Plugins

Sommaire

I. Description

Les plugins constituent l'intelligence du serveur. Ce sont eux qui implémentent les fonctionnalités exploitables par les clients, et les utilisateurs. Sans plugin, le serveur ne fait rien.

Un plugin est une bibliothèque dynamique (dll, so, dylib…) qui implémente des interfaces fournies par le serveur, dont les méthodes sont appelées en fonction de divers événements, qui peuvent provenir du réseau, ou d'interfaces graphiques par exemple.

En plus de ces événements, le serveur propose aux plugins une série d'APIs, qui leurs permettent d'exploiter certaines fonctionnalités du serveur, et de faciliter leur implémentation.

II. Composition

Un plugin est constitué de plusieurs éléments avec chacun un rôle précis :

Elément	Description	Requis
Plugin.*	La bibliothèque dynamique constituant le plugin. Son extension peut être dll, so, .sl, .a, .bundle ou dylib, en fonction du système pour laquelle elle a été compilée. En dehors de l'extension, le nom de ce fichier n'a pas d'importance.	Oui
Configuration	La configuration d'un plugin est stockée à l'intérieur de la configuration du serveur. Si elle n'est pas présente, le plugin est considéré comme désinstallé.	Oui
Queries.xml	Les plugins peuvent utiliser ce fichier pour stocker leurs requêtes SQL.	Non
Logo.png	Ce logo peut être utilisé par une interface graphique afin d'illustrer le plugin.	Non

Tous les éléments d'un plugin sont regroupés dans un dossier qui lui est propre, et qui se trouve lui-même dans le dossier « ***plugins*** » du serveur (par défaut). Il est libre d'y ajouter ses fichiers et dossiers, en fonction de ce dont il a besoin.

Le nom du dossier dans lequel se trouve le plugin est représente son identifiant unique. Si un plugin est présent dans plusieurs sous-dossiers dans le dossier des plugins, son identifiant sera le nom de ces sous-dossiers séparés pas des « **/** ». Par exemple, si un plugin se trouve dans le dossier plugins/Example/Basic, son identifiant sera Example/Basic.

III. Installation et chargement

Pour installer un plugin, sa configuration doit être présente dans le fichier de configuration du serveur, entre les nœuds ***configurations*** :

```
<configurations>
  <plugin id="Example/Basic">
    <!—Configuration of the plugin -->
  </plugin>
</configurations>
```

Pour charger un plugin, l'identifiant du doit être ajouté dans le nœud ***plugins*** de la configuration du serveur. Le plugin sera ainsi chargé au prochain démarrage du serveur. Un plugin doit être installé pour être chargé.

```
<plugins>
  <plugin>Example/Basic</plugin>
</plugins>
```

Il est possible d'installer, charger, décharger, et désinstaller un plugin à la volée, c'est-à-dire en plein fonctionnement du serveur, via l'API Plugins. Si un plugin est déchargé à chaud, le serveur lui laisse le temps de terminer toutes ses tâches en cours, avant de le décharger concrètement. Un plugin peu se décharger lui-même.

Lorsqu'un plugin est désinstallé, il doit s'assurer que tous les fichiers qu'il a créés durant son existence sont effacés.

IV. Métadonnées

Les métadonnées sont des informations fournies par chaque plugin, qui permettent aux utilisateurs de les identifier, et de comprendre leur utilité :

Métadonnée	Description
Name	Le nom réel du plugin. Celui qui sera affiché aux utilisateurs.
Brief	Une courte description de la fonction du plugin.
Description	Une description plus détaillée.
Autor	Nom de l'auteur du plugin.
Site	Le site web de l'auteur.
Email	Son adresse email.
Version	La version actuelle du plugin.
Licence	La licence sous laquelle le plugin est distribué. Les plugins officiels de LightBird sont en *Creative Common BY-NC-SA 3.0*.

V. Configuration

La configuration d'un plugin est stockée dans la configuration du serveur, à l'intérieur du nœud ***configurations*** :

```
<plugin id="Example/Basic">
  <translation>true</translation>
  <contexts>
    <context>
      <transport>TCP</transport>
      <protocol>HTTP</protocol>
      <port>80</port>
      <method>GET</method>
      <method>POST</method>
      <type>text/html</type>
    </context>
  </contexts>
  <timers>
    <myTimer1>1000</myTimer1>
    <myTimer2>60000</myTimer2>
  </timers>
  <resources>
    <resource alias="example.xml">Directory/Example.xml</resource>
  </resources>
</plugin>
```

Les plugins sont libres d'ajouter d'autres nœuds à leur fichier de configuration, et peuvent y accéder via l'API Configuration. Ils peuvent également créer leurs propres fichiers de configuration dans leur dossier.

1. Traduction

Si le nœud ***translation*** vaut **true**, la traduction du plugin est automatiquement chargée par le serveur. La langue utilisée est celle de ce dernier. Elle doit se trouver dans les ressources du plugin, sous le dossier ***languages***. Par exemple, le chemin du fichier de traduction en français du plugin **Example/Basic** est :

« **:plugins/Example/Basic/languages/fr** »

2. Contexts

Les contextes permettent au serveur de savoir quand appeler les interfaces réseau d'un plugin. Un plugin peut avoir plusieurs contextes simultanément, chacun étant dans un nœud ***context*** distinct.

Configuration	Description	Valeur par défaut
Transport	Indique quel protocole de transport est supporté par le plugin (TCP ou UDP).	Tous les protocoles de transport
Protocol	Le protocole pour lequel le plugin a été implémenté. Il ne sera appelé que pour les requêtes utilisant ce protocole. Les protocoles disponibles sur le serveur sont définis dans son le fichier de configuration. La valeur spéciale ***All*** en protocole indique que le plugin est indépendant du protocole, et sera donc utilisé pour tous les protocoles.	Aucun protocole
Port	Le port pour lequel le plugin a été implémenté. Il ne sera appelé que pour les requêtes provenant de ce port. Les ports ouverts sur le serveur sont définis dans son fichier de configuration. La valeur spéciale ***All*** indique que le plugin est indépendant du port, et sera donc utilisé pour tous les ports.	Aucun port

Method	La méthode de la requête pour laquelle le plugin a été implémenté. Il ne sera appelé que pour les requêtes appelant cette méthode.	Toutes les méthodes
Type	Le type MIME de la ressource pointée par la requête pour laquelle le plugin a été implémenté. Il ne sera appelé que pour les ressources avec ce type.	Tous les types

3. Timers

Les timers permettent aux plugins d'effectuer des opérations asynchrones de manière régulière. Plus clairement, le serveur appelle à intervalle régulier une méthode du plugin, dans un thread dédié.

Les nœuds timers de la configuration permettent d'appeler l'interface **ITimer** des plugins régulièrement. Le délai entre chaque appel est définit en millisecondes. Chaque appel est lancé dans un thread. Le timer est suspendu jusqu'à ce que le thread du précédent appel soit terminé. Le nom du nœud de chaque timer est un identifiant transmit au plugin afin de lui permettre d'identifier quel timer c'est déclenché. Dans la configuration ci-dessus, les timers myTimer1 et myTimer2 seront respectivement appelés toutes les secondes et toutes les minutes.

Les plugins peuvent modifier leurs timers pendant l'exécution du serveur, via l'API Timers.

4. Resources

Les plugins peuvent embarquer des ressources Qt dans leur bibliothèque dynamique. Cela permet par exemple d'y stocker une configuration par défaut, qui sera copiée dans la configuration du serveur lors de son installation. Reportez vous à la documentation Qt pour en apprendre plus sur le fonctionnement des ressources.

Chaque plugin possède un chemin de ressource unique, dans lequel il doit stocker ses ressources. Pour le plugin **Example/Basic**, ce chemin est « **:plugins/Example/Basic** ».

Le nœud **resources** de la configuration permet aux plugins de copier automatiquement lors de leur chargement des fichiers de leurs ressources vers leur répertoire s'ils n'existent pas. Par exemple, dans la configuration ci-dessus, le fichier de ressource nommé « plugins/Example/Basic/example.xml » sera copié dans le répertoire « Directory » du dossier du plugin, et se nommera « Example.xml ».

Pour que le serveur puisse copier automatiquement la configuration par défaut du plugin depuis ses ressources lors de l'installation, la ressource du fichier de configuration doit se nommer **configuration**.

De même, si le plugin utilise un fichier Queries.xml pour stocker ses requêtes SQL, il peut le mettre dans ses ressources sous l'alias **queries**.

Ce système permet à un plugin de créer automatiquement tous les fichiers dont il a besoin dans son dossier, seulement à partir de sa bibliothèque dynamique, qui agie ainsi comme un conteneur. De cette façon, seule la bibliothèque du plugin a besoin d'être distribuée. Le reste des fichiers pouvant être déployés à partir de ses ressources.

VI. Evénements de l'API

Les événements qui se produisent sur le serveur sont communiqués aux plugins via les interfaces qu'ils implémentent, en fonction des contextes définis dans leur configuration. Il existe des interfaces pour les logs, les timers, les extensions, les GUIs, mais ce sont les interfaces réseau qui sont les plus importantes.

Toutes les interfaces commencent pas la lettre « **i** » en majuscule. Celles relatives à l'API réseau sont de deux types :

- **Event** : Les events sont appelées lorsqu'un événement se produit. Tous les events commencent par « **IOn** ». Tous les plugins implémentant un event sont appelés dans l'ordre de leur chargement (c'est-à-dire l'ordre de leur apparition dans le fichier de configuration du serveur).
- **Handle** : Les handles permettent quand à eux de prendre la main sur une fonctionnalité du serveur (c'est-à-dire la remplacer). Ils commencent tous par « **IDo** ». Un handle ne peut être exécuté qu'une seule fois par requête. Si plusieurs plugins sont attachés au même handle dans le même contexte, seul le premier dans l'ordre de chargement des plugins sera appelé.

Pour savoir dans quelles circonstances les différentes interfaces sont utilisées, consultez leurs commentaires.

VII. Flux de données

Le schéma ci-dessous représente le flux de données des interfaces de l'API réseau.

Lorsque des données sont reçues, les interfaces **IDoRead**, et **IOnRead** sont appelés, puis les plugins doivent définir le protocole utilisé par la requête via l'interface **IOnProtocol**, après quoi la phase de désérialisation commence.

La désérialisation consiste à transformer les données brutes reçues (qui sont sous la forme d'un QByteArray), en un objet structuré, exploitable par les plugins qui vont se charger d'exécuter la requête. Cette phase peut être effectuée en trois étapes, selon les protocoles. Le **header** est d'abord désérialisé, puis vient le tour du **content**, et enfin du **footer**. Par exemple le protocole HTTP a un header, un content, mais pas de footer.

Puisque que les données peuvent arriver en plusieurs morceaux depuis le réseau, chaque étape est répétée autant de fois qu'il le faut pour la compléter. Par exemple si le header est reçu en trois fois, **IDoUnserializeHeader** sera appelé trois fois avant de passer au contenu. A chaque fois que de nouvelles données sont reçues, IDoRead, et IOnRead sont appelés, avant l'appel à une interface IDoUnserialize*.

La méthode **IOnUnserialize** est appelée après chaque étapes qu'un plugin implémente, ainsi qu'une de plus lorsque toute la requête est totalement désérialisée.

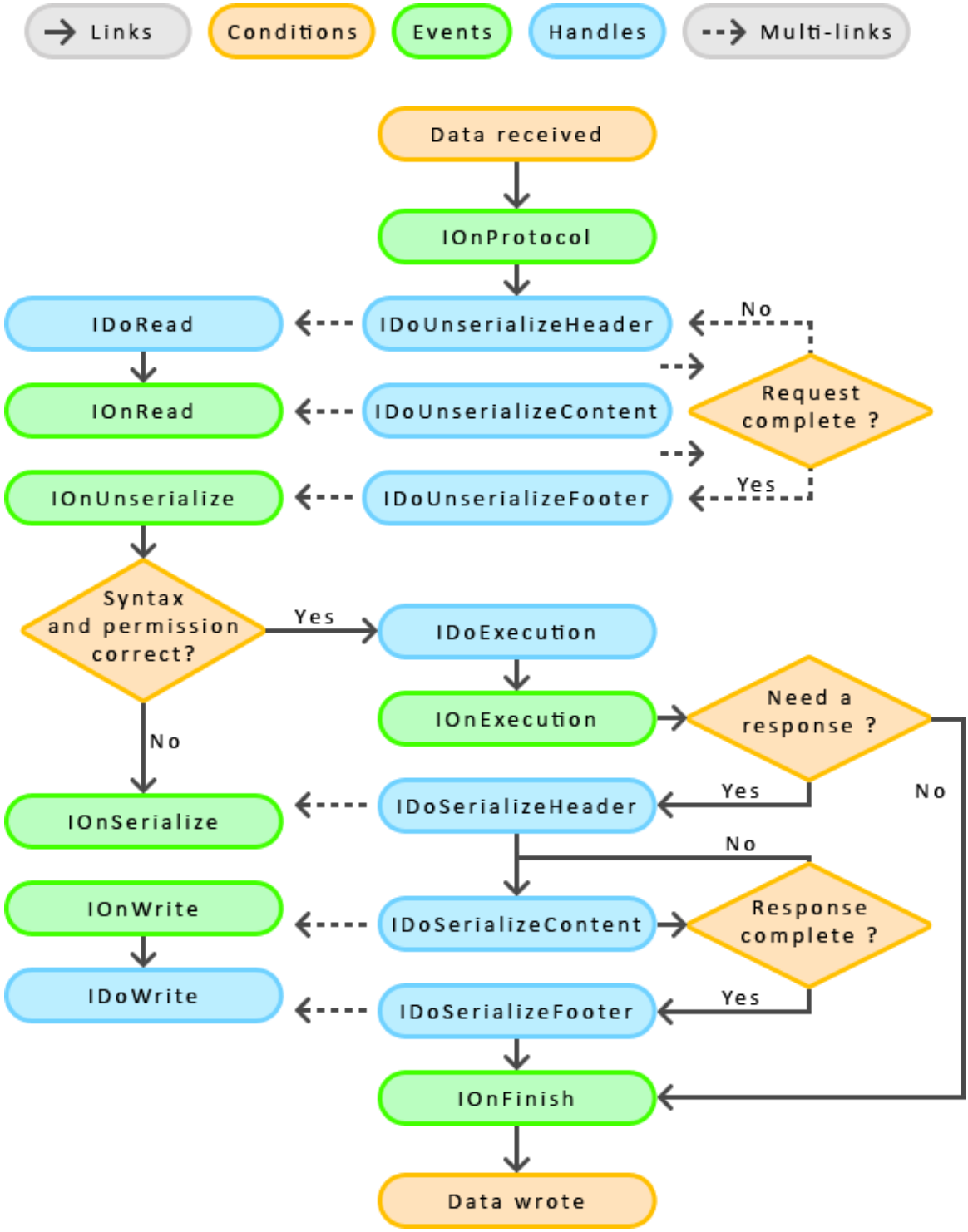
A tout moment lors de la désérialisation, les plugins peuvent indiquer au serveur qu'une erreur c'est produite, ou que le client n'a pas le droit d'effectuer la requête. Dans ce cas, l'étape suivante, à savoir l'exécution de la requête, ne sera pas effectuée, et la sérialisation (de l'erreur) commencera directement.

L'étape d'exécution est centrale dans le flux de données. C'est en effet dans ces méthodes que les plugins peuvent exécuter la requête, générer la réponse, ou décider qu'aucune réponse n'est requise.

Après l'exécution, la phase finale de ce processus est la sérialisation de la réponse. Le principe est de transformer l'objet qui représente la réponse, généré précédemment, en données brutes qu'il est possible d'envoyer au client par le réseau.

Tout comme la désérialisation, la sérialisation (qui est donc son inverse), se fait en trois étapes, à savoir la sérialisation du **header**, du **content**, et du **footer**. Le header et le footer ne sont appelés qu'une fois, tandis que le

content est appelé tant que tout n'a pas été totalement sérialisé, les gros content devant être envoyés en plusieurs fois. **IOnSerialize** est appelée avant chaque étape qu'un plugin implémente, plus une fois avant que la requête n'ait commencée à être sérialisée. Après chaque étape, les interfaces **IOnWrite** et **IDoWrite** sont appelés, et les données sont envoyées. **IOnFinish** est enfin appelée pour signaler que le traitement de la requête est terminé, qu'une réponse ait été envoyée ou non.



VIII. Interfaces

Comme évoqué précédemment, les plugins doivent hériter de certaines interfaces pour être appelés par le serveur. Cette partie décrie de manière succincte les interfaces implémentables. Pour des informations plus spécifiques, consultez les commentaires de ces interfaces.

1. IPlugin

IPlugin est l'interface de base des plugins. C'est elle qui est appelée par le serveur lors de leur chargement, et doit donc être implémentée par tous les plugins.

Méthode	Description
onInstall	Cette méthode sert à prévenir un plugin qu'il est en cour d'installation. Elle lui permet d'effectuer des actions spécifiques à l'installation, tel que créer les tables dont ils ont besoin dans la base de données, copier des fichiers...
onUninstall	Lors de leur désinstallation, les plugins doivent nettoyer toutes les modifications qu'ils ont effectuées lors de l'installation.
onLoad	Cet événement prévient le plugin qu'il est en cour de chargement. Cette méthode peut par exemple lui permettre d'instancier les objets dont il aura besoin lors de son fonctionnement. Peut être comparé à un constructeur.
onUnload	Est appelé lors du décharnement du plugin. Le plugin doit rapidement quitter proprement tous les threads avec lesquels il travaille, et libérer toutes les ressources allouées.
getMetadata	Cette méthode peut être appelée n'importe quand par le serveur, que le plugin soit chargé ou non, et permet de récupérer les métadonnées relatives au plugin.

2. ITimer

ITimer est décrite dans la partie V.3. C'est cette interface qui est appelés lors de chaque échéance d'un timer.

3. IEvent

Est appelée à chaque fois qu'un événement pour lequel le plugin a souscrit se produit. Voir la partie XII.

4. ILog

Permet à un plugin de prendre en charge les logs. ILog est appelé à chaque fois qu'un log est enregistré sur le serveur, et permet ainsi de l'afficher, de le sauvegarder dans un fichier, voir de l'envoyer à un autre programme, ou sur le réseau, à un serveur de log.

5. IGui

Cette interface permet de faciliter la création d'interfaces utilisateurs via les plugins. Par exemple, la méthode **gui** est appelée juste après le chargement du plugin, dans le thread GUI du serveur, ce qui permet au plugin de créer ses widgets, et de connecter ses signaux. Qt limite en effet les opérations GUI au thread qui a instancié QApplication, c'est-à-dire le thread principal du serveur.

6. Réseau

Les interfaces réseau participent au flux de données vu précédemment, et permettent donc de traiter les requêtes des clients.

Le tableau ci-dessous décrit les différentes interfaces que les plugins peuvent implémenter, ainsi que leur enchainement. Le contexte représente les conditions que les plugins doivent remplir pour se faire appeler par un événement. Cela est défini dans leur configuration. Par exemple un plugin qui veut être appelé par l'interface IOnConnect doit être configuré pour le même port (ou le même protocole) que celui auquel le client c'est connecté.

Interface	Description	Contextes
IOnConnect	Lorsqu'un client se connecte au serveur, ce dernier appel cette fonction. Le plugin qui l'implémente peut décider de refuser la connexion au client. Dans se cas, onClose est appelé, et le client déconnecté.	Protocoles Port
IOnDisconnect	Cet événement est appelé après la fermeture d'une connexion. Cette déconnection peut être initié par le serveur, le client, ou par un problème réseau. Elle permet généralement de libérer les ressources allouées pour le client durant sa connexion.	Protocoles Port
IDoRead	Uniquement disponible en TCP, IDoRead est appelée lorsque des données sont disponibles en lecture sur le socket du client, afin de permettre aux plugins de remplacer le la lecture des données normal du serveur. Ceci est par exemple utilisé dans l'implémentation d'un plugin gérant SSL.	Protocoles Port
IOnRead	Est appelé juste après chaque IDoRead, et permet de notifier aux plugins que des données ont été reçus.	Protocoles Port
IOnProtocol	Cette interface permet de définir quel est le protocole utilisé par le client dans ses requêtes. Elle est appelée juste avant IDoUnserializeHeader, autant de fois qu'il le faut pour identifier le protocole de la requête. Si aucun plugin n'est en mesure de trouver le protocole utilisé, toutes les données reçues jusque là sont supprimées. Le nom du protocole retourné par cette interface est utilisé par le serveur pour savoir s'il doit appeler les interfaces qui suivent (à l'exception d'IDoWrite, IOnWrite, et IOnFinish).	Protocoles Port
IDoUnserializeHeader **IDoUnserializeContent** **IDoUnserializeFooter**	Cette interface permet aux plugins de transformer les données reçus sur le réseau en un objet de type IRequest. Cet objet est indépendant du protocole de communication. Les IDoUnserialize* sont appelés en boucle tant que le plugin qui l'implémente estime que la requête n'est pas complète, et que d'autres données doivent être reçus.	Protocole Port
IOnUnserialize	IOnUnserialize est appelé *après* chaque étapes complétés des IDoUnserialize*, ainsi qu'une fois que la requête est complètement sérialisée ; sauf dans le cas de IDoUnserializeContent où IOnUnserialize est utilisé après chaque appels. Ceci permet de suivre l'avancement du téléchargement d'un fichier par exemple.	Protocole Port
IDoExecution	C'est dans cette interface que la requête est exécutée. Concrètement, l'objet IRequest permet de générer un objet IResponse qui sera plus tard sérialisé puis envoyer sur le réseau. Les plugins peuvent également décider ici de ne pas envoyer de réponse à une requête.	Protocole Port Méthode Type
IOnExecution	Cet événement est appelé à la suite de l'exécution de la requête (IDoExecution), et permet également de refuser l'envoi d'une réponse.	Protocole Port
IOnSerialize	Cet événement est appelé *avant* chaque appels aux interfaces	Protocole

	IDoSerialize*, ainsi qu'avant que commence la sérialisation.	Port
IDoSerializeHeader **IDoSerializeContent** **IDoSerializeFooter**	Les trois fonctions de sérialisation ont pour rôle de convertir l'objet IResponse en une chaine de données à envoyer sur le réseau, ce qui est l'inverse de la désérialisation. IDoSerializeHeader a pour rôle de sérialiser le header de la réponse, si le protocole utilisé en a un. IDoSerializeContent sérialise le contenu et est appelée en boucle, tant que le plugin qui s'en charge estime qu'il reste des données à envoyer. Engin, le footer est sérialisé par IDoSerializeFooter.	Protocole Port
IOnWrite	Permet d'être avertis que des données sont envoyées, et de les modifier si nécessaire. Est appelée après chaque appel aux interfaces IDoSerialize.	Protocoles Port
IDoWrite	Appelé à la suite de IOnWrite, IDoWrite autorise les plugins à remplacer l'écriture des données sur le réseau que fait normalement le serveur, comme pour IDoRead. C'est au plugin qui implémente cette interface d'envoyer les données au client.	Protocoles Port
IOnFinish	Une fois qu'une requête a été complètement traitée, et qu'une réponse a été envoyée ou non, cette interface est appelée. Cela permet par exemple à un plugin de déconnecter un client après chaque transaction, plutôt que de laisser la connexion active inutilement.	Protocoles Port

IX. APIs

Le serveur propose aux plugins une série d'interfaces de programmation, leur permettant d'effectuer divers opérations. Ces fonctionnalités sont regroupées par thème, chacune accessible via l'interface IApi, que les plugins peuvent récupérer et stocker lors de leur chargement.

API	Description
Configuration	Accès à la configuration XML du serveur et des plugins. Permet également de charger facilement un fichier XML propre à un plugin, et de l'utiliser : lecture, modification, et suppression de nœuds, ainsi que des opérations plus complexes via le DOM.
Database	Accès à la base de données. Permet d'exécuter des requêtes SQL, d'en charger depuis un fichier XML (queries.xml), ou d'accéder à l'abstraction de la base de données, qui facilite grandement son utilisation.
Guis	Autorise les plugins à effectuer des opérations basiques sur d'autres plugins, relatives à leur interface utilisateur.
Log	Permet d'enregistrer des logs sur plusieurs niveaux.
Réseau	Permet d'effectuer divers actions sur le réseau tel que déconnecter un client, ajouter ou supprimer un port TCP/UDP, et obtenir des informations sur les clients connectés.
Plugins	Les plugins peuvent gérer d'autres plugins grâce à cette interface. Elle permet en effet de charger, décharger, installer, et de désinstaller n'importe quel plugin, ou encore de consulter leurs états. Un plugin peut même se décharger lui-même, à la volée.
Extensions	Permet d'accéder aux extensions chargées sur le serveur. Voir la partie XIII.
Evénements	Permet d'envoyer et de recevoir des événements depuis le serveur et d'autres plugins.
Timer	Cette API permet de gérer les timers d'un plugin : changer leurs intervalles de déclenchement, en ajouter, ou en supprimer.

Pour des informations plus précises sur ces interfaces, consultez leurs commentaires.

X. TCP et UDP

Le serveur abstrait totalement la gestion des protocoles de transport TCP et UDP. Quelque soit le protocole utilisé, cela n'a pas d'impact directe sur les plugins, ce qui signifie qu'un plugin développé et testé avec TCP marchera en UDP (s'il on ignore les pertes possibles de paquets en UDP).

Notez tout de même que les interfaces IDoRead et IDoWrite ne sont appelés qu'en TCP.

XI. Abstraction de la base de données

L'abstraction de la base de données est un ensemble complet de classes facilitant l'usage de la base de données. Chaque interface représente une table, et chaque instance une entrée. Il est ainsi possible d'utiliser la base de données sans avoir à utiliser de requête SQL. Seules les opérations complexes, telles que les jointures peuvent nécessiter des requêtes fournies par les plugins.

La gestion des droits, ou de l'arborescence des fichiers et des dossiers sont également implémentés, ainsi que le support des concepts d'accessors et d'objects.

Les instances sont créées à partir de la méthode getTable de l'API de la base de données. Consultez ses commentaires pour obtenir un exemple de création d'instance.

XII. Evénements

En plus des événements générés par l'API décrit dans la partie VI, le serveur propose aux plugins un système plus souple qui leur permet d'échanger des événements entre eux simplement. Un événement est constitué d'un nom et d'une propriété optionnelle. Leur gestion se fait via l'API d'événements. Les plugins doivent déclarer les événements qu'ils souhaitent recevoir et utiliser l'une de ces deux méthodes pour les récupérer :

- **Push** : Implémenter IEvent qui sera appelée dans un thread dédié pour chaque nouvel événement.
- **Pull** : Demander directement les événements en attente à l'API d'événements.

Le serveur utilise aussi ce système pour envoyer quelques événements :

Evénement	Description	Propriété
server_started	Appelé une seule fois, après la fin de l'initialisation du serveur, juste avant d'entrer dans sa boucle d'exécution.	Aucune
plugin_loaded	Lorsqu'un plugin est chargé.	Id du plugin
plugin_unloaded	Un plugin est déchargé.	Id du plugin
plugin_installed	Un plugin est installé.	Id du plugin
plugin_uninstalled	Un plugin est désinstallé.	Id du plugin

XIII. Implémentation

Les plugins sont composés d'une classe principale, qui hérite de toutes les interfaces dont ils ont besoins. Cette classe doit impérativement être **thread safe**, car le serveur peut appeler les méthodes implémentées par un plugin dans de multiples threads simultanément. C'est donc aux plugins de protéger leurs données communes à plusieurs threads, comme les variables membres, afin d'éviter que deux thread y accèdent en même temps.

Les plugins situés dans le dossier **Example** montre divers implémentations simples. Le plugin **Example/Basic** peut être utilisé comme base pour créer n'importe quel plugin.

XIV. Extensions

Les extensions sont des plugins dont le but est d'étendre l'API du serveur, et donc de proposer plus services aux autres plugins. Une extension pourrait par exemple proposer de convertir des images dans d'autres formats (jpeg à png), et serait utilisé par tous les plugins ayant besoin de cette fonctionnalité.

Concrètement, les plugins extensions sont des plugins normaux, mais qui implémentent l'interface **IExtension**, en plus des autres interfaces. Cette dernière permet de déclarer les extensions que le plugin implémente. Une extension est une partie de ce plugin, qui implémente une interface présente dans le dossier **extensions** du serveur. Ce sont les plugins qui installent des interfaces dans ce dossier.

Un système de compteur interne au serveur permet de savoir quelles extensions sont utilisées par d'autres plugins à tout moment. Ceci permet d'éviter le déchargement d'un plugin dont les extensions sont utilisées ailleurs. Le plugin sera déchargé lorsque toutes ses extensions seront relâchées.