisés.

Afin d'accomplir ces fonctionnalités, le client a été programmé avec une vision orientée objet dont l'architecture interne est subdivisée en trois grands modules :

- NetworkInterface : Ce module qui s'exécute dans un thread séparé se charge d'établir des connexions sécurisées avec le serveur, et se charge de l'envoi/réception de messages. C'est dans ce module qu'est implémenté le protocole de communication entre le client et le serveur. Ce module comporte les classes : Socket, Messages, et NetworkInterface.
- ConfigurationData : Ce module, s'exécutant dans le thread principal, permet de gérer la configuration des données. Il y a trois types de configurations : ConfigurationNetwork (pour la configuration d'accès au serveur : adresse et port), ConfigurationIdentification (pour la configuration de l'identification : pseudo et mot de passe) et ConfigurationFile. Cette dernière classe est très importante. Elle permet de gérer les détections dans les répertoires synchronisés. Pour cela elle possède une arborescence de fichiers à synchroniser (les classes Depot, Dir, et File) et permet de détecter tout changement dans un répertoire.
- HddInterface : Ce module s'exécute aussi dans un thread séparé. Malgré qu'il soit constitué d'une seule classe, il représente le cœur de l'application, car il se charge de la synchronisation des messages reçus et des détections du client. Les deux modules précédents sont reliés grâce à ce module.



3.1.1 Module ConfigurationData

Media

Cette classe représente un média sur le disque dur. Cela peut être un répertoire ou un simple fichier . Elle est justement héritée par les classes **Dir** et **File** sur lesquelles nous reviendrons plus bas. C'est une classe abstraite; elle possède des méthodes virtuelles qui seront implémentées par ses classes filles.

Attributs importants :

- *localPath :* chemin du media sur le disque dur de l'utilisateur.
- *RealPath :* chemin du media stocké sur le serveur SVN
- *detectionState :* liste de tous les changements non encore traités par lesquels est passé le media ; la liste des états possibles est :
  - MediaIsCreating : le media est en cours de création
  - MediaIsRemoving : le media est en cours de suppression
  - MediaIsUpdating : le media est encore de modification

Méthodes importantes :

- ***toXml :*** renvoie un nœud XML décrivant le média et son arborescence (sous forme d'enfants de ce nœud) s'il s'agit d'un répertoire, en précisant pour chaque média son *localPath, realPath* et sa liste *dectectionState* d'états non encore traités, séparés par des virgules. On précise pour chaque fichier son hash, réalisé en appliquant l'algorithme de hash MD5 sur le contenu du fichier.
- ***findMediaByLocalPath*** : Renvoie un pointeur vers le **Media** dont le *localPath* est celui passé en paramètre, NULL sinon. Cette fonction parcourt l'ensemble de l'arborescence d'un répertoire, si le média depuis lequel on l'appelle est un répertoire.
- **findMediaByRealPath** : Renvoie un pointeur vers le **Media** dont le realPath est celui passé en paramètre, NULL sinon. Cette fonction parcourt l'ensemble de l'arborescence d'un répertoire, si le média depuis lequel on l'appelle est un répertoire.

Dir

Cette classe hérite de **Media,** elle représente un répertoire sur le disque dur de l'utilisateur.

Attributs importants :

- *watcher :* objet de type **QfileSystemWatcher** qui surveille les changements apparus dans le répertoire. Il repère aussi les changements apparus dans les fichiers contenus dans ce dernier. A chaque changement, le slot ***directoryChangedAction*** est appelé.
- *subMedias :* liste de pointeurs vers les médias contenus dans le répertoire. Cela peut être aussi bien un fichier qu'un répertoire.
- *Listen :* booléen qui détermine si les changements apparus dans le répertoire doivent être traités ou non. Une fois mis à true, il l'est aussi dans tous les répertoires contenus dans la liste *subMedias.*
- *OldDetections :* liste de pointeurs vers des médias contenus le répertoire dans lesquels des changements sont apparus lors d'un lancement antérieur de l'application cliente mais n'ont pas encore été traités. Ces changements sont traités une fois l'attribut *listen* mis à true pour la première fois.

Méthodes importantes

- ***loadDir :*** méthode statique qui renvoie un objet **Dir** à partir d'un nœud XML. Elle remplit la liste *subMedias* par les éléments fils de ce nœud, qui sont soit des fichiers, soit des répertoires. Elle permet aussi de remplir la liste *oldDetections* du répertoire parent.
- **directoryChangedAction :** on sait qu'un changement est apparu dans le répertoire, cette méthode trouve sur quel média il a porté, et quel était le type de la modification. On signal ensuite au parent du répertoire que le média en question a subit un changement. Ce signale remonte alors jusqu'à la racine des répertoires synchronisés : le dépôt, représenté par un objet de type **Depot.**

File

Classe qui hérite de **Media**, représentant un fichier sur le disque dur de l'utilisateur.

Attributs importants :

- *hash :* signature du fichier, obtenu en appliquant à son contenu l'algorithme de hash MD5.

Méthodes importantes :

- ***loadFile :*** méthode statique qui renvoie un objet **File** à partir d'un nœud XML. Elle récupère les attributs *localPath, realPath* et ajoute le fichier à la liste *oldDetections* de son répertoire parent si l'attribut XML « detectionState » du nœud représentant le fichier n'est pas vide.

Elle dispose en outre de méthodes permettant de savoir si le fichier a été modifié ou supprimé, ainsi que de modifier ou de lire son contenu.

Depot

Classe qui hérite de **Dir.** Elle représente un dépôt synchronisé sur le serveur, c'est à dire la racine d'un ensemble de fichiers et de répertoires synchronisés avec d'autres utilisateurs.

Attributs importants :

- *readOnly :* booléen déterminant l'accessibilité en écriture de l'ensemble des médias du dépôt.
- *Revision :* numéro de révision SVN du dépôt.

Méthodes importantes :

- ***loadDepot :*** méthode statique identique à **Dir ::*loadDir***, à ceci près qu'elle retourne un objet de type **Depot.** Le numéro de révision et l'accessibilité en écriture sont en outre lus depuis le nœud XML.

ConfigurationData

ConfigurationData contient en attribut des objets de trois types différents : **ConfigurationNetwork, ConfigurationIdentification** et **ConfigurationFile.** Elle contient une méthode intéressante : ***save***, qui sauvegarde dans un document XML le contenu de ces trois classes de configuration.

ConfigurationNetwork

Elle contient simplement les informations d'accès au serveur, c'est à dire l'adresse IP et le port.

ConfigurationIdentification

Elle contient quant à elle les informations d'identification de l'utilisateur sur le serveur, son pseudo et son mot de passe.

ConfigurationFile

Attributs importants :

- *detectMediaList :* liste de toutes les modifications apparues dans un des dépôts de la liste *depots* qui n'ont pas encore été traitées.
- *depots :* liste des objets de type **Depot** synchronisés avec le serveur.

Méthodes importantes :

- ***putMediaDetection :*** lorsque un changement apparait dans un répertoire synchronisé, nous avons vu que le signal **Dir ::*directoryChangedAction*** était appelé, remontant de parent en parent. Ce signal arrive alors dans un objet de type **Depot,** contenu dans la liste *depots.* Il est alors connecté au slot ***putMediaDetection***. Si le client a les droits nécessaires pour modifier un média sur ce dépôt, la modification est alors ajoutée à la liste *detectMediaList*. Le thread de **HddInterface** est ensuite réveillé pour qu'il puisse faire part de la modification au serveur, une fois qu'il ne sera plus occupé.

3.1.2 Module NetworkInterface

Socket

Classe implémentant un socket bas-niveau assurant la liaison avec le serveur. Cette classe hérite de la classe **QsslSocket** qui crée une liaison TCP, sécurisée par le protocole SSL. Les méthodes ***connectToServer*** et ***disconnectFromServer*** permettent de se connecter ou de se déconnecter à une adresse IP et à un port que l'on précise.

La structure des paquets que nous faisons transiter entre notre client et notre serveur est la suivante :

i. un entier non signé contenant la taille totale en octet du paquet.
ii. Puis attaché à cet entier, le message en lui-même.

Lorsqu'un morceau du paquet arrive via notre socket, la méthode ***inputStream*** se charge d'attendre que tout le message soit parvenu. Elle le recompose alors et émet le signal ***receiveMessage*** contenant le message.

NB : Pour que cette connexion fonctionne, nous sommes obligés d'ignorer les erreurs SSL. En effet, les certificats que nous utilisons n'ont été signés par aucune autorité de confiance, et l'erreur « The certificate is self-signed and untrusted » obtenue à la réception du certificat du serveur, pendant la phase de « handshake » du protocole SSL, ne nous permettrait pas d'avancer si on ne l'ignorait pas.

Message

Les messages qui transitent entre le client et le serveur respectent un formalisme XML. Ils peuvent être de différentes natures (voir Annexe 1 -).

Pour écrire ces messages, nous disposons des classes **Request** et **Response**, qui hérite d'une classe **Message.** Les attributs et méthodes de ces deux classes sont sensiblement les mêmes :

Attributs importants :

- *type :* précise le type de la requête / réponse. Les différents types sont explicités ci-dessus.
- *parameters :* noms des paramètres (attributs XML) de la requête / réponse et leurs valeurs.

Méthodes importantes :

- ***toXml :*** renvoie un message XML à partir du type et des paramètres de la requête/réponse.

En outre, lorsqu'un message XML est reçu, une méthode statique ***parseMessage*** d'une classe nommée **Messages** permet d'en tirer le type et les paramètres, et renvoie ainsi un objet de type **Message**.

NetworkInterface

Cette classe est gérée par un thread indépendant. C'est par elle que passent tous les messages émis par le client et reçus de la part du serveur.

Attributs importants :

- *socket :* pointeur vers un objet de type **Socket**. C'est donc dans la classe **NetworkInterface** que la liaison avec le serveur est stockée.
- *receiveRequestList :* liste des requêtes reçues du serveur non encore traitées.
- *response :* dernière réponse du serveur à une requête envoyée par le client. Lorsqu'un message de requête est envoyé par le client, une réponse est attendue. Cette réponse arrive dans le slot ***receiveMessageAction*** et elle est alors stockée dans l'attribut *response.*

Méthodes importantes :

- ***receiveMessageAction :*** les messages XML récupérés par la classe **Socket** arrivent ici. La méthode se charge alors de déterminer de quel type de message il s'agit. Si c'est une requête (du serveur), on l'ajoute à la liste *receiveRequestList* grâce à la méthode ***putReceiveRequestList,*** si c'est une réponse (du serveur), on la stocke dans l'attribut *response.*
- ***PutReceiveRequestList :*** ajoute une requête de la part du serveur à la liste *receiveRequestList* puis réveille le thread **HddInterface** pour qu'il traite la requête, quand il ne sera plus occupé.

Cette classe dispose en outre des méthodes **sendMediaCreated, sendMediaUpdated, sendMediaRemoved** pour envoyer au serveur les modifications observées dans l'arborescence synchronisée, ainsi que de **sendIdentification** pour s'identifier sur le serveur.

3.1.3 Module HddInterface

Cette classe est centrale, une seule instance, lancée dans un thread indépendant, en est créée dans toute l'application cliente, et c'est chez elle qu'atterrissent les requêtes envoyées par le serveur, ainsi que les modifications apparues dans les dossiers synchronisés de l'utilisateur.

Les méthodes ***receiveRemovedRequest, receiveUpdatedRequest*** et ***receiveCreatedRequest*** permettent d'exécuter les requêtes reçues par le serveur en effectuant sur le disque dur les modifications demandées, sur les médias demandés.

Les méthodes ***detectedRemovedMedia, detectedUpdatedMedia*** et ***detectedCreatedMedia*** avertissent le serveur d'une modification apparue sur le disque dur. Pour cela, elles utilisent les méthodes **NetworkInterface ::*send*** vues plus haut. Celles-ci renvoient un objet de type **Response,** un message du serveur. Elle est alors analysée pour affichage au client, et la révision du dépôt auquel appartient le média concerné par la modification est mise à jour selon le paramètre contenu dans la réponse.

3.2 Fonctionnement et Architecture interne du serveur

Le tableau ci-dessous présente les fonctionnalités offertes par le serveur:

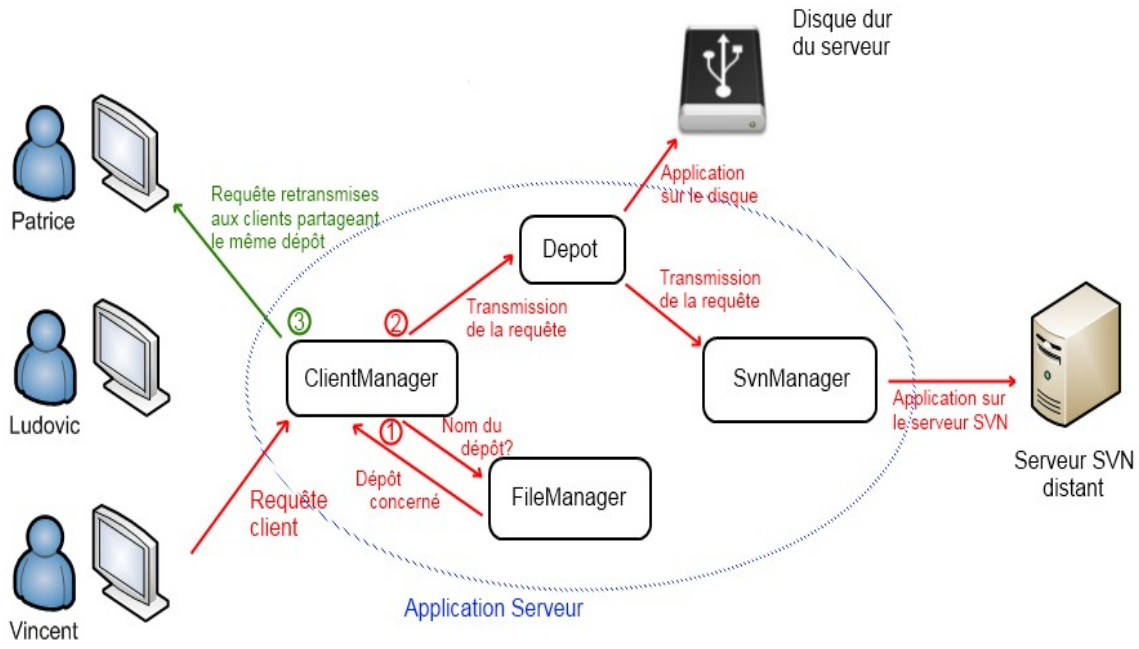
Nom de la fonctionnalité	Description
Recevoir une liste de fichiers modifiés correspondant à une mise à jour d'un client, et les importer sur le serveur SVN.	Le serveur est toujours prêt à recevoir de la part d'un client une liste de fichiers modifiés. Il est ensuite capable de transférer cette modification sur le serveur SVN.
Envoyer la liste des fichiers modifiés	Le serveur est capable d'avertir et de mettre à jour tous les clients

reçue aux autres clients.	lorsqu'une modification est reçue.
Permettre la configuration.	L'interface de configuration du serveur permet l'accès graphique à plusieurs options tel que le paramétrage du serveur SVN.

Afin d'accomplir ces fonctionnalités, le serveur a été programmé avec une vision orientée objet dont l'architecture interne est subdivisée en quatre grands modules :

- ClientManager : Ce module se charge de l'interaction avec chaque client connecté au serveur. A chaque fois qu'un nouveau client se connecte, un objet de cette classe est créé pour démarrer la communication avec le client. Chaque objet de cette classe s'exécute dans un thread différent. Cela permet donc la fluidité de l'application.
- DataBaseManager : Ce module se charge de l'interaction avec la base de données. Toutes les requêtes d'accès et de modification de la base de données passent par ce module. Il est initialisé dès le lancement du serveur. Il vérifie si une base de données existe, pour s'y connecter. Si la base de données n'existe, il est capable de créer une nouvelle base de données en créant toutes les tables et vues dont l'application a besoin pour fonctionner.
- SvnManager : Ce module se charge de faire l'interaction avec le serveur SVN. Toute requête de « Checkout », « Commit », « Update » … passe par ce module. Pour exécuter les requêtes, il se sert de l'utilitaire de ligne de commande SVN.
- FileManager : Ce module s'occupe de gérer les dépôts synchronisés. Il est chargé à partir des modules DataBaseManager et SvnManager, et télécharge les dépôts qu'il garde en local.

Un résumé graphique de ce qui a été dit se trouve ci-dessous :



3.2.1 Module ClientManager

ClientManager

Un objet de cette classe, lancé dans un thread indépendant, est alloué à chaque client qui se connecte au serveur. Il est chargé d'assurer la communication avec lui.

Attributs importants :

- *socket : objet de type* **Socket** *assurant* physiquement la liaison avec le client.
- *clients :* liste d'objets de type **ClientManager** représentant les autres clients connectés au serveur.
- *databaseManager* et *fileManager* : permettent d'avoir accès à la base de données des utilisateurs ainsi qu'aux dépôts synchronisés.
- *state :* état de la connexion avec le client, parmi la liste d'états possibles suivante :
  - CONNECTED : le client est tout juste connecté, mais pas encore identifié.
  - CLIENT_DETECTIONS : le client est en train d'envoyer ses détections.
  - SERVER_DETECTIONS : le serveur est en train d'envoyer à ce client les détections repérées chez d'autres clients.
  - SYNCHRONIZED : le dépôt du client est à la même version que le dépôt SVN.
- *upgrading :* liste de requêtes à envoyer au client.

Méthodes importantes :

- ***receiveMessageAction :*** les messages reçus du client arrivent à cette fonction. Elle le parse puis la requête est traitée dans la méthode ***receivedRequest.***
- ***receivedRequest :*** elle traite chaque requête envoyée par le client selon le protocole de communication établi entre le serveur et le client.

Classe Socket

Classe implémentant un socket bas-niveau assurant la liaison avec le serveur. Cette classe hérite de la classe **QsslSocket** qui crée une liaison TCP, sécurisée par le protocole SSL. Les méthodes ***setDescriptor*** et **disconnectClient** permettent respectivement d'utiliser le socket que le client a créé pour se connecter au serveur, et de déconnecter le client du serveur.

La structure des paquets que nous faisons transiter entre notre client et notre serveur est la suivante :

i. un entier non signé contenant la taille totale en octet du paquet.
ii. Puis attaché à cet entier, le message en lui-même.

Lorsqu'un morceau du paquet arrive via notre socket, la méthode ***inputStream*** se charge d'attendre que tout le message soit parvenu. Elle le recompose alors et émet le signal ***receiveMessage*** contenant le message.

Classe Message

Cette classe est la même que celle implémentée dans l'application client.

Classe Server

Elle hérite de la classe Qt **QTcpServer** qui comme son nom l'indique crée un serveur permettant de recevoir des connexions en mode TCP (mode connecté). Le serveur se met dans à premier temps à écouter les connexions arrivant sur un port donné, grâce à la méthode ***beginListenning.*** Lorsqu'un client souhaite se connecter au serveur, il arrive automatiquement dans la méthode ***incommingConnection.*** Un objet de type **ClientManager** est alors ajouté à l'attribut *clients,* liste de tous les clients connectés au serveur.

3.2.2 Module DatabaseManager

Toutes les informations à propos du serveur SVN et du serveur lui-même sont stockées dans une base de données SQLite créée dynamiquement par l'application serveur. Elle contient les six tables de données suivantes :

1. *infos :* elle contient 6 champs. Les ports sur lesquels sont lancés le serveur et le serveur SVN, l'adresse du serveur SVN, l'identifiant et le mot de passe nécessaire pour se connecter au serveur SVN avec tous les droits, et un chemin sur le disque dur du serveur où seront stockés temporairement les fichiers synchronisés avec le client. Ce dernier champ est ensuite enregistré dans la variable static **Depot ::***GLOBAL_DEPOT_PATH.*
2. *superadmin :* contient les logins et dates d'inscription des super administrateurs.
3. *admin :* contient les logins et dates d'inscription des administrateurs de dépôt, en précisant le dépôt qu'ils administrent. Un utilisateur peut administrer plusieurs dépôts, il suffit que son login soit présent plusieurs fois dans cette table avec les différents noms de dépôts.

4. *depot :* noms des dépôts que le serveur tient synchronisés, ainsi que leur date d'ajout à la synchronisation.
5. *user :* contient les informations de tous les utilisateurs, c'est à dire leur pseudo et mot de passe, leur nom prénom, ainsi que leur date d'inscription.
6. *utilisation :* une ligne de cette table associe à un login d'un utilisateur un dépôt sur lequel il est synchronisé. Elle précise aussi les droits de cet utilisateur sur ce dépôt (*readOnly* ou *read&write*) et la date d'ajout de l'utilisateur à la synchronisation de ce dépôt.

La classe **DatabaseManager** permet de gérer cette base de données. C'est donc elle qui peut vérifier si un utilisateur a les droits nécessaires pour effectuer telle ou telle requête, ou qui permet d'identifier un utilisateur basique ou un administrateur sur un dépôt en particulier.

3.2.3 Module FileManager

Classe FileManager

Cette classe plutôt sommaire contient la liste des dépôts tenus synchronisés par le serveur, sous forme d'objet de type **Depot.** Elle contient des méthodes qui permettent d'ajouter ou de supprimer un dépôt de la liste et du disque dur, et de retrouver le dépôt où est stocké un média dont on a spécifié le *realPath*. Son chemin sur le disque dur du serveur est alors : **Depot ::***GLOBAL_DEPOT_PATH+realPath.*

Classe Depot

Elle est un peu différente de la classe **Depot** du client, car elle permet de gérer le contenu d'un dépôt sur le disque dur du serveur ET sur le serveur SVN, en faisant appel aux méthodes de la classe **SvnManager.** La méthode **ClientManager ::*receiveRequest*** utilise ses méthodes pour appliquer les requêtes de modification de média envoyées par un client.

3.2.4 Module SvnManager

Cette classe assure la communication avec le serveur SVN, dont les informations d'accès, récupérés depuis la base de données, sont stockées dans des attributs.

Méthodes importantes :

- ***checkoutDepot, updateDepot, commitDepot, addFileToDepot*** et ***removeFileToDepot*** effectuent des actions sur des dépôts précisés en argument, avec des options aussi précisées en argument.
- ***getRequestDiff :*** renvoie une liste de requêtes qui permettront de placer le client au même numéro de révision que le serveur. Le client doit pour cela préciser le numéro de révision auquel il s'est arrêté.

Pour effectuer les commandes SVN, nous utilisons des objets QProcess, dont le fonctionnement est identique à la fonction system( ) du langage C. La commande doit donc être installée sur la machine du serveur. Cela est presque toujours le cas sur les systèmes Linux.

3.3 Fonctionnement et Architecture interne de l'interface web

Le tableau ci-dessous présente les fonctionnalités offertes par l'interface web:

Nom de la fonctionnalité	Description
Création (modification ou suppression) d'un nouvel administrateur et d'un nouveau dépôt.	L'interface web permet aux Super Administrateurs de créer un nouveau dépôt et de lui associer des administrateurs. Les informations relatives aux nouveaux comptes crées sont stockées dans la base de données. L'interface web est être capable de relayer la requête jusqu'au serveur SVN, par l'interface de scripts perl dont nous reparlerons ci-dessous.
Ajout (modification ou suppression) d'un nouvel	L'interface web offre aussi la possibilité à un administrateur d'un dépôt d'ajouter (ou modifier les droits ou supprimer) un utilisateur à son

utilisateur à un dépôt existant.	dépôt.
Création de compte et validation	L'interface web offre la possibilité à quiconque de créer son compte et de le faire valider par un administrateur. Le compte ne deviendra actif que lorsqu'un administrateur ou un super-administrateur l'aura validé.
Consulter l'historique des dépôts	L'interface web offre aussi la possibilité de consulter l'activité des historiques sur un dépôt donné.

Afin d'accomplir ces fonctionnalités, l'interface web a été développée en XHTML, CSS et PHP5.

Pour faciliter sa mise en place (installation, configuration), il a été développé de façon à ce qu'il tienne dans un seul fichier PHP nommé index.php.

Ce seul fichier contient à fois les codes HTML, CSS et PHP. Si à l'exécution on a l'impression de voir plusieurs pages différentes c'est parce que ce fichier « index.php » est capable d'adapter son contenu à partir de variables super-globales comme GET, POST, SESSION, et SERVER.

L'interface web nécessite pour fonctionner :

- Un serveur web supportant du PHP5 : Apache de préférence.
- Les bibliothèques PDO pour l'accès à une base de données SQLite.
- Les scripts Perl dont nous reparlerons ci-dessous
- L'utilitaire de ligne de commandes SVN

3.4 Fonctionnement et Architecture interne des scripts PERL

Les scripts Perls sont un ensemble de fonctions écrites en Perl et disponible dans un fichier *script.pl.* Ils permettent de faire interagir l'interface web avec le serveur SVN. Le tableau ci-dessous présente les fonctionnalités offertes par ses scripts :

Nom de la fonctionnalité	Description
Ajouter/Supprimer/Modifier un dépôt	En appelant le script avec les bons paramètres, il est capable de créer un dépôt SVN et le paramétrer de façon à ce qu'il soit utilisable par le serveur d'applications.
Ajouter, supprimer ou modifier les droits d'un utilisateur sur un dépôt.	Le script est capable de gérer les utilisateurs d'un dépôt et les droits avec lesquels ils ont accès au dépôt.
Récupérer l'historique d'un dépôt	Le script est aussi capable de consulter l'historique d'un dépôt entre deux dates passées en paramètres.

Afin d'accomplir ces fonctionnalités, le fichier script.pl a été programmé avec une vision modulaire.

Rédacteurs : Julien GUERY, Hamed Brahane KY

Relecteur : Yann MAHE

4. ADÉQUATION DE LA SOLUTION AU BESOIN DU CLIENT

Cette partie revient sur les exigences fonctionnelles ou non, qui nous ont été fixées par le client Matthieu Arzel, et citées dans le cahier des charges du 3 mars 2011. Elle fait le point sur l'état actuel de la solution par rapport aux attentes du client.

4.1 Capacités fonctionnelles de la solution

Fonctionnalités offertes par la partie client :

Fonctionnalité réclamée	Module concerné	Implémentation
S'identifier au près du serveur.	Application	Ok.
Détecter les changements dans les répertoires partagés.	Application	Ok.
Prévenir le serveur d'une modification dans le(s) répertoire(s) surveillé(s) et lui envoyer la liste du/des fichier(s) modifié(s).	Application	Ok.
Recevoir une liste de fichiers modifiés correspondant à une mise à jour.	Application	Ok, néanmoins quelques bogues surviennent lorsque la taille des fichiers excède quelques mégaoctets.
Permettre la configuration et la personnalisation des répertoires à synchroniser	Interface de configuration	Ok.

Fonctionnalités offertes par la partie serveur :

Fonctionnalité réclamée	Module concerné	Implémentation
Recevoir une liste de fichiers modifiés correspondant à une mise à jour d'un client, et les importer sur le serveur SVN.	Application	Ok. néanmoins quelques bogues surviennent lorsque la taille des fichiers excède quelques mégaoctets.
Envoyer la liste des fichiers modifiés reçue aux autres clients.	Application	Ok.
Création (modification ou suppression) d'un nouvel administrateur et d'un nouveau dépôt.	Interface Web Base de données Application	En cours.
Création (modification ou suppression) d'un nouveau client.	Interface Web Base de données Application	En cours.
Permettre la configuration.	Interface de configuration	Ok.

4.2 Exigences non fonctionnelles de la solution

Exigence	Implémentation
Sécurité	Les connexions entre les clients et le serveur sont cryptées grâce au protocole SSL. Les fichiers de configuration du client sont cryptés grâce à la librairie QCA. Le fichier contenant la base de données, chez le serveur, n'est pas encore cryptée. Nous cherchons actuellement un moyen d'y arriver.
Rendement	Ok. Le client et le serveur s'exécute en fond sans que l'utilisateur ne perçoive de ralentissement (une dizaine de mégaoctets de mémoire vive sont réquisitionnés par le client).
Maintenabilité	Ok. Le code source est commentée et des documents descriptifs (cf. partie 3) détaillant la structure et la dynamique sont disponibles.
Portabilité	Pas encore testée.
Facilité d'utilisation	Ok. Le logiciel est facile d'utilisation et ergonomique. Son interface est encore maintenue à jour.

Rédacteur : Yann MAHE Relecteur : Julien GUERY

5.CONCLUSION

A l'issue des ce trois premiers mois de projet, l'équipe 22 est très fière de présenter l'avancée de sa solution proposée au client Matthieu Arzel. La quasi-totalité de l'architecture système a été codée et déployée à savoir, l'application client, l'application serveur, et l'interface web. L'équipe achève désormais son travail sur l'interface web et notamment sa connectivité avec la base de données. D'ici la fin de la semaine, le groupe sera donc capable d'offrir une solution répondant aux exigences fonctionnelles posées dans le cahier des charges du 3 mars 2011, et à la majorité des exigences non fonctionnelles.

A ce stade, nous enregistrons donc entre une et deux semaines de retard par rapport au planning prévisionnel proposé à la mi-mars. Néanmoins ce retard est à relativiser, compte -tenu de l'avance d'un peu plus d'un mois prévu entre le 9 mai (date prévue de fin de la partie programmation) et le 15 juin 2011 (dépôt du rapport v2). Entre autre, ce temps restant nous permettra encore de corriger quelques bogues nuisibles au confort d'utilisation, et de se concentrer sur certaines exigences non fonctionnelles telles que la portabilité des applications, l'ergonomie et surtout la sécurité (cryptage de la base de données).

Rédacteur : Yann MAHE Relecteur : Hamed Brahane KY

6. BIBLIOGRAPHIE

- [1] Site officiel de Dropbox : « http://www.dropbox.com/» (consulté le 15/05/2011)
- [2] À propos de SQLite : « http://fr.wikipedia.org/wiki/SQLite » (consulté le 10/05/2011)
- [3] Apprentissage de Perl : « http://www.siteduzero.com/tutoriel-3-318130-programmation-perl-modules-cpan.html » (consulté le 11/05/2011)
- [4] Site officiel de Qt : « http://qt.nokia.com/ » (consulté le 15/05/2011)
- [5] À propos de QCA : « http://delta.affinix.com/docs/qca/index.html » (consulté le 12/05/2011)

7. GLOSSAIRE

Vous rencontrerez les termes et abréviations suivants tout au long de la lecture. Il est donc nécessaire de comprendre leur signification et le contexte dans lequel nous les employons.

Terme	**Description**
Produit	Terme générique désignant la suite complète de logiciels à fournir à la fin du projet (serveur, client, outils de configurations, interface web, scripts, base de données …).
Serveur SVN	Serveur utilisant un système de gestion de versions Subversion. C'est le serveur de stockage qui contiendra les fichiers synchronisés.
Dépôt	Répertoire parent d'un projet à synchroniser : il contient donc des sous dossiers et des fichiers. Il est créé par un super-administrateur, est administré par ses administrateurs désignés, et synchronisé avec plusieurs utilisateurs.
Client	Ensemble des applications à l'intention de l'utilisateur, destinées à interagir avec le serveur pour la synchronisation souhaitée des répertoires et fichiers. Il contient l'application cliente à proprement parler et une interface graphique de configuration.

Terme	Description
Serveur	Ensemble des applications à l'intention du super administrateur, destinées à interagir avec le serveur SVN et les clients pour permettre la synchronisation souhaitée des répertoires et fichiers. Il se compose d'une application serveur à proprement parler, d'une interface graphique de configuration, d'une interface web et d'une base de données.
Navigateur web	Application conçue pour consulter le *World Wide Web.*
Interface web	Portail web accessible par un navigateur web permettant à l'utilisateur, selon ses droits, d'ajouter/supprimer/modifier un administrateur/dépôt/client et de consulter l'historique des ajouts/suppressions/modifications de ses fichiers synchronisés.
Scripts	Langage de programmation interprété qui bénéficie d'une syntaxe de haut niveau.
Base de données	Support sur lequel seront stockés des informations utiles au bon fonctionnement du serveur tel que les identifiants et mots de passe des utilisateurs ou encore l'adresse du serveur SVN.
Super-administrateur	Utilisateur hébergeant le serveur. Il est l'utilisateur possédant le plus de droits (notamment l'ajout/suppression/modification de dépôts et d'administrateurs) et il est chargé de créer les dépôts et leurs administrateurs.
Administrateur	Utilisateur chargé d'administrer un dépôt (ajout/suppression/modification des droits d'un client). Son compte est créé par le super-administrateur.
Protocole de communication	Ce terme désigne les règles de communication entre l'application client et serveur.
Thread Slot Signal	

Rédacteur : Yann MAHE
Relecteur : Julien GUERY

Annexes

ANNEXE 1 - INFORMATIONS SUR LES DIFFÉRENTS MESSAGES

Les messages échangés entre clients et serveur respectent un formalisme XML, dont le nœud principal peut prendre différents noms :

- **FileInfo** : information concernant les médias, qui peut être de plusieurs types différents :
  - ✔ UPDATED : le contenu d'un fichier a été modifié. Le message contient alors le nouveau contenu, transformé en BASE64 pour que les données binaires puissent aussi transiter.
  - ✔ CREATED : création d'un nouveau média, à rajouter à la liste de synchronisation.
  - ✔ REMOVED : suppression d'un média
  - ✔ REVISION_FILE_INFO : le client envoie un numéro de révision et le nom du dépôt pour que le serveur lui envoie les modifications effectuées sur ce dépôt depuis la révision en question.
- **END_OLD_DETECTIONS :** message envoyé par le client à la connexion, quand il finit d'envoyer ses modifications détectées quand il n'était pas connecté, puis par le serveur quand il finit d'envoyer les mises à jour qui se sont produit quand le client n'était pas connecté.
- **IDENTIFICATION :** relatif à l'identification du client sur le serveur.
- **RESPONSE :** réponse par rapport à une requête (un message d'une des trois natures précédentes) reçue plus tôt. Là encore, une réponse peut être de différents types :
  - ✔ Des réponses à un **FileInfo** :
    - ACCEPT_FILE_INFO : tout s'est bien passé, on accepte la modification et on l'a bien pris en compte
    - REJECT_FILE_INFO_FOR_PARAMETERS : la requête était erronée, les paramètres avec lesquels elle a été envoyée étaient mauvais.
    - REJECT_FILE_INFO_FOR_RIGHTS : la requête n'a pas pu être accordée car les droits de l'utilisateur l'ayant envoyée ne sont pas suffisants, ou l'utilisateur n'est pas identifié sur le serveur. Il se peut aussi que le serveur soit en train d'envoyer des requêtes, alors les requêtes du client du type END_OLD_DETECTION et REVISION_FILE_INFO sont rejetées par un message de ce type.
    - REJECT_FILE_INFO_FOR_CONFLICT : un conflit SVN a été détecté suite à la requête.
    - REJECT_FILE_INFO_FOR_SVNERROR : une erreur SVN est apparue quand la requête a été exécutée.
  - ✔ Des réponses à une **IDENTIFICATION :**
    - ACCEPT_IDENTIFICATION : l'identification au serveur s'est effectuée sans problème.
    - REJECT_IDENTIFICATION_FOR_PSEUDO : identification rejetée car le pseudo n'existe pas.
    - REJECT_IDENTIFICATION_FOR_PASSWORD : identification rejetée car le mot de passe ne correspond pas au pseudo renseigné.
  - ✔ Des erreurs de connexion :
    - NOT_CONNECT : soit la connexion avec le serveur n'est pas établie, soit nous ne sommes pas identifiés.
    - NOT_PARAMETERS : identique à REJECT_FILE_INFO_FOR_PARAMETERS
    - NOT_TIMEOUT : délai d'envoi de la requête dépassé
    - NOT_SEND : l'envoie d'une requête a échoué, pour une autre raison.

Exemple de message envoyé :

```
<FileInfo revision="82" realPath="test1/fichier.txt" isDirectory="false" type="CREATED"/>
```

www.telecom-bretagne.eu

Campus de Brest
Technopôle Brest-Iroise
CS 83818
29238 Brest Cedex 3
France
Tél. : + 33 (0)2 29 00 11 11
Fax : + 33 (0)2 29 00 10 00

Campus de Rennes
2, rue de la Châtaigneraie
CS 17607
35576 Cesson Sévigné Cedex
France
Tél. : + 33 (0)2 99 12 70 00
Fax : + 33 (0)2 99 12 70 19

Campus de Toulouse
10, avenue Edouard Belin
BP 44004
31028 Toulouse Cedex 04
France
Tél. : +33 (0)5 61 33 83 65
Fax : +33 (0)5 61 33 83 75

TELECOM
Bretagne