

# RAPPORT DE STAGE

*Guillaume ZAVAN*

*Stardragon Barcode Science & Technology Co.Ltd  
Appartement 10-2-2B, 86 avenue Yuantong  
Kunming (République populaire de Chine)  
[www.star-dragon.com](http://www.star-dragon.com)*

*Je tiens à remercier mon tuteur de stage, M. ZHANG Yu Ming,  
pour son accueil et son implication dans la réussite cette  
expérience pédagogique et personnelle hors-normes.*

# SOMMAIRE

<b>Introduction</b>	4
<b>Préparatifs</b>	
<i>Projet</i>	5
<i>Conception</i>	7
<b>Bases</b>	
<i>Affichage</i>	9
<i>Commandes</i>	11
<i>Sécurité</i>	15
<i>Requêtes</i>	17
<b>Développement</b>	
<i>Serveur</i>	20
<i>Client</i>	24
<i>Extensions</i>	27
<b>Conclusion</b>	30
<b>Abstract</b>	32

# A PROPOS ...

Introduction

## ENTREPRISE

---

L'entreprise *StarDragon Barcode* compte parmi la multitude de petites *start-up* technologiques ayant fait leur apparition sur le territoire Chinois durant les dix dernières années.

L'explosion du secteur informatique en asie du sud-est ainsi que le dynamisme impressionnant de l'économie Chinoise favorisent l'implantation de petites et moyennes entreprises dans différents secteurs jusqu'alors inexistantes, dont entre autres les technologies de l'information. Généralement créées de façon nettement plus hasardeuse qu'en Europe, ces modestes SSII occupent un large panel de domaines – systèmes embarqués, sécurité, *webdesign*, *hardware*, et ainsi de suite. La dimension parfois étonnamment réduite de ces sociétés, pourtant appelées occasionnellement à travailler dans des domaines extrêmement pointus, peut s'expliquer par le vide juridique total sur la propriété intellectuelle dans le droit Chinois : en effet, la contrefaçon, la reprise de code ou le détournement de contenus déposés à l'étranger passe ici pour une pratique admise. Si la qualité des produits reste un facteur de second plan – l'usage d'outils d'ingénierie type UML ou MERISE n'existant apparemment pas –, la réactivité et l'énergie déployée par le secteur surprend, en comparaison de ses équivalents occidentaux.

La société *StarDragon Barcode* a été fondée en 1999 par son actuel président, M. Zhang. Employant une dizaine de personnes et implantée dans un immeuble d'apparence parfaitement résidentielle, elle vend des systèmes de sécurité et de contrôle des accès, autant dans le domaine du *hardware* – portillons, cartes à puces – qu'en *software* – logiciels d'identification, contrôle. Les clients sont d'origine diverses, privés et publics – dont notamment l'université de Kunming.

## STAGE

---

Avant toute chose, il semble important de préciser que la façon dont les stagiaires sont accueillis et traités en Asie diffère fondamentalement de la vision occidentale. En effet, le fond de culture confucianiste considère l'apprentissage comme une étape fondamentale dans la vie d'un individu ; en conséquence, les stages – dans l'immense majorité des cas non-rémunérés – ne sont absolument pas un moyen pour l'entreprise de faire du profit. Au contraire, le stagiaire se voit confié un sujet similaire à celui qu'il pourrait rencontrer en temps que salarié, et doit le réaliser avec le soutien de son tuteur. Il n'est pas question ici de récupérer les codes pour une application ultérieure ; toutefois, j'ai traité le sujet aussi bien que ce rapport en admettant que ce soit le cas. Les exigences du tuteur ne s'allègent pas pour autant et son comportement envers nous n'a pas été plus clément qu'avec n'importe lequel de ses employés ; d'ailleurs, on notera que les exigences locales envers les étudiants dépassent largement celles des facultés parisiennes.

Au vu de la dimension restreinte des locaux de l'entreprise, le stage s'est fait à domicile, avec un rapport hebdomadaire en début de semaine. J'attire l'attention sur le fait que – aussi étonnant que cela puisse paraître pour le responsable d'une entreprise informatique – mon tuteur ne parlait qu'un anglais très approximatif, ce qui a entraîné quelques difficultés pour comprendre et analyser le sujet – nécessité de passer par le biais d'un traducteur. Toutefois, une fois la barrière de la langue passée, aucun autre problème n'a été à signaler.

# PROJET

*Préparatifs*

## APERÇU GLOBAL

---

Le projet proposé tient dans la création d'un système de gestion adressé au parc informatique d'une université. Les fonctionnalités offertes peuvent se subdiviser en deux branches distinctes : d'une part, le contrôle des accès aux différents postes en fonction de plusieurs critères ; d'autre part, la surveillance et le contrôle à distance, sous diverses formes, des ordinateurs utilisés par les étudiants.

Le système dans son ensemble peut être divisé en quatre programmes distincts :

1. Un programme serveur, en mesure d'identifier, de recevoir et de transmettre les requêtes des clients de façon fiable et sécurisé.
2. Un programme de connexion, permettant au client de s'identifier auprès du serveur et d'obtenir les droits suffisants pour utiliser la machine concernée.
3. Un programme de surveillance, lancé de façon invisible sur l'ensemble des ordinateurs clients, maintenant un contact régulier avec le serveur et en exécutant éventuellement les ordres.
4. Un programme de gestion, pouvant être activé par le client afin d'envoyer diverses commandes au serveur, éventuellement relayées vers les autres ordinateurs clients.

Évidemment, la liste ci-dessus ne saurait résumer de façon exhaustive l'ensemble du projet ; elle n'en donne qu'un aperçu synthétique et global.

## TRAVAIL ATTRIBUÉ

---

Mon travail consiste dans la création d'un protocole de communication simple et sécurisé entre les diverses applications clientes et le serveur. La liste ci-dessous en résume le contenu de façon succincte :

1. Définir un protocole d'envoi et de réception des commandes supportant les caractères au format Unicode, afin de permettre la transmission d'idéogrammes chinois au même titre que l'ensemble des caractères standard ASCII. Les commandes transférées doivent répondre à plusieurs normes de sécurité afin d'en pouvoir identifier l'émetteur, son niveau d'autorisation tout comme la validité de sa connexion.
2. Concevoir une application serveur fonctionnelle sous XP, Vista et l'ensemble des systèmes Unix, capable de recevoir de façon continue les requêtes des clients, les identifier et les traiter de façon entièrement automatisée.
3. Concevoir un modèle-type d'application client, compatible sous XP et Vista, permettant de transmettre et recevoir des commandes à l'application serveur. Elle doit nécessairement implémenter une interface graphique pour simplifier les démarches des clients.

4. Concevoir, à partir du modèle précédant, un formulaire sécurisé permettant l'authentification auprès du serveur et destiné à bloquer les machines clientes dès le démarrage du système d'exploitation jusqu'à la saisie d'identifiants valides.

# CONCEPTION

Préparatifs

## PRIORITÉS

---

- *Portabilité*

La première des priorités établies s'axe autour des évolutions futures du code et plus précisément sa relecture par des développeurs chinois amenés à s'en servir pour développer diverses applications clientes. Des différences notoires dans les méthodes de travail européennes et asiatiques et la barrière de la langue nécessitent l'usage d'un code structuré, clair et intégralement commenté en anglais.

- *Fiabilité*

La seconde priorité établie, encore une fois en prévision des évolutions ou reprises ultérieures du code, tient dans la fiabilité des programmes réalisés, et plus particulièrement dans le cas du serveur destiné à recevoir les commandes clientes. Il apparaît effectivement fondamental d'épargner un travail de « débogage » plus que pénible aux futurs utilisateurs ; d'où la nécessité de réaliser un ou plusieurs protocoles de test.

- *Sécurité*

La troisième priorité définie tient dans la résistance des programmes réalisés, et plus particulièrement du protocole de communication, à contrer d'éventuels envois faussés ou falsifiés. En effet, les applications finales étant destinées à l'usage en université d'informatique, il apparaît important de pouvoir déjouer d'éventuelles tentatives de piratage de la part des étudiants et autres utilisateurs finaux.

- *Intégrité*

Enfin, la quatrième et dernière des priorités établies tient dans la nécessité de rendre, en fin de mandat, un ensemble de programmes complet et correspondant aux attentes de l'entreprise. En effet, de par la forte autonomie dans mes choix de travail autant que par la distance me séparant de l'entreprise une fois le stage achevé, il apparaît difficile d'apporter un éventuel soutien à distances à mes éventuels successeurs en cas d'anomalie dans les fonctionnalités attendues.

## DÉCISIONS

---

- *Choix techniques*

Le sujet initial laisse libre choix quand aux langages de programmations et bibliothèques employées, sous réserve que les contraintes de portabilité soient respectés. Ont finalement été choisis les éléments suivants :

- **Langage C.** Le choix du – par ailleurs – plus que classique langage de programmation C se justifie par trois raisons : premièrement, la rapidité d'exécution,

en prévision d'un éventuel usage sur des machines anciennes ; secondement, la proximité avec les fonctions systèmes, afin de satisfaire autant que possible aux standards de sécurité notamment en terme de réseau ; et enfin troisièmement, mon expérience personnelle en la matière, relativement conséquente. Le choix d'un langage semi-compilé, à l'instar du Java, a été envisagé, mais rapidement rejeté, à cause des risques de détournement et du poids conséquent sur l'optimisation finale des programmes.

- **Bibliothèques Windows.** Les bibliothèques spécifiques utilisées pour les programmes XP et Vista sont les suivantes : pour le réseau, WinSocks ; pour les applications graphiques, WinAPI. Dans les deux cas, il s'agit des ensembles officiels créés par Microsoft dans le cadre du développement en C/C++ pour ses différents systèmes d'exploitation, et donc les plus indiqués dans le cadre du projet confié.
- **Bibliothèques Unix.** Dans le cas de l'application serveur, devant fonctionner sous des systèmes d'exploitation de type Unix, on fait usage de différentes bibliothèques dédiés au travail en réseau, et remplaçant grossièrement l'ensemble WinSocks. *Pour plus d'informations, référez-vous à la partie du rapport relative au serveur.*
- **Environnement de travail.** Le logiciel de développement utilisé est DevC++, en raison de sa capacité à gérer plusieurs projets partageant des ressources communes, mais également de par ses différents assistants en matière de création d'API graphiques.

Afin de faciliter le développement proprement dit, le programme utilise diverses autres bibliothèques qui ne se reflètent néanmoins pas dans la version finale – exception faite, éventuellement, dans certains fichiers d'en-tête.

- *Choix organisationnels*

Je divise mes dix semaines de travail de la façon suivante :

- **Deux semaines** de formation pour m'initier, de manière autonome, aux bibliothèques WinSocks et WinAPI. Il s'agit de ma première expérience de programmation graphique en C pour Windows d'où mon besoin de rattraper un certain retard en la matière, particulièrement en terme d'interfaces utilisateurs et réseaux.
- **Une semaine** de conception pour créer, coder et mettre à l'épreuve le protocole de communication client-serveur requis par le projet.
- **Une semaine** de conception pour décider d'une politique sécurisée d'authentification des utilisateurs auprès du serveur, incluant une méthode fiable d'encryptage, et pour coder les fonctions correspondantes.
- **Deux semaines** pour concevoir et coder le serveur ainsi que quelques programmes permettant de le mettre à l'épreuve.
- **Deux semaines** pour concevoir et coder une application client modèle, supportant les contraintes d'une interface graphique Windows et en mesure de communiquer avec le serveur précédemment réalisé.
- **Une semaine** pour valider les processus de communication client-serveur, particulièrement en cas de client multiples, et corriger d'éventuels défauts.
- **Une semaine** pour coder le client de connexion au serveur ainsi qu'une application de démonstration ; dans les deux cas, à partir du client modèle.



# AFFICHAGE

Bases

## FICHIERS

---

Le code dédié à l'affichage trouve sa source dans deux fichiers distincts :

1. La source, *shared/console.c*
2. La bibliothèque, *shared/libs/console.h*

Tous deux sont situés dans le dossier partagé, et donc identiques coté client et coté serveur.

## CONCEPTION

---

- *Contraintes*

L'affichage en console doit être clair et lisible. Destiné au serveur, il n'inclut aucune fonction de saisie – les commandes devant impérativement être passées par le biais d'une application cliente. Par ailleurs, il doit impérativement inclure un système de traitement des erreurs.

- *Prototype*

Une ligne d'affichage en console se présente de la façon suivante :

```
[ @ ] This is an exemple line
```

Le symbole @ peut être remplacé, selon le sens de la ligne retournée, par :

- Un point d'exclamation, en cas d'avertissement.
- Un point d'interrogation, en cas de requête.
- Un X majuscule, en cas d'erreur.
- Les lettres DEV, en cas d'information strictement adressée au développeur.
- Un vide, dans tous les autres cas.

Le système ne gère pas de mise en page en cas de ligne multiple – une telle fonction étant particulièrement complexe à coder en C et sans grand intérêt.

## DÉVELOPPEMENT

---

- *Impression*

```
void console( char *s , short i );
```

Cette fonction imprime une ligne de texte dans la console, le préfixe étant défini par la variable *i*.

```
void    console_dev();
```

Cette fonction imprime un message développeur à l'écran, sous réserve que le mode développement soit défini sur TRUE dans le fichier *console.h*.

```
void    console_line();
```

Cette fonction imprime une ligne à l'écran.

```
void    console_blank();
```

Cette fonction imprime un saut de ligne à l'écran.

- *Gestion des erreurs*

```
void    console_error( char *s , int i );
```

Cette fonction imprime un message d'erreur à l'écran et ferme le programme à la pression d'une touche quelconque ; celui-ci retourne le code d'erreur passé en argument *i*. Les erreurs sont définies dans un fichier de configuration dédié : */shared/libs/errors.h*.

- *Autres*

Plusieurs fonctions de saisie, créée dans le cadre du développement du programme, sont visible dans le fichier *console.c*. Elle n'ont toutefois aucune utilité dans les applications finales.

# COMMANDES

Bases

## FICHIERS

---

Le protocole de communication trouve sa source dans deux fichiers distincts :

1. La source, *shared/command.c*
2. La bibliothèque, *shared/libs/command.h*

Tous deux sont situés dans le dossier partagé, et donc identiques coté client et coté serveur.

## CONCEPTION

---

- *Contraintes*

Le protocole de communication client-serveur occupe un rôle fondamental dans la sécurité des échanges réseaux réalisés par le système final. En trois mots, il doit être :

- Accessible, pour pouvoir être facilement réutilisé par la suite.
- Sécurisé, pour éviter toute contrefaçon de la part de programmes pirates.
- Fiable, pour assurer la réception et l'interprétation d'un maximum de messages.

Ces trois contraintes, relativement aisées à respecter sous réserve d'une préparation suffisante, ont des conséquences pour toutes les applications du système.

- *Prototype*

Pour répondre aux contraintes du projet, les commandes destinées à un emplacement distant sont rédigées en trois temps distincts.

Dans un premier temps, l'utilisateur ou le programme inscrit la commande de base, répondant à la classique syntaxe :

```
NomCommande [Arg0] [Arg1] ... [ArgN]  
NomCommande [Args]
```

On notera que les arguments ne sont pas encadrés par des crochets ; le caractère d'espacement fait office de délimiteur.

Dans un second temps, le programme émetteur signe la commande par le biais d'une fonction dédiée, afin d'obtenir la syntaxe suivante :

```
#Login# MDP NiveauAutorisation ID Lieu NomCommande [Args]
```

Les différents éléments étant les suivants :

- *Login* le pseudonyme utilisé par l'émetteur.
- *MDP* le mot de passe de l'émetteur (cf. Sécurité).

- *NiveauAutorisation* le niveau d'autorisation annoncé par l'émetteur – entier.
- *ID* l'identifiant de la machine émettrice – entier.
- *Lieu* l'identifiant de localisation de la machine émettrice – entier.

On notera, d'une part, que le niveau d'autorisation annoncé par l'émetteur n'accorde pas automatiquement les droits associés (cf. Serveur) et, d'autre part, que le serveur possède des identifiants spéciaux, définis en librairie, permettant aux clients de l'identifier.

Enfin, dans un troisième temps, le programme encode la commande pour l'envoi :

```
@#Login#~MDP~NIVEAUAUTHORISATION~ID~LIEU~NOMCOMMANDE~[Args]@
```

L'encodage se fait de la façon suivante :

- Suppression des espaces multiples pouvant apparaître dans la commande.
- Validation de la syntaxe des guillemets – ouverture, fermeture.
- Validation des caractères de la commande – hors-guillemets.
- Conversion des caractères ASCII en majuscules – hors-guillemets.
- Conversion des espaces en un caractère spécial – hors-guillemets.
- Suppression des espaces avant et après la commande.
- Insertion d'un caractère d'encadrement avant et après la commande.

Les caractères admis dans la commande et hors-guillemets sont les suivants : 1 à 9, alphabet majuscule et minuscule, caractère d'espacement simple et encodé, caractère guillemet, double point, caractère « \_ ». Tout autre caractère dans la commande à encoder retourne un message d'erreur et la fermeture du programme (cf. Affichage). Par ailleurs, on notera que tous les caractères spéciaux – espacement, guillemets, encadrement – sont définis dans le fichier de configuration correspondant.

### • *Synthèse*

Ce protocole respecte correctement les exigences du projet :

- Les requêtes sont sécurisées, car aisément traçables par le biais des identifiants et de la localisation de l'émetteur.
- Le processus de lecture est fiable, car la syntaxe n'autorise pas de divergence et retourne un résultat négatif à la première anomalie.
- La syntaxe est accessible à n'importe quel informaticien pouvant être amené à reprendre le projet par la suite.

Par ailleurs, on notera qu'il est parfaitement possible de faire transiter des caractères chinois entre guillemets, sous réserve toutefois que l'affichage final – console, API – en permette l'affichage.

### • *Exemples*

Un exemple de l'encodage en deux temps :

```
Commande brute :    say_all    #Hello world !#
Commande signée :   #MonLogin# mdp 0 1 2 say_all    #Hello world !#
Commande encodée :  @#MonLogin#~MDP~0~1~2~SAY_ALL~#Hello world !#@
```

On notera que le contenu entre-guillemets (à savoir, « MonLogin » et « Hello world ! ») ne sont pas affectés par l'encodage de la commande.

## DÉVELOPPEMENT

---

L'ensemble des fonctions nécessaires au traitement des commandes sont contenues dans le fichier source correspondant. La liste ci-dessous décrit les plus importantes d'entre elles.

- *Encodage*

```
boolean  command_sign( char *command , ... );
```

Cette fonction signe et encode une commande conformément au protocole de communication client-serveur. La chaîne de caractères correspondante se voit directement modifiée dans la mémoire. En cas d'erreur, la fonction retourne FALSE sans modifier la commande passée en argument ; sinon, une valeur TRUE est renvoyée.

- *Validation*

```
boolean  command_check_frame( char *command );
```

Cette fonction vérifie la bonne disposition du caractère d'encadrement autour d'une commande passée en argument. Elle retourne TRUE en cas de succès, FALSE sinon.

- *Nettoyage*

```
void      command_clear( char *command );
```

Cette fonction nettoie une commande pré-encodée ayant transité à travers le réseau.

En effet, les essais de masse réalisés sur les transferts client-serveurs montrent d'occasionnelles anomalies sur les messages reçus – dont, entre autres, l'apparition de caractères inattendus en fin de chaîne. Cette fonction vise donc à assurer la lisibilité des commandes en isolant le texte contenu entre les caractères d'encadrement d'éventuels indésirables.

- *Lecture*

**Note** : les différentes fonctions de lecture suppriment automatiquement les caractères guillemets en cas de nécessité.

```
boolean  command_get_login( char *command , char *buff );
```

Cette fonction lit l'élément login d'une commande encodée et le place dans un buffer. Si l'élément en question se révèle non-précisé lors de l'envoi, la fonction retourne FALSE.

```
boolean  command_get_password( char *command , char *buff );
```

Cette fonction lit l'élément mot de passe d'une commande encodée et le place dans un buffer. Si l'élément en question se révèle non-précisé lors de l'envoi, la fonction retourne FALSE.

```
boolean  command_get_level( char *command , char *buff );
```

Cette fonction lit l'élément niveau d'autorisation d'une commande encodée et le place dans un buffer. Si l'élément en question se révèle non-précisé lors de l'envoi, la fonction retourne FALSE.

```
boolean  command_get_computer( char *command , char *buff );
```

Cette fonction lit l'élément ID d'une commande encodée et le place dans un buffer. Si l'élément en question se révèle non-précisé lors de l'envoi, la fonction retourne FALSE.

```
boolean  command_get_place( char *command , char *buff );
```

Cette fonction lit l'élément localisation d'une commande encodée et le place dans un buffer. Si l'élément en question se révèle non-précisé lors de l'envoi, la fonction retourne FALSE.

```
boolean  command_get_name( char *command , char *buff );
```

Cette fonction lit le nom de la commande et le place dans un buffer. Si celui-ci n'a pas été précisé lors de l'envoi, la fonction retourne FALSE.

```
boolean  command_get_arg( char *command , char *buff , int i );
```

Cette fonction lit l'argument *i* d'une commande – départ 1 – et le place dans un buffer. Si celui-ci n'a pas été précisé lors de l'envoi, la fonction retourne FALSE.

- *Autres*

L'ensemble des prototypes de fonctions utiles dans le cadre de la lecture/écriture sont définis dans le fichier *command.h* et nettement commentées dans le fichier *command.c*.

# SÉCURITÉ

## Bases

## FICHIERS

---

Le protocole de sécurité trouve sa source dans deux fichiers distincts :

3. La source, *shared/sha1.c*
4. La bibliothèque, *shared/libs/sha1.h*

Tous deux sont situés dans le dossier partagé, et donc identiques coté client et coté serveur.

## CONCEPTION

---

- *Contraintes*

La sécurité des mots de passe doit être assurée par un processus d'encryptage fiable, reconnu et, si possible, générant des chaînes de caractères de taille fixe – pour simplifier le stockage en mémoire vive du serveur ainsi qu'en base de données.

- *Prototype*

Après documentation sur le sujet, j'ai choisi le SHA-1 comme méthode d'encryptage pour les mots de passes.

- *Synthèse*

Cette méthode respecte les contraintes du projet, à savoir :

- Résistances aux attaques assurée malgré l'âge déjà avancé de la méthode.
- Génération de chaînes de caractères de taille fixe (40).

- *Exemples*

Un exemple de l'encodage en SHA-1 :

<b>Mot de passe brut</b>	: kawa
<b>Mot de passe encodé</b>	: F61CE7DB387D825478AF954C67C21217DDABFAD5

Le mot de passe présenté dans cet exemple est également celui utilisé par défaut par le serveur pour s'authentifier auprès des clients.

## DÉVELOPPEMENT

---

Le développement se base sur une source de Paul E. Jones, initialement protégée par droits d'auteur puis déposée sous licence GNU/GPL. Il s'agit d'une structure et de trois fonctions,

conservés – sous demande de l'auteur – dans leur intégralité, commentaires compris, et exploitées par le biais d'une quatrième fonction originale, décrite ci-dessous.

```
boolean to_sha1( char *str , char *buff );
```

Cette fonction convertit une chaîne de caractères au format SHA-1 et place le résultat dans un buffer dédié. Elle se base sur trois autres fonctions, qu'elle complète par l'ajout d'un caractère fin de chaîne « \0 » en position 40 sur le buffer.



# REQUÊTES

Bases

## REMARQUE

---

La liste ci-dessous résume les requêtes utilisés dans les applications client-serveur réalisées durant le stage. Relativement courte, elle vise à être enrichie au fil des différents développements ultérieurs du projet. Les systèmes de gestion des traitements sont visibles dans les deux fichiers *server\_treatments.c* – coté serveur – et *client\_treatments.c* – coté client.

Les arguments indiqués entre crochets sont obligatoires ; ceux entre accolades sont optionnels. On retiendra que les commandes sont ici indiquées sous leur forme basique ; à l'expédition, elles sont signées et encodées comme décrit dans le protocole de communication, ci-dessus.

## Liste des requêtes

---

- *Demande de fermeture*

```
Commande Client > Serveur uniquement  
SHUTDOWN_USER [Login] [Time]
```

Cette commande demande au serveur d'ordonner la fermeture d'un ordinateur client.

- L'argument *Login* contient le login de l'utilisateur devant être éteint.
- L'argument *Time* contient le temps, en secondes, avant la fermeture de l'ordinateur à dater de la réception du message par le client.

Note : nécessite un niveau d'autorisation 4.

- *Erreur*

```
Commande à double sens  
ERROR [Id]
```

Cette commande indique le renvoi d'une erreur d'un côté ou de l'autre d'une transaction. Elle entraîne dans la majorité des cas la fermeture de cette dernière.

- L'argument *Id* contient le code de l'erreur, tel que décrit dans le fichier */shared/libs/errors.h*.

- *Fermeture*

**Commande Serveur > Client uniquement**  
SHUTDOWN [Time]

Cette commande ordonne à un ordinateur client de s'éteindre.

- L'argument *Time* contient le temps en secondes et à partir de la réception du message avant l'arrêt de la machine.

- *Message console*

**Commande à double sens**  
SAY [Message]

Cette commande affiche un message dans la console du destinataire.

- L'argument *Message* contient le message à afficher.

Note : commande de démonstration, sans utilité en usage normal.

- *Message général*

**Commande Client > Serveur uniquement**  
SAY\_ALL [Message]

Cette commande affiche un message signé dans la console de tous les clients, exception faite de l'expéditeur.

- L'argument *Message* contient le message à afficher.

Note : commande de démonstration, sans utilité en usage normal.

- *Login*

**Commande Client > Serveur uniquement**  
LOGIN [Login] [Password\*]

Cette commande authentifie le client auprès du serveur.

- L'argument *Login* correspond au login utilisateur.
- L'argument *Password* correspond au mot de passe utilisateur encodé en SHA-1.

Note : la localisation et l'identifiant machine sont communiqués en préfixe de la commande ; le niveau d'autorisation se situe quand à lui dans la base de données du serveur.

- *Ping*

**Commande Client > Serveur uniquement**  
PING

Cette commande indique que le client souhaite recevoir les message lui étant adressés mémorisés sur le serveur. Elle est appelée de façon invisible et ponctuelle par le processus client lancé en arrière-plan sur les machines concernés par le système.

- *Validation*

```
Commande Serveur > Client uniquement  
OK {Level}
```

Cette commande clos une transaction en indiquant sa réussite.

- L'argument *Level*, optionnel, se produit dans le cas précis d'une réponse à un login utilisateur. Si réussite, il est égal au niveau d'autorisation attribué à ce dernier.

# SERVEUR

Développement

## FICHIERS

---

Le serveur trouve sa source dans six fichiers distincts :

- Le code principal, *main.c*
- Le gestionnaire général, *server\_manager.c*
- Le système de traitement des requêtes, *server\_treatments.c*
- Le système de gestion des clients, *server\_logs.c*
- Le système de gestion des commandes, *server\_messages.c*
- La bibliothèque de configuration, */shared/libs/server.h*

On notera que la configuration du serveur se situe dans le répertoire partagée et peut donc être utilisée par les programmes clients. Par ailleurs, les deux fichiers suivants interviennent dans le fonctionnement du programme :

- Le gestionnaire réseau, */shared/libs/network.h*
- Le gestionnaire de service Windows, */shared/winspecific.h*

## CONCEPTION

---

- *Contraintes*

Le programme serveur prévu par le projet se présente sous une forme somme toute assez classique. Une fois lancé sur une machine virtuellement active en permanence, il doit pouvoir détecter, analyser et traiter des connexions en provenance d'un nombre variable de programmes clients, sans se bloquer – y compris en cas d'erreur – ni ralentir les ordinateurs distants. De façon synthétique, il doit inclure les fonctionnalités suivantes :

- Réception et envoi de requêtes suivant le protocole précédemment défini.
- Mémorisation des clients connectés.
- Mémorisation des messages à expédier.
- Traitement des requêtes reçues et génération de réponses.
- Système de gestion des erreurs non-bloquant.

Par ailleurs, le sujet inclut d'éventuelles évolutions du serveur – dans le cas, par exemple, de l'ajout de nouvelles fonctionnalités –, d'où la nécessité de rédiger un code clairement défini et explicites pour les développeurs appelés à le modifier.

- *Prototype*

Le programme fait usage du protocole UDP pour gérer les échanges réseau. Au contraire de la très populaire méthode TCP/IP, le protocole UDP n'attend aucune réponse du destinataire suite à l'envoi d'un paquet ; d'où la possibilité de traiter rapidement et de manière linéaire un grand nombre de clients, même si l'un d'eux vient à se déconnecter ou rencontrer une avarie technique.

Sous Windows, les protocoles réseaux sont implémentés par la bibliothèque unique *winsocks2.h* ; sous Unix, l'affaire se révèle plus complexe, et nécessite l'emploi de plusieurs fichiers d'en-tête : *sys/types.h*, *sys/socket.h*, *netinet/in.h*, *arpa/inet.h*, *unistd.h* et *netdb.h*. Pour le reste, les différences de fonctionnement entre les deux systèmes d'exploitation sont mineures : quelques constantes non-définies, fonctions nommées différemment, ainsi que l'activation d'un service – WSA – sous Windows. La compatibilité se voit toutefois intégralement assurée par les fichiers *network.h* et */shared/winspecific.c*.

L'ouverture du serveur, assurée directement dans le *main*, suit un cheminement courant pour ce type de programme :

- Ouverture, si nécessaire, du service WSA.
- Création et validation d'un socket d'écoute.
- Création d'une interface réseau sur un port pré-défini.
- Connexion du socket et de l'interface, validation.
- Mise à disposition des clients.
- Création de deux listes réactives : clients connectés et messages en attente.

Une fois le socket d'écoute correctement initialisé, le programme entame une boucle d'attente des connexions clientes. Pour pallier à d'éventuels conflits, les transactions se basent sur deux règles simples :

- Le client ouvre toujours la transaction ...
  - Par une requête ponctuelle de type PING, uniquement pour se signaler auprès du serveur et recevoir d'éventuels messages lui étant destiné.
  - Par une requête particulière – par exemple, LOGIN ou SAY – entraînant des conséquences sur le comportement du serveur.
- Le serveur ferme toujours la transaction ...
  - Au moyen d'une commande de type OK, en cas de succès de la transaction.
  - Par le biais d'une commande ERROR, en cas d'échec.

Une transaction client-serveur consiste donc en un échange de messages clos de façon automatisée. Pour chaque transaction, le client doit se connecter puis se déconnecter, évitant ainsi l'usage d'une fonction multipliant les processus – un par client –, complexe à coder en assurant la compatibilité multi-plateforme.

Le serveur conserve en mémoire deux types d'informations.

Premièrement, la liste des clients connectés. Avant toute tentative de transaction, un client doit préalablement se signaler auprès du serveur par le biais d'une commande de type LOGIN, contenant son login et son mot de passe – encodé – pour vérification sur une base de donnée. En cas de succès, le serveur mémorise les identifiants, le niveau d'autorisation – indiqué par la base de donnée –, l'ID machine et l'ID localisation du client. L'entrée reste en mémoire jusqu'à expiration, déconnexion ou connexion sur un autre poste de la part de l'utilisateur. Si un utilisateur n'apparaissant pas dans cette liste tente de passer n'importe quelle autre requête auprès du serveur, il se voit systématiquement refusé. Techniquement, la mise en mémoire utilise une liste chaînée – sources : *server\_logs.c* et */shared/libs/server.h*.

Secondement, la liste des messages à expédier. Étant donné que le serveur n'ouvre jamais de connexion vers les clients, il doit attendre que ces derniers se manifestent pour expédier d'éventuelles consignes. La mise en mémoire se fait donc sous forme requête-destinataire ; les entrées sont effacées dès l'envoi. Le programme fait usage d'une liste chaînée – sources : *server\_messages.c* et */shared/libs/server.h*.

A la réception d'une requête, le serveur recherche le traitement correspondant dans une liste bien déterminée, mémorise d'éventuels nouveaux messages et retourne une réponse immédiate. La liste des traitements se situe dans le fichier *server\_treatments.c*. On notera que les

commandes de type LOGIN n'y figurent pas ; pour diverses raisons techniques, elles sont traitées en amont, directement dans le gestionnaire de transaction.

Enfin, les éventuelles erreurs pouvant être générées par le serveur lors du traitement – argument manquant, commande inexistante, utilisateur non-connecté – sont stockées dans la bibliothèque partagée */shared/libs/errors.h*. Si générées, elles sont expédiées au client par le biais d'une commande ERROR comprenant en argument le code de l'erreur.

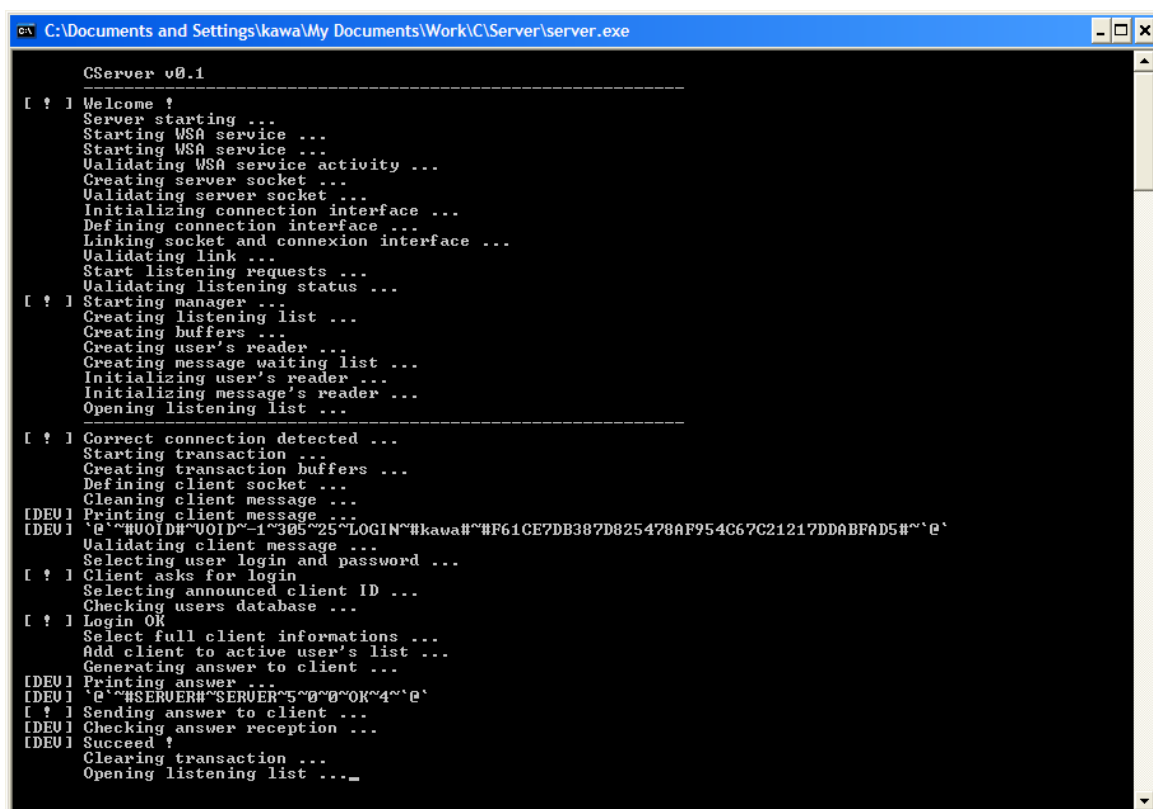
- *Synthèse*

Le modèle de serveur proposé ci-dessus répond globalement aux exigences du projet :

- **Fiabilité** : pas de multiplication des processus ou de risques de blocage liés à la déconnexion inattendue d'un client – grâce aux règles de transaction et l'usage du protocole UDP. Par ailleurs, les erreurs sont gérées de façon simple et efficace, par le biais de la commande ERROR.
- **Souplesse** : possibilité de rajouter à volonté des traitements par simple inscription dans le fichier correspondant.
- **Portabilité** : le serveur fonctionne parfaitement sous XP, Vista et Linux ; il a été testé en ce sens par des envois de masse.

On notera toutefois que les fonctions d'encodage et décodage des commandes, un peu complexes, risquent de ralentir une machine trop vieille. Les exigences du programme demeurent toutefois largement dans la limite du raisonnable.

- *Exemple*



```
CServer v0.1
[ ! ] Welcome !
Server starting ...
Starting WSA service ...
Starting WSA service ...
Validating WSA service activity ...
Creating server socket ...
Validating server socket ...
Initializing connection interface ...
Defining connection interface ...
Linking socket and connexion interface ...
Validating link ...
Start listening requests ...
Validating listening status ...
[ ! ] Starting manager ...
Creating listening list ...
Creating buffers ...
Creating user's reader ...
Creating message waiting list ...
Initializing user's reader ...
Initializing message's reader ...
Opening listening list ...

[ ! ] Correct connection detected ...
Starting transaction ...
Creating transaction buffers ...
Defining client socket ...
Cleaning client message ...
[DEU] Printing client message ...
[DEU] 'e~#U0ID#~U0ID~1~305~25~LOGIN~#kawa#~#F61CE7DB387D825478AF954C67C21217DDABFAD5#~'e'
Validating client message ...
Selecting user login and password ...
[ ! ] Client asks for login
Selecting announced client ID ...
Checking users database ...
[ ! ] Login OK
Select full client informations ...
Add client to active user's list ...
Generating answer to client ...
[DEU] Printing answer ...
[DEU] 'e~#SERVER#~SERVER~5~0~0~OK~4~'e'
[ ! ] Sending answer to client ...
[DEU] Checking answer reception ...
[DEU] Succeed !
Clearing transaction ...
Opening listening list ...
```

*Lancement du serveur, réception d'une commande de type LOGIN et réponse*

## DÉVELOPPEMENT

---

```
int      main();
```

Même s'il ne s'agit pas d'une fonction à proprement parler, il faut signaler que le *main* du programme contient l'intégralité du processus d'initialisation du serveur.

```
void      server_manager( SOCKET *s );
```

Cette fonction contient la boucle globale du serveur, assurant les différentes transaction avec le client. Virtuellement, elle ne se ferme jamais une fois appelée. A noter que le socket passé en argument doit d'ores et déjà être configuré pour recevoir des requêtes au moment de l'appel de fonction.

```
void      server_transaction_manager( SOCKET *s , ... );
```

Cette fonction gère une transaction – échange d'un ou de plusieurs couples de question et réponse – entre le serveur et un client préalablement connecté. Elle inclut le gestionnaire de login ainsi que l'appel à la fonction de traitement.

```
boolean   server_treatments( char *command , char *answer , ... );
```

Cette fonction transmet une commande cliente devant être traitée par le serveur. Elle inscrit dans un buffer la réponse à la requête – réussite, erreur ou autre – qui sera par la suite expédiée au client concerné – il peut s'agir d'un message stocké en mémoire. En cas de commande inconnue, de client non-connecté ou d'information manquante, la fonction retourne FALSE ; dans tous les autres cas, elle retourne TRUE.

```
void      st_ping( ... );  
void      st_say_server( ... );  
void      st_say_all( ... );  
void      st_shutdown_user( ... );
```

Les fonctions ci-dessous correspondent à différentes commandes pouvant être traitées par le serveur. Elles ne retournent rien, mais inscrivent le résultat de la requête dans le buffer de réponse communiqué par le biais du gestionnaire de traitements. Pour plus d'informations, référez-vous à la partie « Requêtes » de ce rapport.

# CLIENT

Développement

## FICHIERS

---

Les différentes applications clientes utilisent les six fichiers suivants :

- Le programme principal, *main.c*
- Le gestionnaire réseau, *client\_manager.c*
- Le gestionnaire de communication, *client\_sender.c*
- Le gestionnaire de traitements, *client\_treatments.c*
- Le gestionnaire d'erreurs, *client\_errors.c*
- La bibliothèque de configuration, */shared/libs/client.h*

Par ailleurs, les deux fichiers suivants interviennent dans le fonctionnement des applications :

- Le gestionnaire réseau, */shared/libs/network.h*
- Le gestionnaire de service Windows, */shared/winspecific.h*

## CONCEPTION

---

- *Contraintes*

L'objectif prévu par le projet tient dans réalisation d'un modèle d'application cliente, facile à comprendre et à modifier, répondant à deux principales contraintes :

- L'accessibilité : le programme doit présenter une interface graphique fonctionnelle sous Windows.
- La compatibilité : le programme doit pouvoir se connecter au serveur de façon automatique et échanger avec lui des messages.

Deux exemples d'applications finales sont présentés dans la partie « Extensions » du rapport.

- *Prototype*

Les applications clientes, évidemment compatibles avec le serveur, utilisent le protocole UDP pour émettre et recevoir des messages. A chaque envoi de requête, une connexion s'établit avec le serveur – sous réserve que celui-ci ne traite pas d'autre client ; le client envoie une requête et reçoit une réponse, possiblement définitive ; si non, il doit alors retourner une nouvelle requête, et ce jusqu'à réception d'un message de fermeture – type OK ou ERROR. Je ne reviendrais pas plus longuement ici sur le processus utilisé pour l'envoi et la réception des commandes, déjà largement décrit dans les section « Serveur » et « Commandes ».

La partie graphique se base sur la bibliothèque Microsoft par défaut pour Windows XP et Vista, à savoir *windows.h*. Elle inclut l'ensemble des fonctions nécessaires à la création d'une API événementielle. Un peu complexe, notamment à cause d'un gestionnaire d'événements particulièrement élaboré, elle se révèle une fois déployée assez facile d'utilisation – rajout, suppression d'éléments dans la fenêtre sont assurés par des fonctions bien définies. L'ensemble de la documentation – en anglais – est disponible à l'adresse suivante :



[http://msdn.microsoft.com/en-us/library/cc433218\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/cc433218(VS.85).aspx)

Le détail des différentes fonctions à appeler pour permettre l'ouverture d'une fenêtre se révélant pesant et sans grand intérêt, je me contenterai d'un aperçu synthétique du système.

Une API Windows se divise en deux principales parties :

- Le lanceur ou *main* du programme – du moins, le cas échéant, *WinMain()* – prépare l'application. Il contient les informations synthétiques sur la fenêtre, stockées dans une pseudo-classe de type *WNDCLASS*.
- Le gestionnaire d'évènement – *WinProcess()* – détecte toutes les interactions entre l'utilisateur et le programmes : clics, saisie au clavier, etc. Il peut en outre prendre en compte des évènements ajoutés par le programme lui-même.

Au fil de l'utilisation du programme, le gestionnaire d'évènement traite de différentes façons les actions réalisées par l'utilisateur ; sur le plan linéaire, son fonctionnement correspond à une boucle ne se rompant qu'à la fermeture de la fenêtre. Dans le cadre du projet, il traite également les appels réseaux – par exemple, si l'utilisateur clique sur le bouton « Login », le gestionnaire sélectionne les identifiants dans les champs prévu à cet effet, les concatène et envoie une commande de type LOGIN via le gestionnaire prévu à cet effet.

Par ailleurs, à l'instar du serveur, les programmes clients possèdent leur propre gestionnaire de traitement, capable de récupérer, analyser et répondre aux requêtes.

- *Exemple*

Le modèle d'application cliente étant vide pour permettre son utilisation dans le cadre des développements ultérieurs, je vous invite à consulter la section « Extensions » pour visualiser les programmes réalisés.

## DÉVELOPPEMENT

---

```
int WinMain( ... );
```

Le *main* du programme. Implémente la fenêtre – mais pas son contenu, réalisé lors de l'évènement « Création de la fenêtre » dans le gestionnaire.

```
LRESULT CALLBACK WinProcess( ... );
```

Le gestionnaire d'évènement, lancé de manière automatique une fois la classe fenêtre correctement initialisée.

```
int client_connect( SOCKET *s );  
void client_disconnect( SOCKET *s );
```

Ces deux fonctions permettent à l'application cliente de se connecter et se déconnecter automatiquement du serveur via le service WSA exigé par Windows. Elles sont respectivement appelées avant et après une transaction avec le serveur.

```
int                client_send( SOCKET s , ... , char *command , char *answer );
```

Cette fonction envoie une requête au serveur et collecte la réponse dans un buffer dédié. Elle retourne 0 – constante `FONCTION_SUCCEED` – en cas de succès ou un code d'erreur cliente – défini dans le fichier `/libs/shared/errors.h` – en cas d'erreur.

```
boolean           client_treatment( char *command , char *answer );
```

Cette fonction traite une commande reçue du serveur et génère au besoin une réponse qu'elle place dans un buffer. La fonction retourne `TRUE` en cas de poursuite de la transaction et `FALSE` en cas d'interruption – par exemple, dans le cas d'une commande `OK` ou `ERROR`.

```
void              ct_server_error( char *command );  
boolean          ct_say( char *command );  
boolean          ct_shutdown( char *command );
```

Ces différentes fonctions correspondent à autant de traitements pouvant être exécutés par le client. Dans certains cas, la fonction retourne le succès ou l'échec du traitement par le biais d'un entier booléen – pour renvoi ultérieur d'un message d'erreur.

```
void             client_error( int i , HWND wp );
```

Cette fonction traite une erreur client en retournant un message à l'écran par le biais d'une fonction de type `MessageBox()`.

```
void             client_catched_error( int i );
```

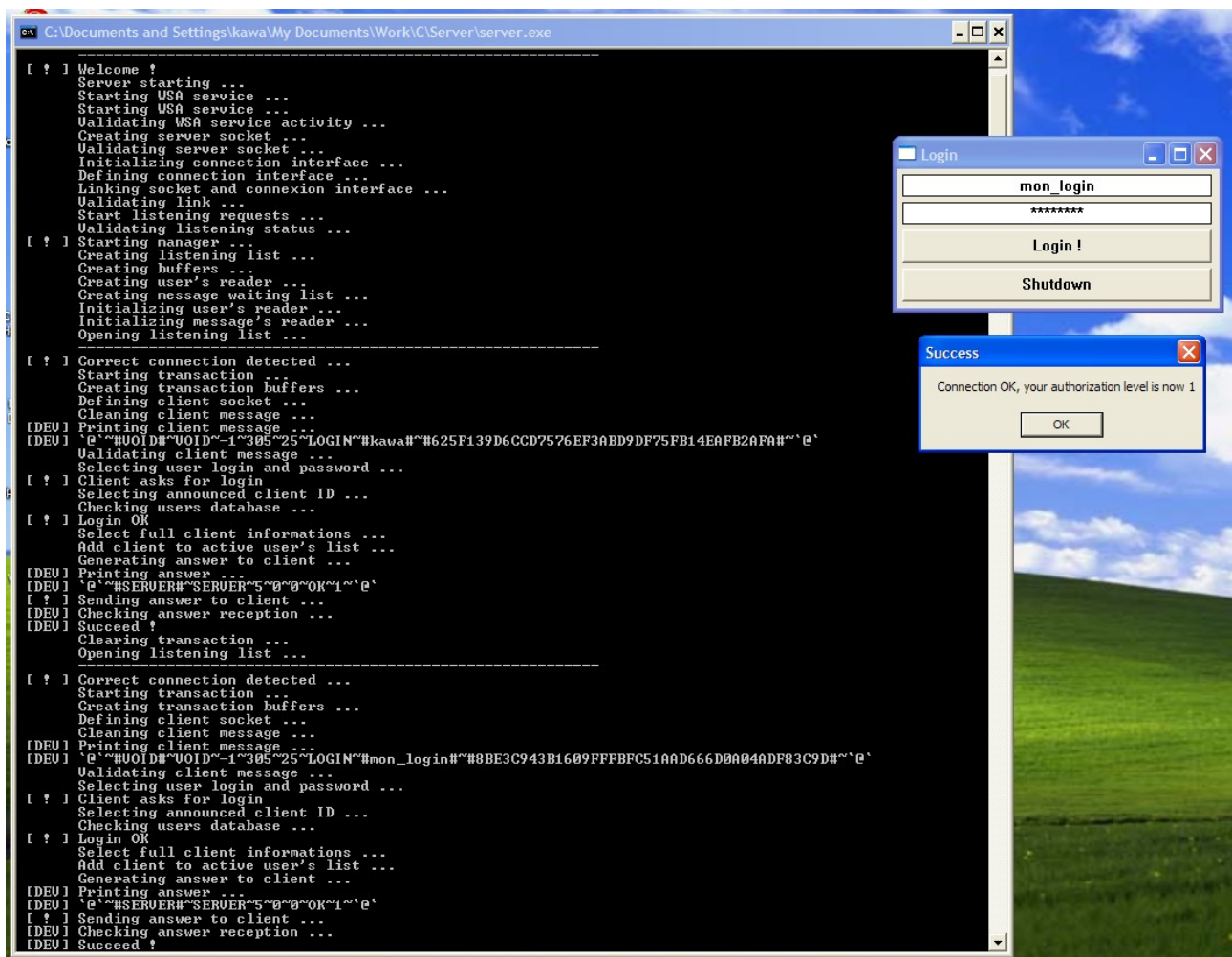
Cette fonction traite une erreur serveur – reçue par le biais d'une requête de type `ERROR` – par le biais d'un retour console – invisible – ou autre mesure.

# EXTENSIONS

Développement

## LOGIN

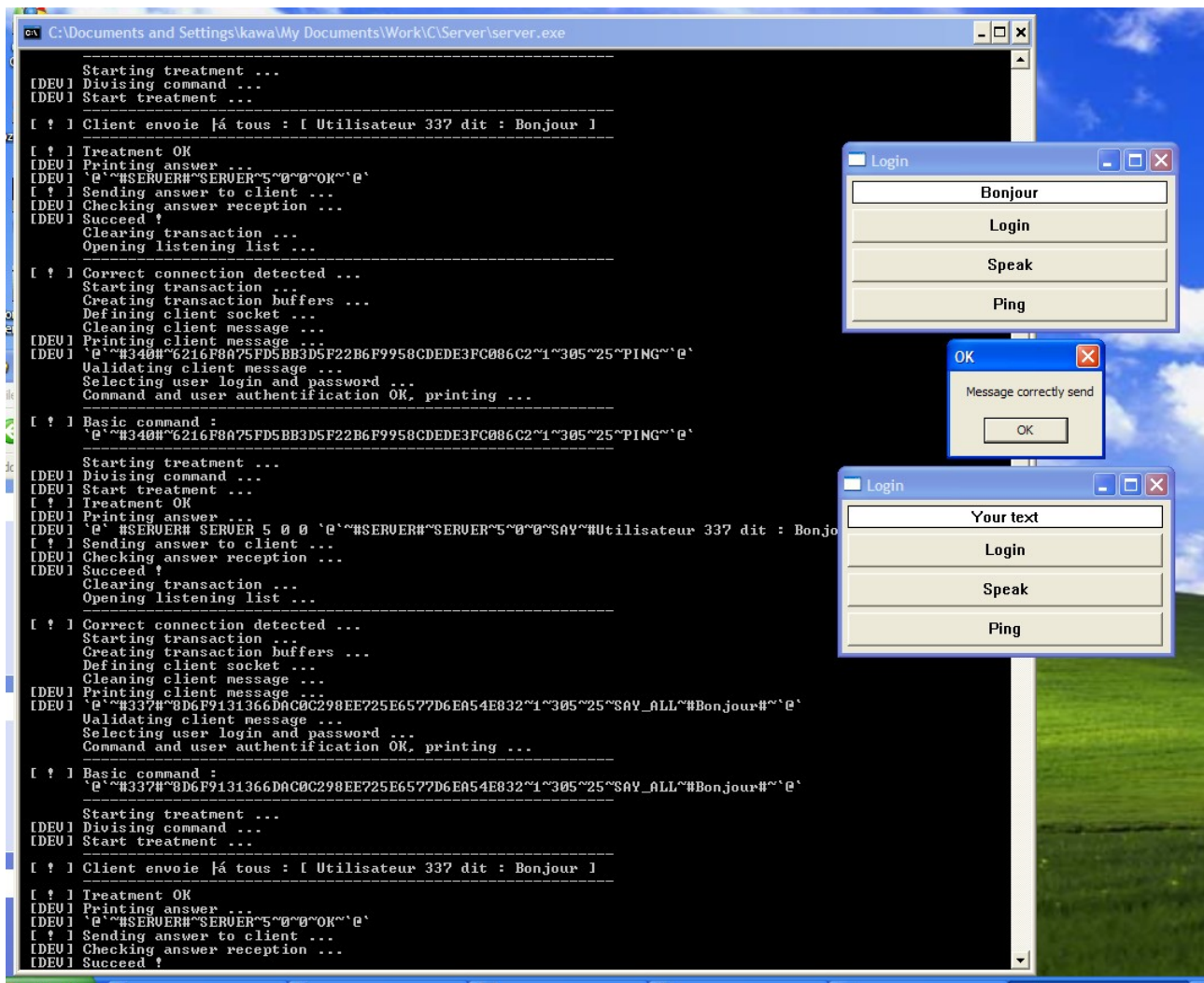
L'application « Login » permet à un client de s'identifier auprès du serveur par le biais d'une commande de type LOGIN. Destinée à bloquer l'accès aux machines clientes, l'application doit être installée en remplacement du formulaire par défaut de Windows.



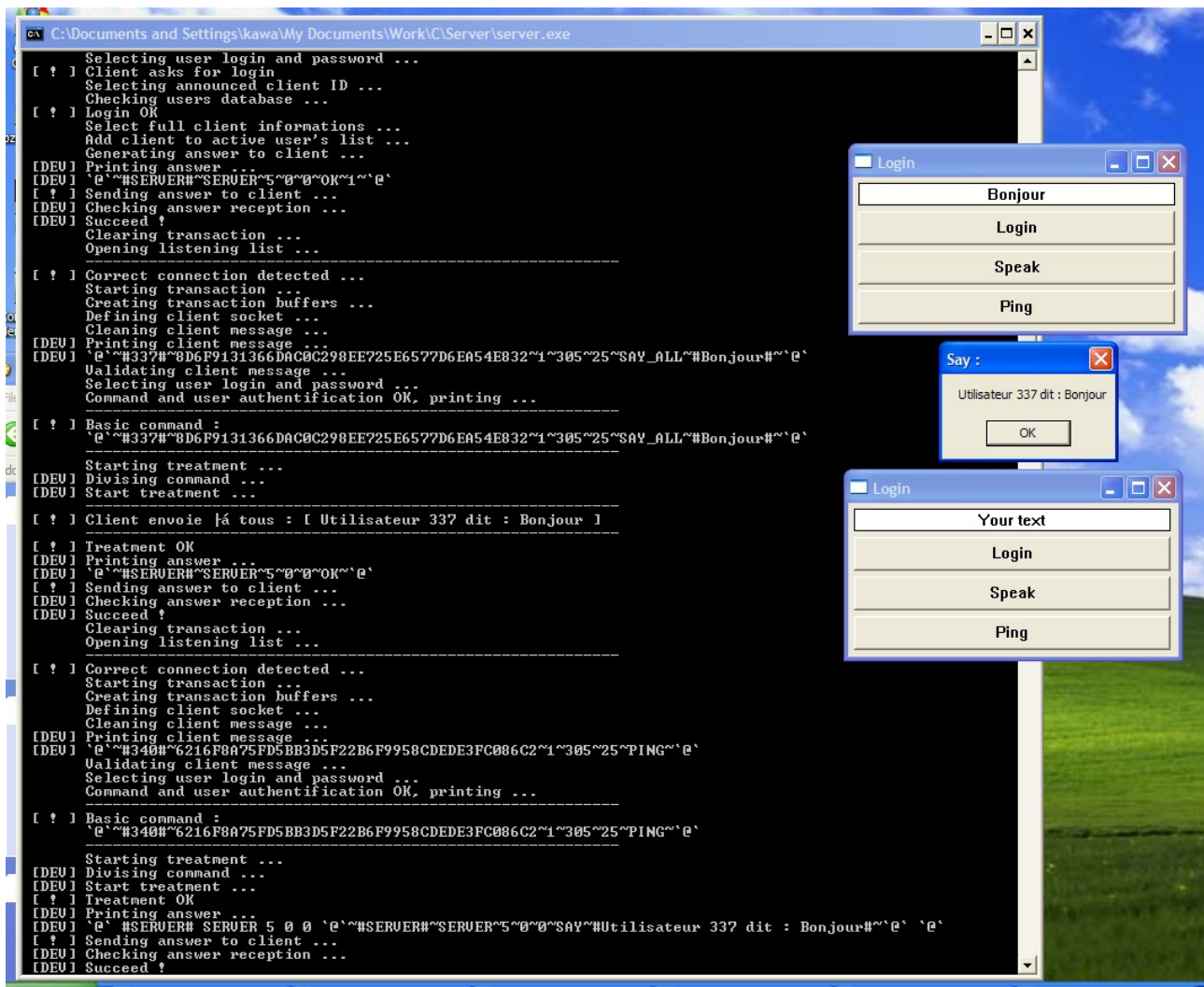
Application « Login » et serveur local

## SAY-PING

L'application « Say-Ping » permet à des clients multiples d'échanger des messages texte en différé. Le client commence par se logger – au moyen, le cas échéant, d'un bouton prévu à cet effet – puis envoie un texte qui sera reçu par l'ensemble des clients connectés au serveur.



Application « Say-Ping » et serveur local  
Envoi du message « Bonjour » par le client 1 (en haut)



Application « Say-Ping » et serveur local  
Réception du message par le client 2 (en bas)

On remarquera sur cet exemple l'apparition d'éléments étrangers dans la commande reçue à la quatrième ligne en partant du bas – traités par le biais de la fonction `command_clear()`.

# BILAN

*Conclusion*

## PROJET

---

Le projet a été rendu en temps et en heure au tuteur de stage, sous la forme d'un CD contenant l'ensemble des sources ainsi que des versions compilées du serveur et des deux extensions réalisées.

De façon générale, je m'estime satisfait du travail réalisé, qui correspondait visiblement aux attentes de l'entreprise, et sur lequel j'ai porté un réel effort de qualité autant pour la fiabilité que pour les relectures ultérieures du code. Toutefois, je pense qu'une ou deux semaines de développement supplémentaires eussent pu se révéler utiles, afin d'achever les différents points listés ci-dessous.

- L'installation de l'application « Login » au démarrage des machines concernée en lieu et place de l'API automatiquement lancée par Windows.
- La création d'un réel fichier de configuration pour simplifier les portages ultérieurs, notamment à l'installation du serveur.
- La poursuite des protocoles de test sur le serveur, notamment dans des cas de surcharge.
- Et enfin, la prise en charge des erreurs de façon graphique dans le modèle client, encore trop approximative à mon goût.

J'ai tiré de ce projet un lot de connaissances extrêmement utiles, dont un aperçu précis de la programmation brute de sockets en C – finalement peu utilisée dans les applications finales –, mais également et surtout en développement d'interfaces utilisateurs graphiques sous Windows. Par ailleurs, je pense avoir affiné mon niveau en C, via l'utilisation de bibliothèques, l'usage de prototypes, et une première mise en place de listes doublement chaînées – étudiées au courant de l'année passée. En somme, je garde un souvenir très positif de ce projet, qui m'a apporté une réelle expérience en programmation locale – le reste de mes acquis en milieu professionnel s'axant uniquement sur le Web.

## STAGE

---

Si les deux premières semaines dans l'entreprise se sont révélées difficiles, autant par l'ampleur technique du projet – au vu de mes connaissances propres – que par les problèmes causés par la barrière linguistique, je pense avoir fait sur un plan un excellent séjour, enrichissant aussi bien sur le plan personnel que professionnel.

L'Asie du sud-est – et tout particulièrement la Chine – est appelée à devenir l'un des acteurs économiques majeurs des années à venir, si ce n'est le premier. Possédant une culture radicalement différente des standards anglo-saxons et ne succombant que de façon très restreinte à la mondialisation, l'entreprise asiatique utilise ses propres repères : le rapport hiérarchique, la vision de l'argent, voir la notion même de travail n'y souffrent guère des fortes connotations ancrées dans le monde occidental. L'emploi ou l'action d'entreprendre ne sont pas vus comme des contraintes mais, au contraire, en temps qu'opportunités à saisir ; leur économie connaît de fait un véritable dynamisme, axé non pas sur des moyens techniques et financiers importants, mais sur

une volonté visible de la populations à prendre des risques et augmenter son niveau de vie. Si le contexte économique peut expliquer en partie ces différences, je pense que le fond de culture joue également un rôle important, que ce soit au niveau religieux, politique ou idéologique.

Néanmoins, il paraît néanmoins difficile de résumer l'ensemble du séjour en quelques lignes. Conscient d'avoir bénéficié d'une opportunité rare, j'en garde un souvenir très positif et n'exclue pas la possibilité de retourner travailler sur place au cours des prochaines années.

# ABSTRACT

*Training at the Stardragon Barcode Science & Technology Co Ltd.  
at Kunming (People republic of China)  
from May to July 2009, ten weeks period*

*During ten weeks, I worked in a small enterprise specialized in software and hardware conception for security and access control systems. This training closed my third year of university degree in IT, computing engineering and business economy.*

*During this period, I had to design a client-server communication protocol and develop a couple of software using it. I had to realize Microsoft OS compatible code in C language, offering a final user interface, and easily editable by other programmers. I used to code on Linux, so I need some adaptation time, and learn how to use the library winsocks2.h and window.h implemented by default in the Windows package – and documented by MSDN. After this step, I conceived a strong, reliable and simple communication protocol, and used it for sending and decode commands through software using UDP sockets. Finally, I created a model of client application, easily exploitable by other developers, and two final application to demonstrate my system.*

*I keep from this training a real professional and personal experience, in a completely different culture of enterprise and in a country becoming one of the most powerful economies in the world. This work allowed me to increase my skills in local development and to discover a new approach of the IT conception business.*