Travail pratique #04

Objectifs

— Expérimenter avec la communication de deux programmes par le réseau
— Utiliser le protocole UDP
— Utiliser une API de programmation bas-niveau (winsock)
— Traitement de commande à distance
— Utilisation d'appels systèmes pour aller chercher des informations utiles
— Utiliser les pipe pour récupérer la sortie console d'un programme

Objectifs facultatifs

Pour ceux qui auront fait le bonus :
— Se familiariser avec l'utilisation d'un Makefile sous linux

Règles importantes

1. Chaque classe doit avoir son propre fichier *.h* et *.cpp*
2. N'oubliez pas les *"include guard"*
3. Prenez grand soin de respecter la casse pour les noms de classe, méthode, etc
4. Veuillez commenter votre code *intelligemment*
5. Votre code doit être de qualité exemplaire
6. **Votre code doit compiler sans erreurs pour être corrigé**

Instructions

(SVP lire l'ensemble de ce document avant de commencer le travail)

Vous devez implémenter une **application serveur** qui écoute sur un port UDP et répond aux demandes qui lui sont faite. Le serveur doit pouvoir répondre à un client (qui est fourni) peu importe qu'il fasse ses demandes du même ordinateur (127.0.0.1) ou d'un ordinateur distant relié par un réseau IP. Vous pouvez vous inspirer de la démonstration faite en classe.

Un client très rudimentaire est fourni avec l'énoncé de ce travail pour que vous puissiez tester votre serveur, il est fortement suggéré de vous en créer un vous-même pour tester votre programme (et surtout tester les cas limites). Vous ne devez pas me le remettre.

Votre application serveur doit écouter sur le port **6666**. Chaque commande reçue est en minuscule, **est obligatoirement suivit d'un espace** et peut optionnellement être suivit par d'autre texte faisant office de paramètres.

Votre serveur doit être capable de recevoir plus d'une commande successive sans avoir à le redémarrer chaque fois.

Voici une description des commandes que votre serveur doit gérer, et pour chacune d'elle un exemple de réponse (attention ! il est possible que certaines réponses soient différentes dépendamment de votre environnement) :

Commande ping

```
Commande du client: ping
Réponse du serveur: pong
```

Commande echo

Le serveur renvoie exactement ce qui suis la commande echo.

```
Commande du client: echo Voici un message
Réponse du serveur: Voici un message
```

Commande date

Retourne la date courante sous forme de chaine de caractere (le format dépend de la langue de votre windows). Pour y arriver vous pouvez utiliser les fonctions *time* et *ctime* définies dans *time.h*.

```
Commande du client: date
Réponse du serveur: Mon Apr 15 23:50:47 2013
```

Commande usager

Cette commande retourne le nom d'usager courant (API à utiliser sous Windows : GetUserName)

```
Commande du client: usager
Réponse du serveur: aouellet
```

Commande exec

Cette commande fait en sorte que le serveur exécute la commande qui suit le mot clef exec. Le serveur exécute la commande et retourne la sortie console (les 10 premières lignes, jusqu'à concurrence de 300 octets maximum). Si la commande n'affiche rien en console, vous devez retourner "OK" au client.

Hint : popen, pclose, fread

```
Commande du client: exec ver
Réponse du serveur: Microsoft Windows [Version 10.0.10586]
```

Commande bye

Cette commande ne retourne rien, mais ferme la connection proprement et provoque la fermeture de votre serveur (*exit*).

```
Commande du client: bye
Réponse du serveur: (aucune)
```

Attention de respecter les majuscules/minuscules.

Votre programme doit être robuste et être capable de gérer les situations imprévues qui peuvent survenir (commande vide, commande trop longue, commande inconnue, etc). Lorsque ce genre d'erreur survient, vous devez uniquement retourner au client le mot ERREUR en majuscule, et continuer à accepter d'autres commandes.

Fonctionnement sous Linux

L'utilisation de sockets sous Linux et sous Windows est très semblable, mais il y a quand même certains détails à faire attention.

Vous devez rendre votre code portable (en utilisant des *#ifdef/#endif* au besoin et/ou d'autres astuces pertinentes) pour qu'il **compile et fonctionne sans modifications** sous Linux aussi (en le compilant avec g++ (et non mingw) pour générer un exécutable natif).

Vous devez vous renseigner et chercher l'information nécessaire par vous-même pour réaliser cette section. MSDN et les *man pages* de linux vous seront utiles. Assurez-vous de tester votre programme entre les 2 systèmes d'exploitation (serveur sous linux et client sous windows par exemple).

Bonus

Cette section est facultative mais donne la possibilité de recevoir des points bonus si elle est terminée en temps et avec succès.

Recherchez de l'information sur la création et l'utilisation d'un Makefile sous linux. Vous devez créer un fichier Makefile qui permettra de compiler votre application en respectant les dépendances d'include.

Il doit être possible de faire les commandes suivantes :

make : compile votre programme en respectant les dépendances, en **compilant toujours le minimum** selon les fichiers sources qui ont été modifiés (ou non). Par exemple, l'appel de la commande *make* deux fois successivement doit provoquer la compilation la première fois, et ne rien faire la seconde fois puisque les fichiers sources n'ont pas changé entre les 2 appels.

make clean : nettoie votre répertoire, en supprimant si applicable les fichiers objets intermédiaires, ainsi que votre fichier exécutable si il existe.

Veuillez placer votre fichier Makefile dans le même répertoire que vos fichiers de code source.

Remise

Vous devez me remettre sur vortex votre solution *visual studio* complète **une fois la solution nettoyée** :

1. Menu Générer, Nettoyer la solution
2. Effacer manuellement les autres fichiers inutiles (ne pas oublier le répertoire .vs)