Labo 2

Serveur itératif sans connexion

© 2008 HEIG-VD, IICT

**Auteur**: J. Ehrensberger
Version 1.0, Mai 2008

Objectifs du laboratoire

Ce laboratoire est le deuxième d’une série de manipulations dans lesquelles nous allons développer une application de chat client – serveur.

Lors du premier labo nous avons développé un serveur itératif avec connexion pour une application de chat. Le défaut principal de ce serveur pour une application de Chat est qu’il ne gère qu’un client à la fois. Il ne permet donc pas de réaliser une communication entre plusieurs clients.

Dans ce deuxième labo nous allons réaliser un serveur de chat sans connexion. L’avantage de ce serveur est qu’il permet facilement de gérer plusieurs clients. Or, la fiabilité de la communication n’est pas garantie et l’application doit prendre en charge la gestion des communications, contrairement à un serveur avec TCP.

Après cette manipulation vous saurez :

1. réaliser un serveur simple en mode itératif sans connexion
2. géréer (copier, comparer, …) la structure d’adresse sockaddr\_in.

# Introduction

Le serveur développé lors de la manipulation précédente ne permet l’échange de message entre plusieurs clients, comme il ne gère qu’une connexion à la fois.

Afin de réaliser un serveur multi-client, nous allons modifier le serveur précédemment développé pour utiliser UDP à la place de TCP. Nous allons donc développer un serveur itératif sans connexion.

Pour effectuer ce laboratoire, vous avez besoin de :

1. un ordinateur avec Linux, Mac OSX ou BSD
2. un éditeur de programme à votre choix
3. un compilateur C/C++ avec éditeurs de liens
4. l’outil make
5. les pages man pour le développement
6. l’outil netcat pour effectuer des tests

# Serveur écho itératif sans connexion

Dans une première étape nous allons modifier le serveur TCP développé pour qu’il utilise UDP à la place de TCP. Le serveur permettra la communication parallèle avec plusieurs clients mais ne fait qu’envoyer un message reçu au client. Il s’agit donc d’un serveur Echo et non pas de Chat.

Tâche à effectuer

1. Faire une copie des fichiers tcpserver.cpp et tcpserver.h et les renommer udpserver.cpp et udpserver.h. Ne pas oublier de modifier aussi les références dans les fichiers.

A inclure dans le rapport

* Faites un tableau comme le suivant qui indique les fonctions sockets utilisées par le serveur TCP. Décrivez les modifications à faire pour convertir le serveur TCP en un serveur écho UDP.

|  |  |
| --- | --- |
| **Serveur écho TCP** | **Modification pour serveur écho UDP** |
| Fonction socket |  |
| Fonction bind |  |
| Fonction listen |  |
| Fonction accept |  |
| Fonction read |  |
| Fonction write |  |

Tâches à faire

1. Effectuer les modifications nécessaires.
2. Compiler le serveur et le démarrer.
3. Tester le serveur UDP avec netcat en mode UDP :

nc -u localhost 1234

Le serveur doit correctement renvoyer les messages reçus.

1. Tester le serveur avec plusieurs clients : simuler plusieurs clients en utilisant plusieurs commande *nc* en parallèle. Est-ce que le serveur permet plusieurs communications en parallèle ?

A inclure dans le rapport

1. Code source complet du serveur (\*.cpp et \*.h)
2. Protocole des tests effectués
3. Interprétation du comportement du serveur lors de la communication avec plusieurs clients.

# Serveur chat multi-client

Pour réaliser l’application de Chat, le serveur doit envoyer un message reçu à tous les clients actuels. Le serveur doit donc gérer une liste des clients connectés. Le problème principal qui se pose est la déconnexion d’un client. Contrairement à TCP le protocole UDP ne permet pas de détecter si un client s’est déconnecté. C’est donc l’application qui doit gérer ceci.

## Protocole d’application

Nous définissions le protocole d’application suivant :

1. **Connexion d’un client**
	* Le premier message envoyé par un client au serveur contient uniquement le nom d’utilisateur du client (p.ex. « *Jean*»).
	* Le serveur enregistre le client avec son adresse IP, port source et son nom d’utilisateur dans une liste des clients.
	* Le serveur envoie un message « *\*\*\* New user: Jean* » à tous les clients actuels, y compris le nouveau client comme confirmation.
2. **Chat**
	* Lorsque le serveur reçoit un message d’un client, il cherche la source du message dans la liste des clients.
	* S’il trouve la source dans la liste des clients, il préfixe le message avec le nom d’utilisateur de la source et le transmet à tous les clients, y compris la source.
	* S’il ne trouve pas le client, il s’agit d’une nouvelle connexion (voir ci-dessus).
3. **Déconnexion**
	* Un client peut se déconnecter en envoyant le message « *\*\*\* STOP \*\*\** » au serveur.
	* Un message du type « \*\*\* *Disconnected user: Jean* » sera envoyé à tous les clients, y compris le client déconnecté, comme confirmation. Ensuite le client sera effacé de la liste des clients.

## Réalisation

Un serveur réel utiliserait une liste chaînée pour la liste des clients. Pour garder la réalisation simple nous allons utiliser un tableau avec une longueur fixe, avec p.ex. 20 places.

Vous aurez probablement besoin des fonctions Unix suivantes dans votre programme :

|  |  |
| --- | --- |
| memset(void \*b, int c, size n) | Initialiser n octets à partir de b à la valeur c.  |
| memcpy(void \*dst, void \*src, int n) | Copier n octets de la plage mémoire src dans dst. |
| memcmp(void \*s1, void \*s2, int n) | Comparer n octets des plages mémoires s1 et s2. Retourne 0 si s1 == s2.  |
| strncpy(char \*dst, char \*src, int n) | Copier n octets du string src dans le string dst.  |
| strncat(char \*dst, char \*src, int n) | Concaténer au maximum n caractères de src au string dst.  |
| int strlen(char \*s) | Retourne la longueur de la chaîne de caractères s. Ne compte pas le caractère terminal ‘\0’.  |
| char \*strpbrk(char \*s, char \*list) | Trouve la première occurrence d’un des caractères de list dans la chaîne de caractères s. Utile p.ex. pour trouver les fins de lignes.  |

Tâches à faire

1. Dans le fichier *udpserver.h* définir une nouvelle structure *clientDesc* qui décrit un client connecté.

*struct clientDesc {*

 *struct sockaddr\_in clientAddress ;*

 *char username[10] ;*

*} ;*

1. Dans le fichier udpserver.h définir un tableau clientList comme variable de classe. Il contient jusqu’à 20 éléments du type clientDesc. Une variable de classe qui indique l’indice du dernier élément valable du tableau sera aussi nécessaire.

ATTENTION : ne pas oublier d’initialiser cette variable, p.ex. dans la méthode Listen du serveur.

1. Définir une méthode SendToAll qui prend comme argument une chaîne de caractères et qui l’envoie à tous les clients enregistrés dans clientList. La fonction ne retourne pas de valeur.

*void SendToAll(char \*msg)*

1. Définir une méthode FindClient qui prend comme argument le pointeur vers une structure du type sockaddr\_in et qui renvoie l’indice du client dans la liste des client ou -1 si le client n’est pas dans la liste.

*int FindClient(sockaddr\_in \*clientAddr)*

1. Définir une méthode AddClient qui prend comme argument le pointeur vers une structure sockaddr\_in et une chaîne de caractères (nom d’utilisateur). Elle ajoute le client dans le tableau clientList et retourne l’indice du client dans la liste si l’opération s’est déroulée avec succès et -1 autrement.

ATTENTION : vous devez copier le contenu des arguments clientAddr et username.

*int AddClient(sockaddr\_in \*clientAddr, char \*username)*

1. Définir une méthode DelClient qui prend comme argument le pointeur vers une structure sockaddr\_in. La méthode efface le client de la liste et retourne TRUE ou FALSE.

ATTENTION : N’oubliez pas de déplacer le dernier élément de la liste dans le nouveau trou et de mettre à jour l’indice du dernier élément de clientList.

*bool DelClient(sockaddr\_in \*clientAddr)*

1. Modifier la méthode MainLoop du serveur pour réaliser le protocole décrit dans la section 3.1. Principalement, le serveur lit un message du socket et regarde s’il s’agit d’un nouveau client ou d’une déconnexion ou d’un message chat. En fonction du résultat il envoie un message à tous les clients connectés.
2. Compiler et exécuter le serveur.

Tests à effectuer

* Tester le comportement correct du serveur :
	+ Connexion d’un nouveau client
	+ Chat entre plusieurs clients
	+ Déconnexion d’un client

A inclure dans le rapport

* Code complet du serveur
* Protocole de tests effectués.

# Fiabilité du serveur

Bien que le serveur développé dans les sections précédentes fonctionne assez bien, il souffre de plusieurs défauts :

* La gestion des clients est fastidieuse. Le serveur doit enregistrer leurs adresses bien qu’il ne s’y intéresse pas. Des connexions TCP permettraient de simplement avoir plusieurs sockets connectés, sans gérer les adresses associées.
* Le serveur doit implémenter un protocole d’application pour gérer la connexion et déconnexion de clients. En mode connecté, TCP gère la connexion et déconnexion.
* Le serveur dépend du comportement correct des clients. Un client qui ne se déconnecte pas correctement restera dans la liste des clients. Un client qui se plante fréquemment risque de remplir la liste des clients du serveur.

Le test suivant illustre le dernier point.

Tâches à faire

* Démarrer le serveur
* Démarrer Wireshark sur l’interface de rebouclage.
* Se connecter avec un client 1 et s’enregistrer avec un nom d’utilisateur.
* Se connecter avec un client 2 au serveur et s’enregistrer avec un nom d’utilisateur.
* Arrêter le client 2 avec Ctrl-C.
* Envoyer un message depuis le client 1.

A inclure dans le rapport

* Capture de Wireshark
* Interprétation de la capture de Wireshark