Mole

Extension Firefox destinée à l'utilisation du réseau Molesys

Erik Clairiot & Rémi Melisson

June 2008

Table des matières

1 Introduction au projet Mole

1.1 Présentation fonctionnelle

Molesys est un système permettant à des utilisateurs connectés sur un même réseau (Internet ou local) de se partager divers contenus. Nous distinguerons dans l'ensemble de ce document le système global, *molesys*, et le logiciel utilisateur, *Mole*.

Regroupement des utilisateurs
A la manière d'*IRC*, les utilisateurs sont connectés à un ou plusieurs salons. A l'interieur d'un salon, les utilisateurs ont la possibilité d'interagir entre eux afin de communiquer et d'accéder aux divers contenus que chacun de ces utilisateurs partage.
L'intéret majeur du fonctionnement par salon est de permettre à différentes personnes de se regrouper par thématique. Et par la même, de faciliter la prise de contact avec d'autres personnes que le thème intéresse.

Partage de contenu
La principale fonctionnalité de *Molesys* est le partage de contenus entre utilisateurs.
Le premier intérêt est de faciliter le partage de son utilisation d'internet via un navigateur, avec d'autres personnes. En effet, par le biais d'internet, beaucoup de technologies de diffusion d'informations ont vu le jour.
Citons par exemple l'utilisation des flux *RSS* ou encore des podcast. Nous pouvons être amené à vouloir partager ce genre de sources avec d'autres personnes susceptibles d'être interessées.
De-même, il n'est pas rare de trouver un site, ou juste un lien (article, webradio, image...) qu'on voudrait faire découvrir à d'autres.
Ensuite, l'utilisateur de Mole a la possibilité de rendre public une page le concernant. Entendons par cela, un espace défini de manière totalement libre, dans lequel il pourrait par exemple rendre disponible son e-mail, ou simplement une image de son choix, ou, tout aussi simplement, un contenu *HTML* ; les utilisateurs pouvant transférer entre eux des flux *XML*, beaucoup de choses sont imaginables.
Finalement, il pourra être possible de partager des fichiers et ainsi permettre aux autres de les télécharger (bien que nous imaginons surtout le transfert de fichiers de taille raisonnable, n'utilisant pas de technologies *P2P*).

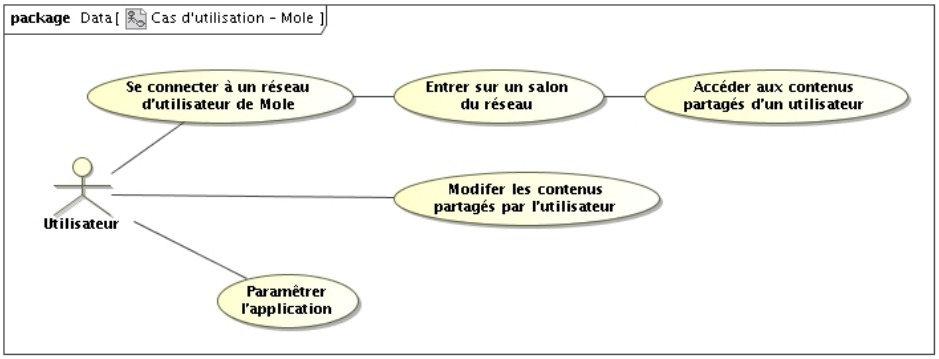


Diagramme des cas d'utilisation de *Mole*

1.2 Architecture générale de la solution

L'architecture globale de la solution repose sur plusieurs points :

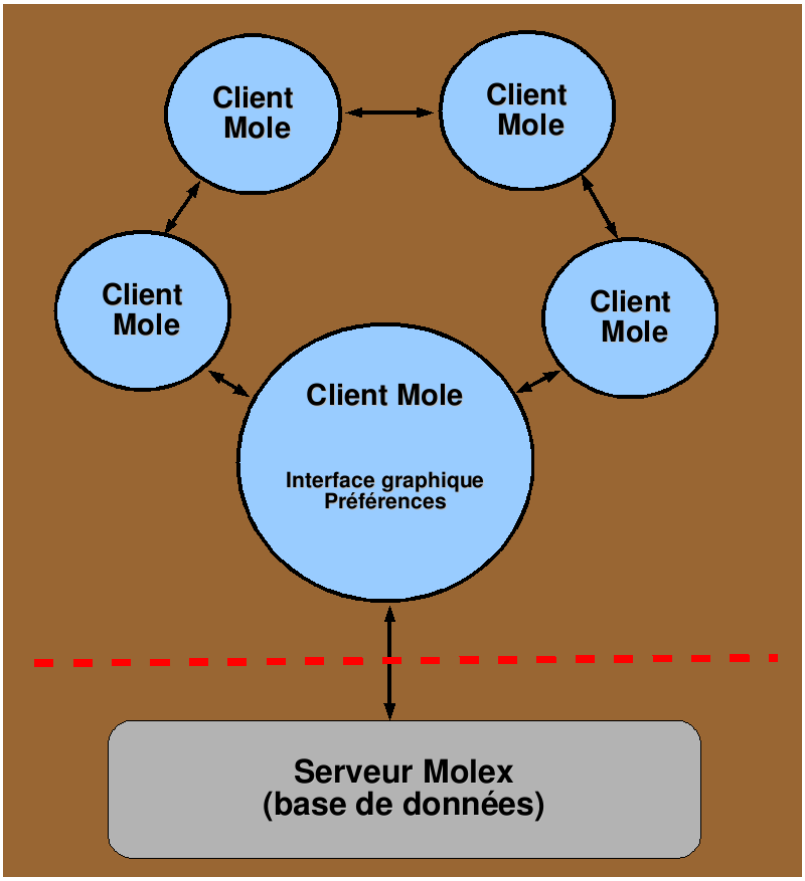
Mole
Mole désigne le programme sur le poste de l'utilisateur, que l'on nommera client par la suite.
L'utilisateur doit disposer d'une interface graphique simple reposant sur plusieurs axes :

- Une navigation facile entre les différents salons et les personnes avec lesquelles l'utilisateur communique.
- Un affichage protéiforme et modulable en fonction des contenus partagés par un utilisateur. Cette interface doit pouvoir restituer de manière efficace les fonctionnalités habituelles de la navigation sur internet.
- Une gestion des paramètres de l'application et des préférences de l'utilisateur.

Mole fait appel aux communications réseaux, c'est pourquoi l'utilisateur devra paramètrer son poste afin d'autoriser les communications entre l'application et l'extérieur par le biais de plusieurs ports (configuration du firewall).

Le serveur
Pour mettre en relation les utilisateurs, ceux-ci devront tout d'abord se connecter à un serveur, puis à un des salons de ce serveur.
C'est dans la base de donnée associée au serveur que seront centralisées les informations nécessaires à la connection entre les utilisateurs par le biais des salons.

Réseau d'utilisateurs
Les utilisateurs pourront avoir besoin de communiquer avec l'ensemble des participants d'un salon. Par exemple, pour l'utilisation d'un chat sur le salon, pour une information susceptible de concerner tout le monde ou encore pour avertir les autres de la déconnection de quelqu'un. Plutot que de solliciter à chaque fois le serveur pour ce genre de requêtes, il est plus interessant de laisser le soin aux machines utilisateurs de faire vivre le salon.
Pour cela, il faut choisir une architecture de liaison entre les clients permettant la propagation d'un message à travers le salon, avec la possibilité de contacter, entre autre, le serveur. Plus les clients seront autonomes, plus le travail du serveur sera allégé.



Architecture générale de l'application

1.3 Méthodologie de travail

Pour pouvoir mener à bien ce projet, notre travail aura profité des ces quelques points :

Documentation
Notre travail de documentation peut-être décomposé en deux parties.

Tout d'abord il nous fallait rechercher les différentes technologies permettant la construction d'une telle architecture. Pour ce faire, s'appuyant sur les connaissances acquises en cours, nous avons parcouru internet entre forums et *Wikipedia* de façon à approfondir les idées que nous avions. Ceci afin de faire des choix judicieux, en accord avec l'architecture globale présenté ci-dessus. Au fils des liens, nous avons finalement trouvé chaussure à notre pied pour chacun des axes de développement.

Dans un second temps, l'implémentation des différentes couches nécessitait la maîtrise de technologies que nous n'avions jamais cotoyé. C'est pourquoi nous nous sommes reposé cette fois sur de la documentation plus technique afin de parvenir à manier ces nouveautés. Cette fois, nous avons pu clairement bénéficier d'un des pilliers de l'informatique libre : la communauté. Entre forums, *IRC*, sites de documentation (*MDC*, Mozilla Developpement Center), nous trouvions reponses à nos questions et guides pour débuter.

Finalement, cette documentation aura été une phase majeure de notre travail sur ce projet.

Développement modulaire
Nous avons tâché de travailler sur ce projet de façon modulaire afin de pouvoir séparer clairement les différentes couches. Cela pourrait nous permettre, à terme, d'utiliser *Molesys* avec un serveur différent, ou avec un interface en ligne de commande...
De plus cette façon de faire nous aura permis de se répartir le travail plus facilement. Ainsi nos tâches étaient rigoureusement séparées et, lors de la mise en commum, il nous suffisait de "recoller les morceaux".

Sourceforge, Subversion
Lors de la construction d'un tel projet, il est très utile d'utiliser un gestionnaire de version. Même si nous travaillions à des niveaux différents, regarder le travail de l'autre, récupérer facilement les sources ou avoir accés aux anciennes versions sont autant de fonctionnalités qui simplifient vraiment le travail.
Nous avons voulu construire une application dans l'esprit du logiciel libre et c'est pourquoi, dès le début du projet, nous l'avons enregistré sur *Sourceforge*, afin de bénéficier, entre autre, du serveur *Subversion*.

1.4 Récupérer Mole (client et serveur)

Mole est un logiciel libre, sous licence *GNU GPL*. Il est donc librement fourni, sources et éxécutables.

Un site Internet est en ligne concernant *Molesys*, accessible à l'adresse *http ://molesys.sourceforge.net*. Un accés en consultation au dépôt *SVN* est possible à partir de ce site.

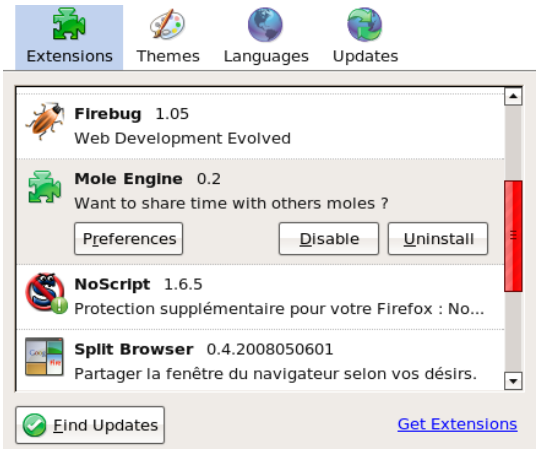
Vous pourrez y trouver l'ensemble des sources du projet, du serveur au client.
L'extension *Firefox* finie est aussi téléchargeable et installable directement à partir du site, mais uniquement pour les utilisateurs de Linux.

1.5 Manuel d'utilisation

Ceci n'est qu'un résumé du manuel d'utilisation que vous pourrez trouver à l'adresse *http ://molesys.sourceforge.net/*.

Prérequis à l'utilisation de Mole
A l'heure actuelle, Mole ne fonctionne que sous un système Linux, mais à terme, sera compatible Windows.
Mole utilisant le moteur de rendu *Gecko*, il est nécessaire d'avoir installé une application l'intégrant : *Firefox* est l'application la plus populaire l'intégrant, *XULRunner* en est une autre.
Le transfert d'information entre utilisateurs de *Mole* utilise des échanges réseaux via deux ports TCP : le port 80 pour communiquer avec le serveur, et le port 6242 pour les clients (par défaut).
Le port 80 étant souvent ouvert par défaut, veillez à bien ouvrir le port 6242 aux entrées externes.
Finalement, il peut être utile de connaître l'adresse d'un serveur dans lequel trouver des salons correspondant à vos envies. Par défaut, l'application utilise le serveur que nous avons mis en place.

Installation
L'installation se fait par l'intermédiaire du gestionnaire d'extensions *Firefox*.
Vous pouvez dès lors modifier vos préférences dans la gestion des extensions : *outils/modules complémentaires*.
Après le redémarrage de Firefox, vous pouvez accéder à *Mole* de la sorte : *outils/Mole*.



Configuration de l'application

Connection au serveur
Pour se connecter au serveur, il faut préciser un nom d'utilisateur. A la connection, si le nom n'est pas déjà utilisé, la connection devrait être effective. Consultez la fenêtre de logs pour avoir état de la connection.

Connection à un salon
Pour vous connecter à un salon existant, vous n'avez qu'à double-cliquer sur le nom de ce dernier pour être connecté et avoir accés à la liste des autres membres du salon. Il est aussi possible de le rejoindre en entrant directement dans le champ *Join channel* le nom du salon à rejoindre.
Notez que pour créer un salon, il suffit de joindre un salon qui n'existe pas.

Communication avec un utilisateur
Il vous suffit de double-cliquer sur le nom d'un autre membre à l'intérieur de la liste des utilisateurs du salon pour voir apparaître celui-ci dans un nouvel onglet.
Vous avez dès lors accés à la page personnel de cet utilisateur.

2 Technologies approchées

Il était nécessaire, avant même la phase de conception, d'analyser les technologies existantes qui pourraient nous être utiles pour notre application. Cette phase aura permis de comparer des technologies entre elles, de manière à utiliser celles qui nous conviendraient le mieux pour le développement de l'application. Au dela de nous apporter une solution, ces comparaisons nous auront surtout permis de comprendre au mieux le fonctionnement de ces technologies.

2.1 XML-RPC, XML Remote Procedure Call

XML-RPC est un protocole de communication à la spécification simple, qui permet d'appeler des méthodes à distance.
Le protocole utilisée pour le transport des données est le protocole *HTTP*, l'encodage des données est lui fait en respectant la norme *XML*.

Sa spécification simple ne le bride pas pour autant quant à son utilisation : il est possible d'utiliser des données aussi simples que complexes en autorisant la manipulation des types primitifs, des tableaux, des structures... Il est aussi possible d'envoyer du contenu binaire encodé en *base64*.

Exemple Exemple d'appel de méthode en XML-RPC

```
<?xml version="1.0"?>
<methodCall>
    <methodName>mole.connection</methodName>
    <params>
 <param>
     <value><string>nick</string></value>
     <value><string>ip</string></value>
     <value><int>6242</int></value>
 </param>
    </params>
</methodCall>
```

Exemple Exemple de réponse d'appel XML-RPC

```
<?xml version="1.0"?>
<methodResponse>
    <params>
 <param>
     <value>Ok</value>
 </param>
    </params>
</methodResponse>
```

Il est l'ancêtre du *SOAP*, qui conserve l'encodage en *XML* et le principe d'enveloppe. Cependant, *SOAP* est bien plus complexe que *XML-RPC*, et *XML-RPC* convient parfaitement pour les services que l'on implémentera.

2.2 XPFE, plateforme de développement Mozilla

XPFE, pour *Cross Platform Front End*, désigne l'ensemble des technologies utilisées par *Mozilla*. On peut le comparer à un framework de développement, puisqu'il met à disposition du développeur tout ce qui est nécessaire pour développer des applications de même type que celles de *Mozilla*.

Des technologies que la plateforme supporte, on détaillera plus en précision celles que nous utiliserons plus profondément. Parmi les plus importantes, nous citerons le moteur de rendu *Gecko*, l'abstraction des couches réseaux avec *Necko*, *XUL*, le descripteur de ressources *RDF*, *DOM*, *Javascript*, *CSS*, *XPCOM*...

Une application développée en utilisant ce framework n'est exécutable que si l'utilisateur dispose d'un logiciel de la suite *Mozilla* installé. Il est cependant possible de s'affranchir de cette condition en utilisant une plateforme

appelée *XULRunner*, qui est un environnement d'éxécution pour les applications basées sur ce framework. Cette application contient le moteur de rendu *Gecko* ainsi que nombre des *API* utilisées par *Mozilla*.

2.2.1 XUL, XML-based User Interface Language

XUL est un langage qui utilise le méta-langage *XML* pour décrire une interface graphique. C'est une technologie qui fait partie de *XPFE* : l'ensemble des applications basées sur ce framework sont rendues visuellement selon des interfaces définies en *XUL* : *Firefox*, *Thunderbird*, *Komodo*...
Ce language permet d'écrire rapidement l'interface d'une application riche accessible localement ou proposée en tant que service par *HTTP*.

XUL seul ne permet pas de mettre de la dynamique à l'interface. Il faut pour cela utiliser un langage de script, qui puisse manipuler le contenu du document. Puisque le *XUL* est basé sur le *XML*, le *Javascript* correspond parfaitement pour cette utilisation, grâce aux fonctions *DOM* qu'il propose. Il est donc possible de modifier le contenu du document de la même manière que l'on ferait pour modifier le contenu d'un fichier *HTML*.

L'on peut modifier la structure du document *XUL* dynamiquement, ce qui est pratique mais peu utile si l'application n'a pas de fonction. La puissance de *XUL* se revèle lors de l'utilisation de composants *XPCOM*, qui fonctionnent similairement aux web-services et qui seront détaillés par la suite. *XUL* ne manipule pas directement ces composants, mais passe par l'intermédiaire du *Javascript* pour commander le composant au gré des actions utilisateurs.

Un document *XUL* est couramment divisé en trois parties :

1. **content** Définit l'agencement de l'interface graphique en utilisant les balises *XUL* à disposition.
2. **skin** Détaille le rendu et l'apparence que l'interface graphique aura, en utilisant des feuilles de style *CSS* et des images.
3. **locale** *XUL* permet très facilement de gérer les problèmes de distribution d'une application multilingues, en utilisant un fichier d'entités *DTD* par langue qui sera automatiquement chargé selon la localisation de l'utilisateur. Ainsi, ajouter le support d'une langue consiste uniquement, sans aucune autre modification, à offrir un fichier d'entités correctement rempli.

Les documents *XUL* ne sont consultables qu'en utilisant une application embarquant le moteur de rendu *Gecko* : *Mozilla Firefox* ou *XulRunner* par exemple.

Les applications développées en *XUL* peuvent donc utiliser des composants *XPCOM*. Quand est-il de la sécurité offerte lors de l'utilisation d'une application *XUL* en local ou par Internet ? En effet, des composants enregistrés ont des permissions pouvant aller jusqu'à l'accès au système de fichiers de l'utilisateur, ou à l'ouverture de connections réseaux. Il est rapide de voir qu'il est risqué pour les utilisateurs d'accéder à des applications disponibles sur Internet, qui pourraient utiliser ces composants à des fins malveillantes.

Une réponse apportée à ce problème par *XPFE* est la distribution de permissions graduées aux applications, selon qu'elles sont accessibles localement sur l'ordinateur de l'utilisateur ou sur un ordinateur distant.
Pour qu'une application bénéficie des permissions maximales, il est nécessaire de l'enregistrer dans le *chrome*, qui est un espace de sécurité agencé à la manière d'un système de fichiers. Ces applications deviennent accessibles via l'adresse de type *chrome ://nom*.

2.2.2 XPCOM, Cross Platform Component Object Model

XPCOM est une technologie developpée par *Mozilla*, largement utilisée par *XPFE*, et qui s'inspire des technologies *CORBA* et *.COM*.

Le but de la distribution d'un composant en objet *XPCOM* au sein du *XPFE* est d'offrir la possibilité à chaque application basée sur *XPFE* d'exploiter les ressources et services proposés par le composant. Ces services sont définis par une interface *XPIDL*, qui est une implémentation basée sur *IDL*. Cette interface permet à tout développeur de connaître l'étendue des services proposés par le composant.

La plateforme *XPFE* dispose donc d'une grande bibliothèque de composants (plus de mille) enregistrés en son sein, en plus des composants qui peuvent provenir d'applications basées sur le framework *Mozilla*. Parmi ces composants fournis, on pourrait par exemple distinguer le gestionnaire de marque-pages, le gestionnaire de téléchargements, la gestion d'un historique, un moteur de navigation.. Autant de composants que n'importe quel développeur peut charger dans son application et exploiter.

Parmi les avantages, particularités de *XPCOM*, l'on pourrait donc citer :

- Modularité du code
  Un composant est exploitable par n'importe quelle application basée sur *XPFE*. Plutôt que de développer au sein de chaque application du code ayant une même fonction, un unique composant est idéalement la solution pour en faire profiter les autres applications.
- Architecture dynamique
  Les composants sont chargés lors de leurs éxécutions. Il est donc possible d'ajouter dynamiquement un composant *XPCOM* à la librairie, et de l'exploiter directement, sans devoir relancer l'application.
- Multi-langages L'écriture d'un composant peut être réalisée en divers langages, bien que le langage le plus usité soit le *C++*. Actuellement, les langages supportés sont le *Python*, *Ruby*, *Perl*, *Javascript*... Un travail est actuellement en cours pour ajouter le support du *Java*.

2.2.3 Extensions Mozilla Firefox

La mise en place d'extensions *Firefox* est une des idées des développeurs *Mozilla* qui a eu le plus d'impact quant au fonctionnement et au développement d'un de leurs logiciels. Comment faire pour que d'autres développeurs que ceux de *Mozilla* puissent aisément ajouter des fonctionnalités à leur logiciel favori ? Comment faire pour les motiver à developper des modules que nous, encore développeurs de *Mozilla*, n'avons pas le temps de développer ?
Simplement, ils ont, dans la logique de fonctionnement du *XPFE*, implémentés un système d'utilisation de modules externes au sein de *Firefox*. Cela se traduit par un gestionnaire d'extensions, ou *add-ons*.

Une extension *Firefox* est distribué en tant qu'archive **.xpi* (prononcer *zippi*), via le site des développeurs de l'extension, ou par un site généraliste d'extensions [1]. Ces extensions, une fois téléchargées, se décompressent dans le profil *Firefox* utilisé à ce moment là. Ce sont ces fichiers décompressés qui définiront le mode de fonctionnement de l'extension, ainsi que l'intégration de celle-ci au sein de *Firefox*.

Il n'y a pas de limite à ce que les extensions peuvent proposer comme fonctionnalités : client *FTP*, client *IRC*, client pour réseau *Molesys*, gestionnaire avancé de téléchargements, contrôle du navigateur à la manière de *Vi*, bloqueur de publicités...
La seule chose requise pour distribuer une extension étant de respecter l'architecture du contenu de l'archive **.xpi*, de proposer des fichiers d'interface graphique fonctionnels et à l'intégration cohérente dans *Firefox*.

Une grande partie des extensions peuvent être développés uniquement en *Javascript*, car *XPFE* propose d'office beaucoup de librairies *XPCOM* (utilisables en *Javascript*). Lorsque *XPFE* ne couvre pas les besoins du développeur par une librairie, il est nécessaire de développer soi-même un composant *XPCOM*, de le fournir dans le **.xpi* et de l'exploiter correctement en *Javascript*.

L'intégration à *Firefox* se fait en utilisant le langage d'interface utilisateur *XUL*, et en précisant dans un fichier les directives d'intégrations (ajout d'un bouton au menu contextuel, ajout d'une icone à un endroit de la barre de titres...). Il est possible avec une extension de redéfinir tout l'ensemble de l'interface du navigateur, afin de proposer une utilisation fonctionnelle optimale pour l'utilisateur.

L'accès à une extension se fait, de manière généraliste, par le gestionnaire d'extensions. Il est possible d'éxécuter à partir de là chacune des extensions enregistrées sur le profil courant de *Firefox*, mais aussi de les paramètrer ou supprimer.

Un aspect important des extensions est de comprendre qu'elles sont, comme toutes applications *Mozilla*, enregistrées distinctement dans le *chrome*, et que dés lors, l'on peut y accéder à partir de là : une extension qui s'appelerait *Mole* serait disponible à l'adresse *chrome ://mole/content/* (*content* pour l'accès à l'interface utilisateur de l'extension).
L'enregistrement de l'extension dans le *chrome* est signe de confiance envers l'extension, et donne d'office un maximum de permissions à l'extension. Aussi vous veillerez donc à ne pas installer n'importe quel extension trouvée sur Internet, sans avoir consulté au préalable les opinions que des utilisateurs ont sur elles. Cela confère encore une grande force aux extensions libres, qui de par l'ouverture de leur code source, permettent de vérifier l'intégrité des volontés du développeur.
Il est à noter qu'il est possible d'authentifier auprès de *Mozilla* son extension, mais cela n'est apparement pas très aisé d'être autorisé, et le développement d'extensions étant un travail voué à la modification, au gré des évolutions de *Firefox* à travers le temps, pour rendre de nouveau compatible son extension pour la version la

[1] http ://addons.mozilla.org, http ://www.geckozone.org

plus récente de *Firefox*, s'attacher à disposer d'une authentification constante est un tantinet contre-productif.

2.3 PHP / MySQL

Un des langages le plus utilisé pour éxécuter des scripts à distance s'avère être le langage *PHP*, qui en est actuellement à la version 5. Il est très facilement utilisable sur des serveurs, et est notamment fourni par une très grande majorité de serveurs Internet. Le déploiement d'un serveur *HTTP* avec le service *PHP* reste une chose relativement aisée.

L'arrivée de la programmation orientée objet dans ce langage, la très grande bibliothèque de scripts déjà existants et de librairies / frameworks disponibles en font un langage qui peut rapidement combler les attentes de développement, si par exemple une librairie *XML-RPC* existe déjà.

Le *PHP* est un langage couramment utilisé en conjonction de *MySQL*, lorsque le besoin d'une base de données se prononce. Il est vrai qu'il est extrêmement aisé pour le développeur connaissant *SQL* de comprendre l'utilisation de ces bases au sein d'un script *PHP*.

SQL est un système de base de données solide et efficace, pour une utilisation courante. Il est vrai que les performances ne peuvent être comparable à un système tel qu'*Oracle*, mais pour une utilisation que nous qualifierons de "grand public", et sans avoir d'extrêmes exigences, le *SQL* est très adéquat. De plus, tout comme *PHP*, les bases de données *SQL* fleurissent chez les hébergeurs depuis nombres années.

2.4 Réseau d'utilisateurs

Afin de limiter la charge du serveur, nous avons fait le choix de connecter les utilisateurs entre eux pour toutes les communications concernant un salon.
Ainsi, lorsqu'un utilisateur se connecte à un salon, il est intégré au réseau. Cette intégration lui permettra alors de transmettre un message à l'ensemble du salon sans pour autant avoir à solliciter le serveur.

Nature des messages

- Messages de gestion du salon
  Lorsqu'un utilisateur arrive ou quitte le salon, il doit prévenir tout le monde. De même, lorsqu'un utilisateur se rend compte de la déconnection de quelqu'un, il propage l'information à tout le monde (en passant éventuellement par le serveur pour le mettre à jour).
- Autres
  Ce réseau pourra servir à la transmission de messages d'un autre type, c'est par exemple par cette propagation que seront transmis les messages du chat. On pourrait imaginer aussi un système de notification d'événements permettant à quelqu'un de toucher le salon entier.

Problématique de la propagation d'un message
Le choix d'une architecture de ce réseau d'utilisateurs revient à traiter la problématique de propagation d'un message à travers un réseau de machines.
Les architectures doivent être regardées selon certaines caractéristiques :

- Résistance aux pannes
  Le réseau doit être capable de subir plusieurs déconnections simulannées et conserver une structure fonctionnelle.
- Vitesse de propagation
  Pour une plus grande réactivité, nous chercherons à minimiser le temps écoulé entre l'émission de l'information par l'utilisateur et la reception par tous les utilisateurs du réseau.
- Charge du traitement
  Selon l'architecture choisie, le traitement de la propagation d'un message peut être plus ou moins lourd. En effet, si l'utilisateur ne doit envoyer qu'un message, cela sera bien moins long que s'il doit le transmettre à 15 personnes différentes.
- Intégration et désintégration d'un utilisateur
  Lors de l'arrivée d'un nouvel utilisateur sur le salon, celui-ci doit être intégré au réseau et, à son départ, le réseau doit se reconstituer sans lui.

Exemples d'architectures implémentables

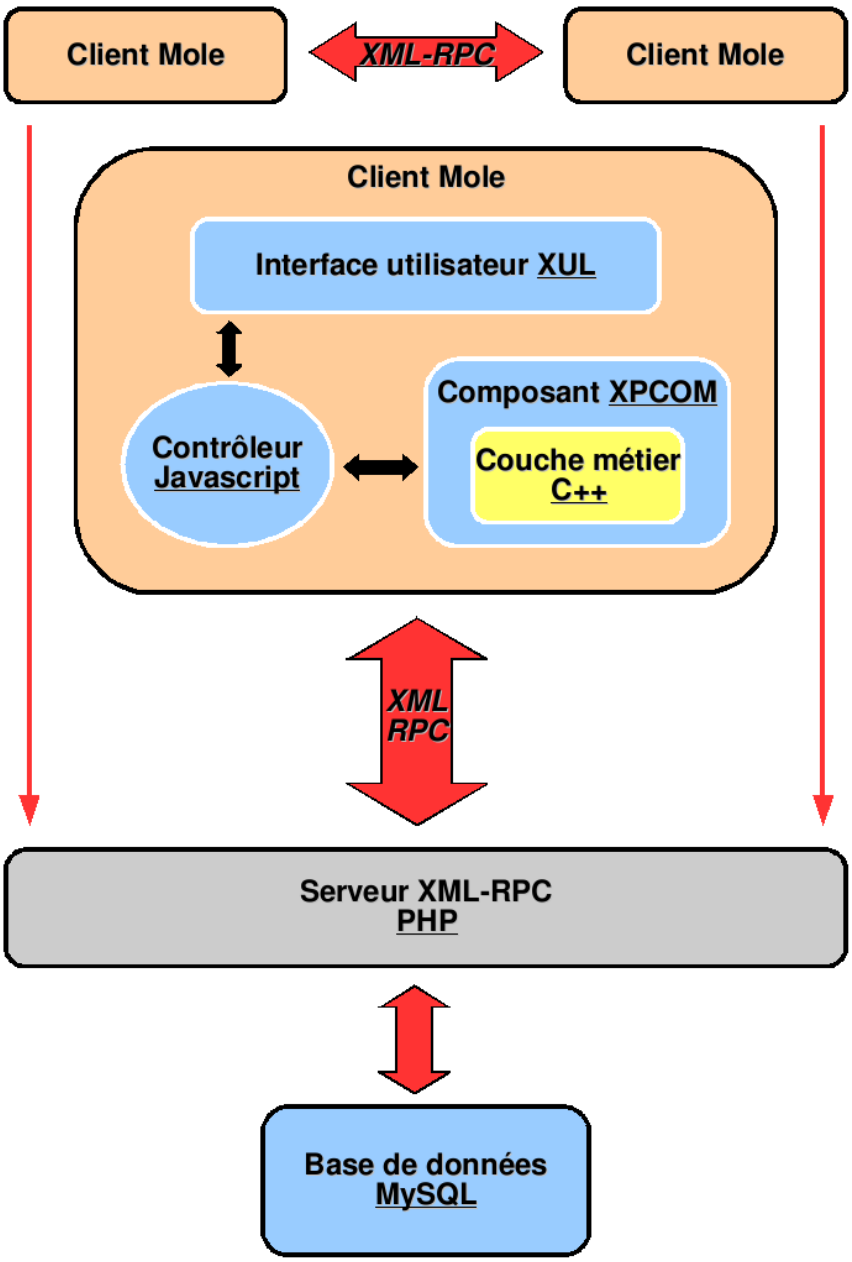
- Anneau
  Dans une architecture en anneau, chaque utilisateur est connecté à un voisin de gauche et de droite. S'il reçoit une information à droite, il la transmet à gauche et l'information se propage de la sorte jusqu'à ce que l'initiateur de cette transmission recoive son message et stoppe la propagation.
  En rapport avec les caractéristiques citées ci-dessus l'anneau n'est pas une très bonne solution. Le temps de propagation est maximisé, et s'il y a une seule panne, le réseau sera défaillant. Même si la charge de traitement est très faible et que l'intégration d'un utilisateur est aisée, cette architecture n'est pas viable. Par contre il suffit de transformer l'anneau pour envoyer à gauche et à droite le message (avec arret lorsque le message a déjà été recu) pour modifier ses caractéristiques (détection d'une panne, accélération de la transmission).

- Pair-à-pair
  Les réseaux pair à pair sont autant d'exemples de la gestion d'un réseau décentralisé, et nous aurions pu nous inspirer des différentes architectures pour les adapter à nos besoins (Kademlia, Freenet, Bittorent..).

- Bibliothèques annexes
  Il existe des solutions pour gérer cet aspect de l'architecture. Elles permettent d'abstraire le travail de transmission en utilisant des technologies développées dans ce sens. Citons par exemple le projet JXTA (de Sun), dont il existe une bibliothéque C++ que nous aurions pu éventuellement utiliser.

3 Solution applicative

D'après l'architecture générale de l'application, nous avons modelisé de manière plus précise l'ensemble de nos besoins, et avons abouti sur un modèle à cinq couches, allant du client au serveur, que nous détaillerons distinctement. Le schéma suivant distingue les couches de l'application.



3.1 Spécification des communications

Nous avons décidé d'utiliser le protocole *XML-RPC* pour communiquer au travers du réseau *Molesys*, pour sa simplicité de compréhension et l'efficacité qui en résulte.

Les communications dans le réseau *Molesys* sont notamment de deux types :

1. De l'utilisateur au serveur
2. De l'utilisateur A à l'utilisateur B

Le serveur doit donc fournir des services *XML-RPC*, qui doivent permettre à un client de profiter de l'ensemble des actions proposées par le réseau *Molesys*. Les services seront détaillées plus en profondeur par la suite, dans l'explication de la couche serveur.

L'utilisateur A doit quant à lui pouvoir communiquer avec un utilisateur B directement, donc doit aussi proposer des services *XML-RPC*. Tout client *Mole* est donc aussi un serveur *XML-RPC*, en plus d'un client *XML-RPC*.

3.2 Couche serveur

La couche serveur régissant le fonctionnement de nécessite de pouvoir communiquer avec les clients au moyen de XML-RPC.
Elle doit être aussi capable de mémoriser une liste d'utilisateurs utilisant à chaque instant le système, ainsi que l'ensemble des salons existants sur molesys et les utilisateurs qui y sont présents.

De nombreuses librairies permettant de communiquer avec XML-RPC sont réalisées en PHP.
Le MySQL convenant parfaitement pour un stockage de données de ce type, se mariant très bien avec PHP, et étant un couple applicatif facile à mettre en place et largement en service sur les serveurs mutualisés, nous avons opté pour ce choix technologique.

Toutefois, à cause du mode non-connecté du protocole qui est le protocole utilisé par , il n'est pas possible de conserver une connection entre le client et le serveur, ce qui pose un problème d'identification d'origine des requêtes. Hors, il est essentiel de pouvoir définir si le client est déjà connecté sur le serveur, s'il est ou non sur un salon afin de savoir si l'on a le droit de lui envoyer la liste des utilisateurs qui y sont connectés..
Afin que le serveur puisse identifier le client à chaque appel d'un service, un identifiant de session a été créé, que nous avons lui-même identifié par le nom de *molid*.

Une chose nécessaire pour l'intégrité du réseau est de détecter les déconnections d'utilisateurs, afin de ne pas afficher d'utilisateurs fantômes sur les salons, et de ne pas saturer la base de données démodées.
De nombreuses politiques peuvent être imaginées ici pour palier à ce problème, allant d'une implémentation simple à de bien plus complexes.
Nous avons décidé d'utiliser un simple ticket de session que le client se verra affecter pour une durée limitée. Ce sera à la charge du client d'actualiser son temps de session, en reprenant contact avec le serveur ; le serveur aura lui pour tâche d'effectuer à des moments optimaux un nettoyage des données.

3.2.1 Serveur XML-RPC en PHP

La librairie utilisée en *PHP* est la librairie *phpxmlrpc*[2], en version 2.2 .

Le point d'accés au réseau molesys est le script *public/server.php*, qui instancie un serveur *XML-RPC* et publie cinq services.

- *string hello(name)*
  Service destiné uniquement à des fins de tests. Retourne une phrase contenant le nom envoyé.

- *molid connection(login, ip, port)*
  Ajoute un utilisateur au réseau *molesys*.
  Enregistre l'adresse IP du client et son port de transfert.
  Retourne un *molid* valide au client.

- *boolean joinChannel(name, molid)*
  Connecte un utilisateur à un salon. *molesys*. Renvoie l'état de la connection au salon.

- *channels[] listChannels(molid)*
  Recupére la liste de tous les salons enregistrés et l'envoie au client..

- *users[] listChannelUsers(name, molid)*
  Récupére une liste de tous les utilisateurs présents sur le salon précisé en paramètre.

Typiquement, le fonctionnement d'un serveur *XML-RPC* en *PHP* se fait en instanciant le serveur dans un script, enregistrer des méthodes *PHP* en tant que services, en précisant le nom du service, les paramètres d'entrées et les paramètres de réponse, et de lancer ce serveur *XML-RPC*. Dans notre conception, chaque service est une classe derivée de *Service*, et implémente la méthode *execute()*. Le déroulement de cette méthode est généralement décomposable en trois phases : décodage des paramètres d'entrée, réalisation du service, encodage des paramètres de sortie.

[2]Disponible à l'adresse *http ://phpxmlrpc.sourceforge.net*, en licence GPL

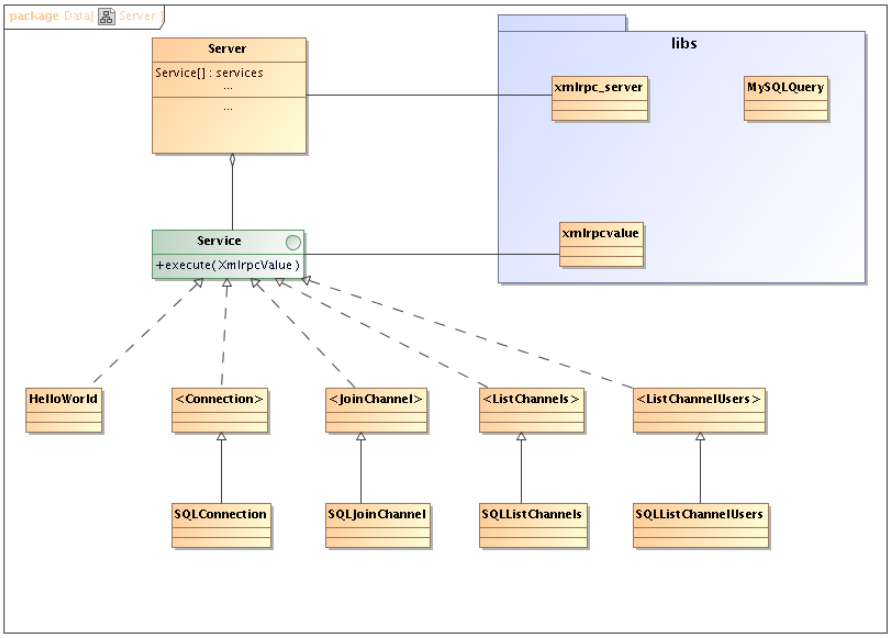


Diagramme des classes de la couche serveur

Le diagramme de classes montrent un héritage entre chaque service, défini par une classe abstraite, et une classe typée SQL le réalisant. Nous avons détaché la base de données de la réalisation du service, afin de garder une certaine indépendance lorsque l'on voudrait changer de type de base de données.

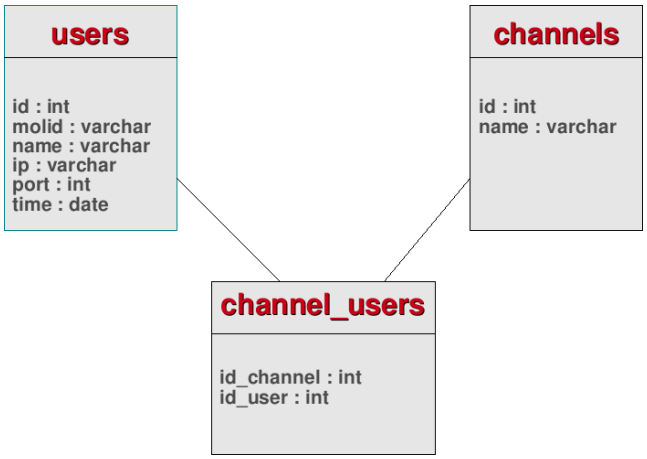
3.2.2 Base de données en MySQL

Une base de données MySQL est utilisée pour mémoriser le statut du réseau *molesys*. Trois tables permettent de maintenir le réseau cohérent.

- *users*
  Utilisateurs courants du réseau.

- *channels*
  Salons existants sur le réseau.

- *channel-users*
  Association entre les salons et les utilisateurs qui y sont présents.



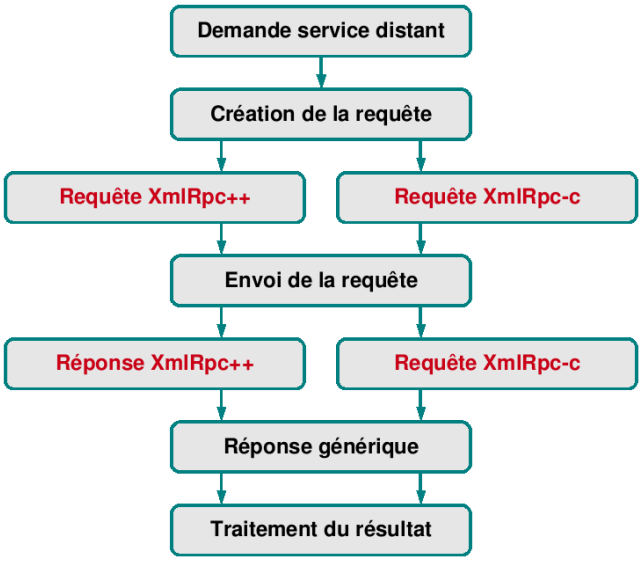
Représentation des tables du réseau *molesys*

3.3 Couche client

3.3.1 Couche métier en C++

Notre application devant être portable, il est essentiel que chaque librairie utilisée le soit, principalement pour les librairies XML-RPC, qui présentent comme trait commun un faible attrait pour la portabilité.
De plus, ne voulant pas dépendre trop fortement d'une librairie plutôt qu'une autre, un travail a été fait afin de pouvoir facilement changer de librairie sans pour autant modifier l'ensemble de l'architecture.

Nous avons donc décidé de décomposer l'appel d'un service distant par *XML-RPC*, afin de distinguer ce qui dépend des librairies et ce qui peut être générique.
Le diagramme suivant montre cette décomposition en étapes, dans le cas ou nous utiliserions plusieurs librairies (*xmlrpc++* et *xmlrpc-c*).



Décomposition de l'appel d'un service distant

Lors d'une requête vers un service *XML-RPC*, seuls l'envoi de la requête et la récupération de la réponse dépendent de la librairie ; la préparation du service étant toujours le même. ainsi que le traitement en fonction de la réponse.
Si l'on souhaite donc utiliser plusieurs librairies différentes, il suffit donc de définir pour chacune le traitement de la requête, de l'envoi à la récupération de la réponse, le reste étant générique pour chacun des services.

Tout comme pour les requêtes, le client *XML-RPC* est fourni par la librairie, il est donc nécessaire pour chacune des librairies de créer une classe dérivant de *Client*, afin d'abstraire le code provenant de librairie ; de même pour le serveur.

Package mole :ctx L'espace *ctx* concerne tout ce qui touche aux connections, et à leurs réalisations. Ainsi, on y trouve les classes clients et serveurs, les requêtes et celles implémentées pour les librairies, et une *Factory* pour faciliter la création de ces requêtes.

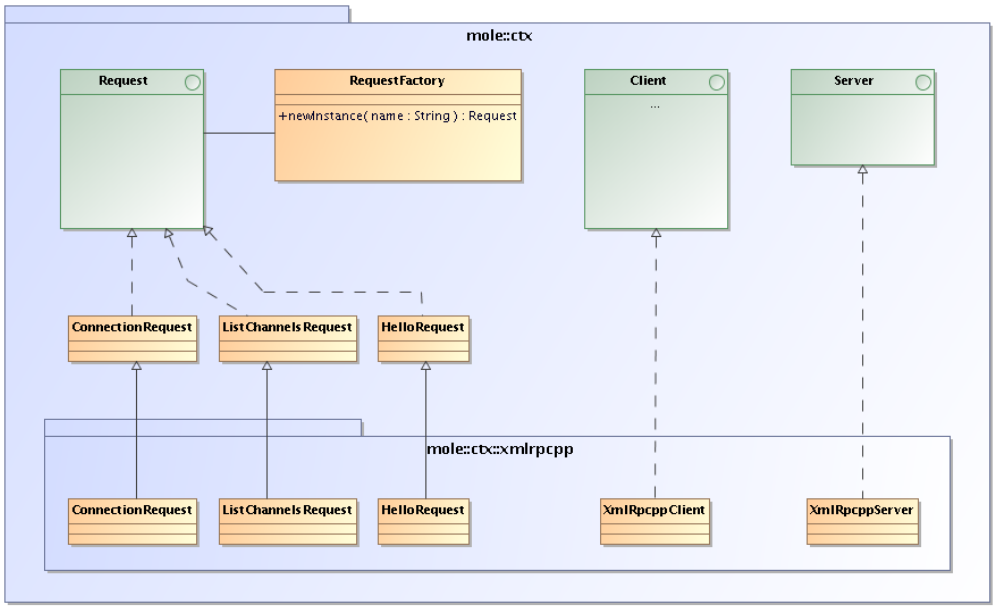


Diagramme de classes des communications

Package mole :serv Cet espace *serv* définit majoritairement l'ensemble des composants d'un service. Chaque service est composé d'une requête et d'une réponse. Pour aider à l'instanciation des requêtes et services selon la librairie, une classe *ServiceManager* permet de relier chaque service à chaque librairie avant utilisation de l'éxécutable.

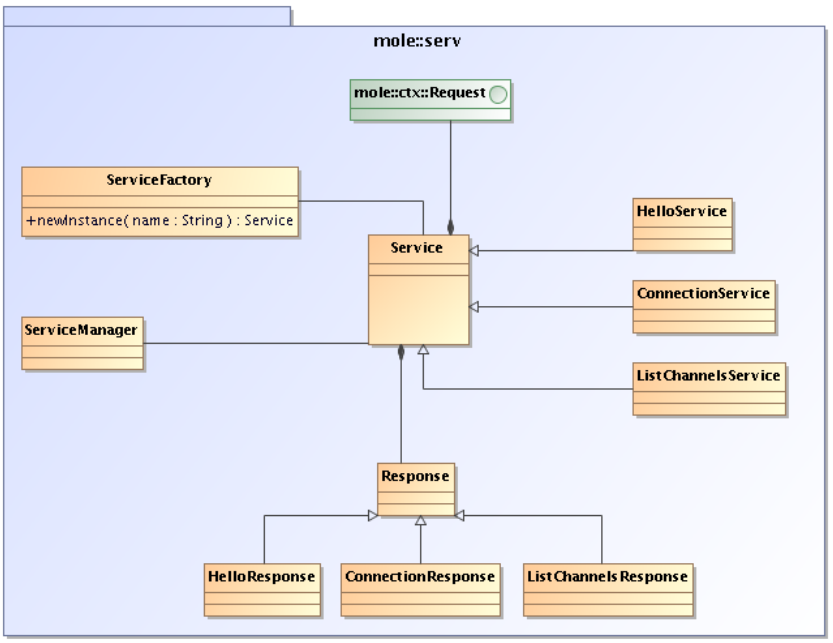


Diagramme de classes des services

La couche métier est validée par plusieurs éxécutables de tests, qui permettent de constater le fonctionnement de chaque appel de méthode *XML-RPC*, dans un cadre similaire à celui de l'application. Ce sont les fichiers *Test** disponibles dans le répertoire de compilation.

Pour les communcations entre clients, un serveur *XML-RPC* est démarré chez chacun après avoir enregistré les méthodes que le client propose. Il est nécessaire de séparer l'éxécution de ce serveur du thread courant. Ce que pour l'instant nous ne sachons faire au regard de nos connaissances C++, donc pour l'instant, le lancement du serveur se fait de manière totalement indépendante à l'éxécution de l'application.

3.3.2 Distribution du code métier en composant XPCOM

Pour utiliser du code compilé en C++ avec une application Mozilla, par le biais de Javascript, il est nécessaire de distribuer ce code en tant que composant XPCOM, et de l'enregistrer dans le gestionnaire de composants. Notre application étant distribué en tant qu'extension Firefox, l'enregistrement n'est pas nécessaire et se fait juste en fournissant le composant dans l'archive de l'extension.

La distribution de notre couche métier en tant que composant XPCOM se fait en compilant notre code C++ avec le *glue-xpcom* et le kit de développement Gecko, de manière à fournir une librairie dynamique. C'est cette librairie qui sera fourni dans l'archive de l'extension.

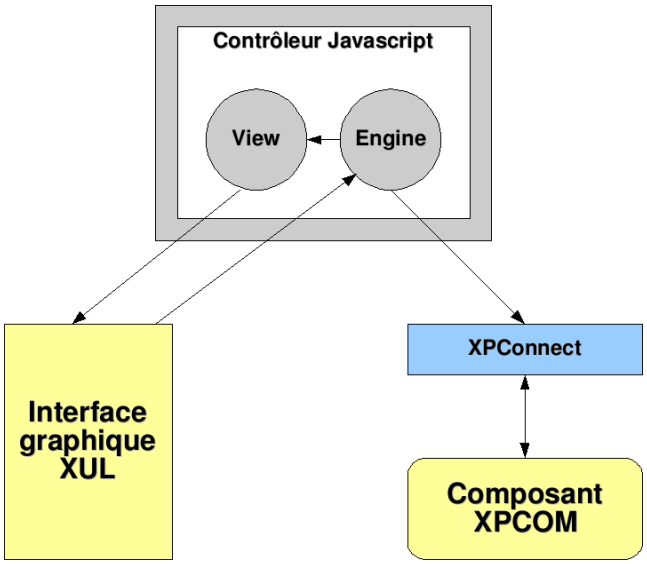
En complément de cette librairie, il est nécessaire de fournir un fichier *.xpt* qui TODO.

Chaque service fourni par le composant sera une méthode invoquable par Javascript, une fois le composant instancié.
Les cas d'utilisation distingués permettent de définir l'interface XPIDL nécessaire à nos besoins. Concrétement, le composant fournit sept services :

- *init(ip, path, port)* Initialise le moteur *mOle*.
- *hello(nom)* Récupére un message du serveur *molesys*.
- *connect(login, in, port)* Connecte le moteur *mOle* au réseau *molesys*.
- *joinChannel(name)* Rejoins un salon du nom en paramètre.
- *listChannels()* Liste tous les salons existants.
- *listChannelUsers(channel)* Liste tous les utilisateurs d'un salon
- *getHomepage(ip, port)* Appelle le service *homepage* chez un client *mOle* et récupére un flux *XML* correspondant à sa page d'accueil.

3.3.3 Contrôleur Javascript

Pour respecter l'architecture *MVC* que nous avons choisie, il est nécessaire d'implémenter un contrôleur qui puisse servir d'intermédiaire entre la couche métier et la vue graphique.



Architecture Modèle Vue Contrôleur

XPFE donne la possibilité d'utiliser l'ensemble de la bibliothèque de composants *XPCOM* au moyen de la technologie *XPConnect*, qui permet au développeur d'employer le *Javascript* pour charger un composant *XPCOM* et exploiter les services qu'il offre.
XUL est lui extrêmement exploitable et modifiable avec *Javascript*, en utilisant l'interface *DOM* offerte par le navigateur.

Du point de vue de la couche métier, le contrôleur doit se charger d'instancier le composant *XPCOM*, puis de demander à ce composant une interface exploitable. A partir de cet instant, le contrôleur peut appeler tous les services fournis par cette interface.
La classe *Engine* contient toutes les manipulations et utilisations directes du composant métier. Lorsqu'une modification de l'interface est nécessaire suite à une action demandée par l'utilisateur, le contrôleur se sert de la classe *View* pour directement influer sur l'interface graphique.

Cette classe *View* propose autant de prototypes qu'il y a de modifications possibles à faire dynamiquement sur l'interface graphique (ajout d'un onglet, mise à jour d'un arbre d'utilisateurs...).

Le fichier *mole.js* se charge d'enregistrer dans un gestionnaire tous les événements susceptibles d'être levés par une action de l'utilisateur.

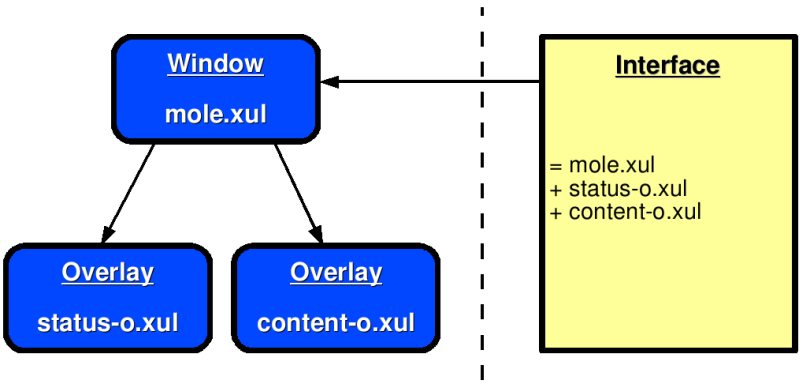
3.3.4 Interface graphique en XUL

Notre couche graphique XUL peut se décomposer en 2 parties :

- L'application riche
- L'intégration de l'application à Mozilla Firefox

L'application La couche graphique est séparée en plusieurs fichiers, utilisant le mécanisme d'*overlays* proposé par la solution XUL. Ce mécanisme très puissant permet non seulement de bénéficier d'une meilleure organisation de code, mais surtout, de faciliter l'évolution graphique et l'intégration de n'importe quel élément XUL au sein d'un fichier XUL.

Un *overlay* n'est autre qu'un fichier XUL contenant des éléments XUL, avec leur *id* de renseigné.
Le contenu des éléments DOM de même identificateur présents dans le fichier XUL chargeant les *overlays* et dans ces fichiers *overlays* sont fusionnés, et rendu dynamiquement dans la page.
Ainsi, l'on peut conserver une page principale très épurée, présentant uniquement une architecture à remplir, et qui le sera grâce aux fichiers *overlay* que cette page aura déclarée en entête.



Interface graphique *XUL* : overlays

Notre application étant majoritairement dynamique (ouverture et fermeture d'onglets, remplissage de ces onglets selon l'action demandée..), le contenu XUL demande a être constamment modifié : ajout d'un onglet, changement de la barre de status, remplissage d'un panel *XML*...
Le DOM permet de modifier dynamiquement un fichier *XUL*, au gré des actions demandées par l'utilisateur.
Cependant, *XUL* ne permet pas un tel dynamisme sans l'aide d'un langage de script. Le langage qui nous vient naturellement à l'esprit pour de telles modification est le Javascript, qui grâce aux fonctions DOM peut modifier l'arbre *XML* du fichier *XUL*.
Ainsi, cela explique pourquoi le fichier XUL principal (mole.xul) est particulièrement vide : la majorité de l'affichage sera créé de manière dynamique et non-déterministe.

L'intégration à Mozilla Firefox La création d'une extension ne suffit pas à integrer l'application dans Mozilla Firefox, un fichier *chrome.manifest* est là pour commander l'intégration.

L'intégration à *Firefox* se fait majoritairement par *overlays*. Il suffit pour modifier l'interface de *Firefox* de trouver le fichier *XUL* correspondant dans le *chrome*, et de lui appliquer un *overlay*.
Ainsi, l'on peut modifier aisément la barre de statut, la barre d'outils...

Nous avons décidé de n'intégrer l'application que par le menu *Outils*, puisque l'application serait déjà accessible par le menu *Add-ons*, ou à l'adresse peu parlante *chrome ://mole/content*.

4 Conclusion au projet

4.1 Résumé

Ce projet aura finalement traversé plusieurs phases résumant assez bien le déroulement de notre travail.

Tout d'abord Mole est la réponse à un besoin que nous ressentions dans notre utilisation régulière d'internet. En effet le partage de liens par exemple est toujours quelque chose d'assez peu pratique à réaliser. Dans le même ordre d'idées, permettre à un groupe d'amis d'accéder à un fichier n'est pas non plus chose facile. C'est alors qu'en creusant un peu cette idée nous avons choisi de réaliser une application qui pourrait résoudre ce genre de problèmes.

Après avoir obtenu l'accord de Mr.Seinturier pour être notre tuteur, et nous l'en remercions, nous nous sommes penchés sur l'architecture globale permettant de répondre aux besoins de notre application. Cette définition nous aura permis de mieux identifier les différentes couches de Mole.
Il était donc temps de se documenter sur les technologies utilisables pour chacune de ces couches. Après moult recherches aux quatres coins d'internet, nous avions finalement fini par choisir chacune des couches ; par curiosité et intéret pour les technologies retenues, ou encore pour la réputation dont elles jouissent.

Après avoir consulté Mr.Seinturier et débattu au sujet de nos choix, nous avons pu commencer la phase d'implémentation, après nous être réparti le travail. C'est alors que nous avons rencontré la majeure partie de nos difficultés. De fait, ce fut une phase assez longue, qui explique le fait que *Mole* soit encore actuellement en développement. Nous avons commencé par construire d'un côté le serveur *XML-RPC* et le client associé, et de l'autre un objet *XPCOM*, communiquant simplement avec une interface *XUL*. Puis nous avons lié le tout à l'interieur d'un composant pour finalement l'intégrer en tant qu'extension associée à *Firefox*.

Finalement, nous avons mené la partie la plus importante de la réalisation de molesys. Les techniques choisies ont été mis en oeuvre, et sont désormais fonctionnelles. Même si l'utilisation régulière n'est pas encore possible, il reste peu à faire pour pouvoir déjà s'en servir de manière convaincante. Vous trouverez par la suite les différentes voies à explorer pour continuer l'implémentation du système.

4.2 Difficultés rencontrées

Durant la période de conception et de développement, nombreux furent les obstacles rencontrés. Aucun ne nous aura définitivement bloqué, mais du temps y fut consacré. Nous n'avions pas prévu de perdre tant de temps à ses problèmes, qui se posèrent principalement au début du développement. Bien qu'ils nous aient limité quand à la réalisation finale de l'application, ses problèmes sont désormais maitrisés, et peuvent être expliqués.

Prise en main des librairies XML-RPC Nous avons du, pour les couches client et serveur, chercher et utiliser deux librairies *XML-RPC*. Ces deux librairies ont eu pour trait commun de ne pas proposer de manière très précise comment manipuler les paramètres que l'on donne aux requêtes, ou que l'on récupère à partir des réponses. Une fois cela maitrisé, nous avons pu utiliser les deux librairies sans rencontrer de nouveaux problèmes.

Maîtrise du langage C++ Le début du développement de la couche métier a connu de nombreux problèmes de compilation, qui ont été reglées à partir du moment ou nos connaissances en C++ se sont ameliorées.

Distribution du composant XPCOM L'exportation de la couche métier en tant que composant *XPCOM* n'a clairement pas été la phase la plus agréable du développement, nous offrant successivement problèmes, interrogations, et remises en cause de notre méthode de compilation. Car de toute la documentation que nous avons consulté pour la phase de compilation, moultes étaient les méthodes diffèrentes, sans détailler les avantages de certaines par rapport à d'autres. Ce qui, lorsque nous rencontrions un problème, ne nous rassurait pas sur nos choix.

- **Compilation en librairie dynamique *.so** En vue d'être utilisée dans un composant *XPCOM*, la couche métier doit être compilée en librairie dynamique. La distribution se fait en créant un module métier, qui doit respecter l'interface *XPIDL* définie pour le composant. Il est donc nécessaire que la compilation se fasse avec le *Gecko SDK*, pour construire correctement un composant.
  Nous avons donc procédé à la compilation en utilisant cette méthode, ce qui nous a généré un composant qui était fonctionnel avec *XULRunner*, mais non avec *Firefox*. Des explications que nous avons pu trouver sur Internet, le problème aurait pu venir de l'utilisation d'une librairie statique, une solution serait de compiler l'application avec les sources de *Firefox*. Après avoir avancé dans cette voie là, et sans résultat,

la solution est venue d'un développeur présent sur un réseau *IRC* de *Mozilla*, pour un problème qui était finalement situé dans nos options de compilation...

- **Enregistrement du composant XPCOM** Une fois le composant compilé, il doit être enregistré auprès du *chrome*, pour être accessible par n'importe quelle application basée sur *XPFE*. Nous utilisons l'extension *XPCOMViewer* pour vérifier si l'enregistrement est correctement effectué. L'enregistrement se fait en déplacant dans le dossier */usr/lib/mozilla-firefox/components/* la librairie. Pour que la mise à jour de la bibliothèque de composants se fasse, il est nécessaire de supprimer les fichiers *compreg.dat* et *xpti.dat* du dossier de votre profil *Firefox*, et de redémarrer *Firefox*. Cependant, la mise à jour d'un composant ne se faisait pas toujours, rendant cette manipulation toujours plus aléatoire quand au résultat.
  A cela, nous avons trouvé une solution extrêmement pratique, qui non seulement règle le problème de cette installation aléatoire, mais aussi nous évite de redémarrer souvent *Firefox* lors du développement (qui est souvent utilisé pour consulter de la documentation).
  L'utilisation d'un profil *Firefox* de développement permet d'enregistrer et d'utiliser des configurations diffèrentes de *Firefox*, et surtout, d'utiliser deux instances diffèrentes de *Firefox*. Cela permet lors d'une mise à jour de composant de conserver un profil de *Firefox* activé, et de ne redémarrer que celui qui contient le composant.
  De plus, nous avons découvert comment associer notre répertoire de travail à une extension, ce qui a comme avantage de ne plus procéder à la distribution du composant après chaque mise à jour de la couche métier, et de pouvoir utiliser l'extension immédiatement après recompilation du composant et rechargement du *chrome*.

4.3 Le futur de Mole ?

De nombreuses améliorations sont envisageables sur notre client dés maintenant. Les évolutions du réseau *Molesys* dépendront des besoins émis par les clients, et des constats que nous pourrions faire des utilisations courantes du réseau.
Parmi les évolutions les plus indispensables et prévues :

- Compatibilité avec *Firefox 3*
- Distribution d'une version pour autres systèmes : *Windows*, *Mac*
- Transfert de fichiers entre clients
- Définir des droits d'accès aux salons
- Définir une politique de communication entre réseaux de clients
- Gestion avancée des types de contenus transferables
- Optimiser l'intégration de l'application dans *Firefox*, selon les contenus que l'on partage
- Gestionnaire des contenus, de manière à offrir un espace rangé et optimisé aux utilisateurs

Le futur de *Mole* n'est pas encore défini, mais dépendra des développeurs. Nous envisageons de continuer à améliorer *Mole*, tout en espérant que des développeurs annexes seraient interessés par l'évolution du projet, la licence *GNU/GPL* apposée à l'application allant dans ce sens.

4.4 Apports personnels

Le travail accompli sur *Molesys* nous aura été utile sur deux plans distincts :

Apports techniques

En rapport avec l'architecture globale, certains aspects de l'application nécessitaient l'emploi de technologies inconnues. Ainsi, nous avons dû nous approprier ces technologies en autonomie, loin de la démarche cours/TP, dont nous avons l'habitude. Et ce simple fait de découvrir une technologie par ses propres moyens constitue déjà en soi un apport technique. Ou trouver des commentaires, des exemples ? Comment tester ?... Nous voilà maintenant plus disposés à nous former par nos propres moyens.

Ce projet aura aussi été un bon moyen de découvrir le *C++*. La couche métier de notre application n'étant pas d'une complexité renversante, son implémention aura constitué un bon premier exemple d'utilisation de *C++*. De plus l'utilisation et l'intégration d'une bibliothèque tierce à notre application aura également contribué à notre apprentissage de ce langage, et aux méthodes de compilation.

Utilisation du *XPFE*, le framework *Mozilla* : les applications développées par Mozilla jouissent d'une bonne réputation dans le monde du logiciel libre, et les outils mis à disposition pour le développeur sont de très bonne facture. Nous n'avons donc pas eu trop de difficulté à les prendre en main, et leur maîtrise nous auras déjà

permis une réalisation transversale (interface graphique d'une application pour le cours de COA), ce qui laisse présager que cet apport pourrait nous être à nouveau utile par la suite.

Finalement, nous avions entendu parler de *XML-RPC* mais ce fut notre première occasion de nous en servir concrêtement.

Apports en méthodologie de travail

Pour travailler efficacement et proprement sur ce projet nous avons du établir une méthodologie de travail en équipe (voir 1.3). Cette façon de faire nous aura convaincu de son sens et de l'importance à accorder aux méthodes pour mener à bien une réalisation.

La recherche de documentation en autonomie, l'identification de l'architecture de l'application ou encore les choix effectués sont autant d'éléments qui constituent un vrai projet. Et c'est ce genre de projets, de plus en plus conséquents, auxquels nous allons être confronté par la suite. C'est pourquoi le fait d'avoir traversé ces "phases" en autonomie s'inscrit déjà comme une expérience dans la gestion de la réalisation d'une application.

Finalement, le fait de travailler en équipe constitue une véritable force. Pouvoir échanger nos idées après la lecture de documents, proposer des idées ou encore se partager le travail ne sont que quelques exemples de l'ensemble des situations dans lesquelles le travail en équipe participe franchement à l'amélioration de la réalisation.

5 Annexe

5.1 Outils de développement

Extension developper [3] Extension *Firefox* spécialement conçue pour les développeurs d'extensions. Parmi les outils qu'elle propose :

- Rechargement du *Chrome*, pour éviter un redémarrage de Firefox lors d'un changement au niveau du composant *XPCOM*
- Un shell interactif *Javascript* pour influer directement sur les scripts en cours d'éxécution dans le navigateur
- Un éditeur dynamique *HTML* et *XUL* pour rapidement visualiser un rendu graphique.
- Un évaluateur *XPath*, un constructeur d'extensions *.xpi* ...

XPCOMViewer [4] Extension qui permet de visualiser l'arbre des composants *XPCOM* enregistrés dans *Mozilla*, et de consulter les services qu'ils fournissent. Utile pour savoir si un composant *XPCOM* a correctement été enregistré, et si l'extension est correctement installé.

Firebug [5] Extension reconnue par les webmasters en particulier, qui permet à tout instant d'inspecter le code de la page consultée, des feuilles de style et scripts chargés, mais surtout d'ajouter, supprimer, modifier dynamiquement les attributs et valeurs contenus dans ces fichiers. Très utile à des fins de déboguage graphique, ou de peaufinage esthétique.

Gecko SDK Kit de développement Mozilla nécessaire à la compilation d'un objet *XPCOM*. Permet de définir une interface de composant *XPCOM* avec l'outil *XPIDL* proposé dans le kit.

Documentation Mozilla Center [6] Documentation fournie par les développeurs *Mozilla*. Extrêmement exhaustive, une partie est consacrée au langage *XUL*, ainsi que pour le *Javascript*, l'enregistrement d'un composant *XPCOM*, la structure des extensions, les conseils de développement à suivre pour l'écriture de code destiné à la plateforme *Mozilla*... Une mine.

XMLRPCDebugger [7] Outil proposé les développeurs de la librairie *XML-RPC* en *PHP*. C'est un client Web de services *XML-RPC*, qui permet d'invoquer n'importe quel service sur n'importe quel serveur, pourvu que la requête soit bien formée. C'est très utile lors du développement du serveur *XML-RPC*, notamment pour se dédouaner à ce moment de la construction d'un client de test, et réduire le domaine de bugs possibles.

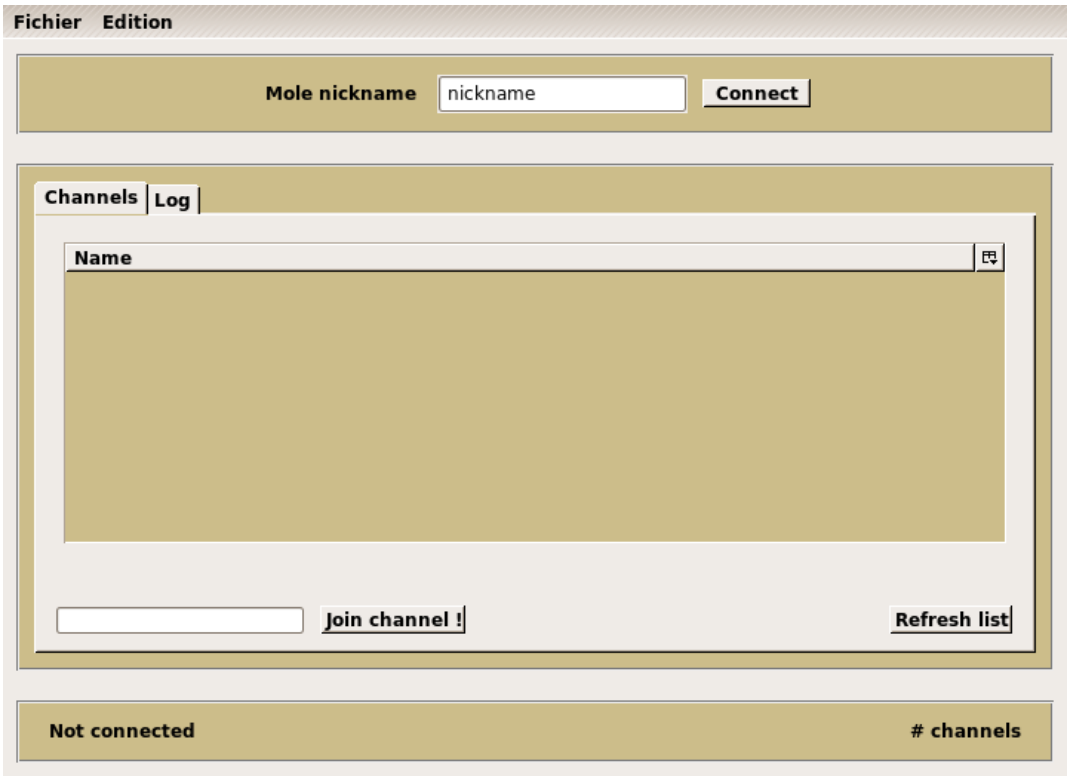
Sourceforge [8] Outil très utiles pour les équipes de développement de logiciels libres. Met à disposition des services tels qu'un Wiki, dépôt *Subversion*, serveurs de téléchargements...

Subversion Gestionnaire de versions de projet. Evite les conflits que peuvent surgir entre fichiers lorsque plusieurs personnes travaillent sur le même projet.

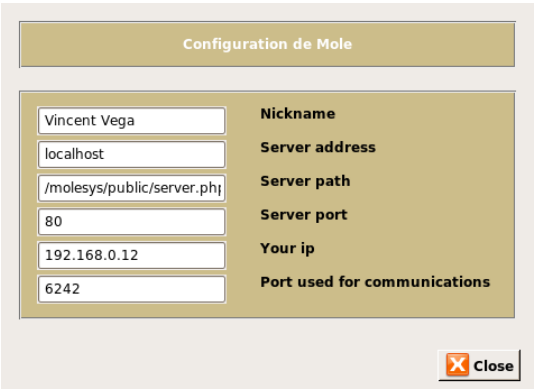
Vi Parce qu'il n'a jamais failli.

[3] *http ://ted.mielczarek.org/code/mozilla/extensiondev/*
[4] *http ://xpcomviewer.mozdev.org/*
[5] *http ://www.getfirebug.com*
[6] http ://developer.mozilla.org
[7] http ://gggeek.raprap.it/debugger/
[8] http ://www.sourceforge.net

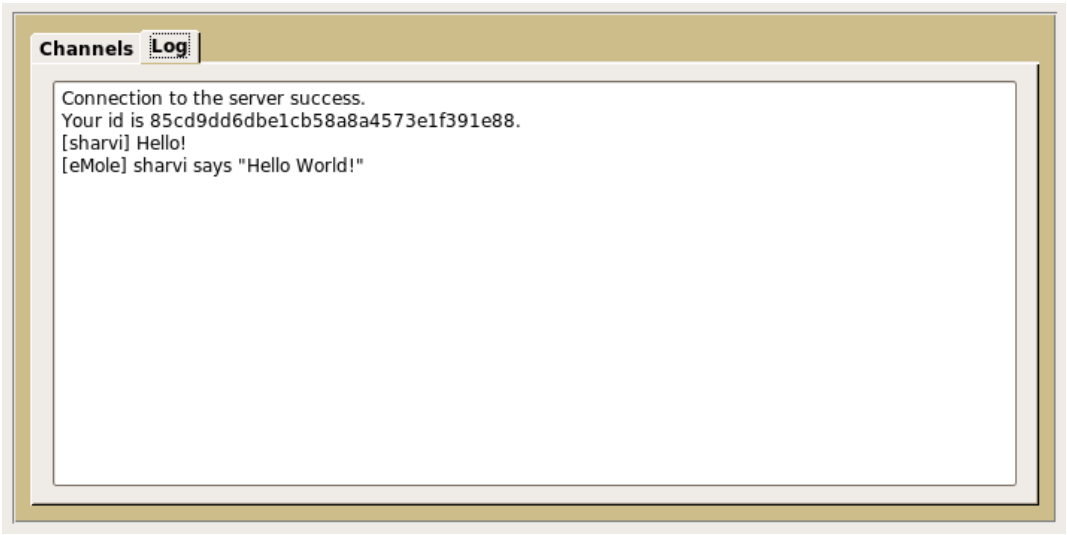
5.2 Captures d'écrans



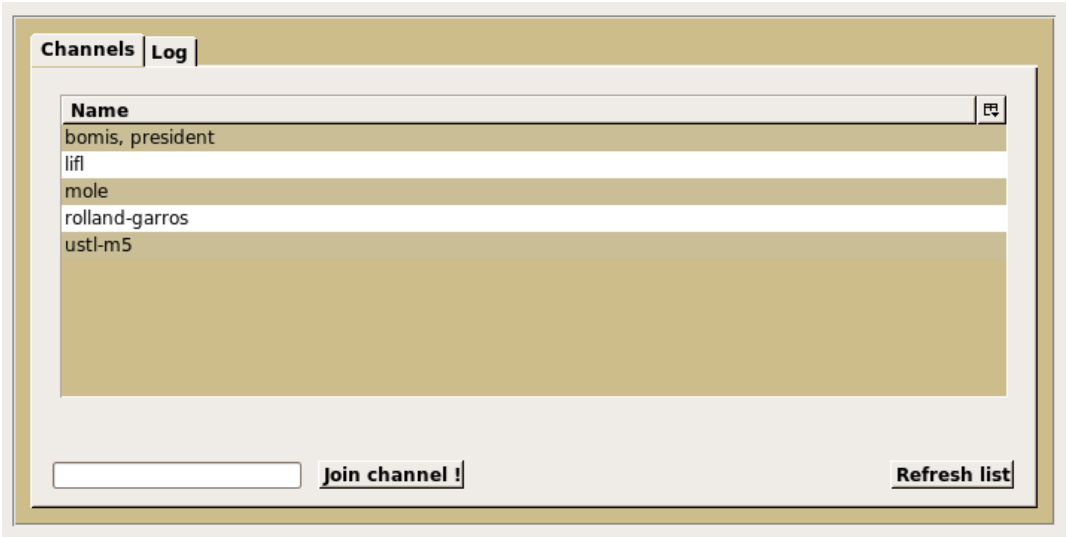
Se connecter au réseau *Molesys*



Paramètrage de l'application

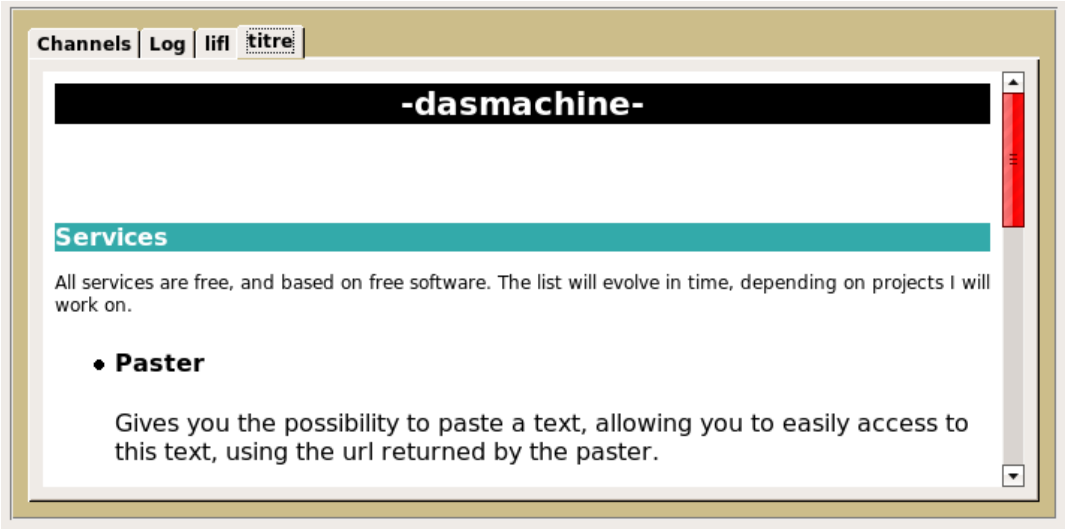


Consultation des logs



Liste des salons existants

Vue d'un salon et de ses utilisateurs



Récupération de la page d'accueil d'un client