Labo 1

Serveur itératif avec connexion

© 2008 HEIG-VD, IICT

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

Objectifs du laboratoire

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

La première étape est le développement d’un serveur simple. Le serveur fonctionne en mode connecté, donc avec le protocole TCP. Il effectue un traitement itératif : seulement après la fin de la connexion en cours il peut accepteur la prochaine connexion.

Le traitement effectué par le serveur est simple. Le serveur reçoit des messages venant du client et renvoie le même message comme echo.

Après cette manipulation vous saurez :

* utiliser les fonction socket, bind, listen, accept, read, write
* utiliser les sockets passifs
* développer un serveur simple en mode itératif avec connexion.

# Introduction

Pour effectuer ce laboratoire, vous avez besoin de :

* un ordinateur avec Linux
* un éditeur de programme à votre choix
* un compilateur C/C++ avec éditeurs de liens
* l’outil make
* les pages man pour le développement
* l’outil netcat pour effectuer des tests

Dans Ubuntu, vous pouvez installer les paquetages suivants :

* gcc
* g++
* binutils
* make
* manpages-dev
* netcat

Vous pouvez les installer avec la commande :

sudo apt-get install gcc g++ binutils make manpages-dev netcat

Vous aurez également besoin d’un éditeur de programme comme kate ou vim. Il n’est pas nécessaire d’installer Eclipse ou NetBeans pour nos programmes relativement courts.

# Fichiers du programme

Le programme à développer est programmé en C++. Il utilise les fichiers suivants :

* fichier tcpserver.h : fichier header avec la déclaration de la classe ChatServer.
* fichier tcpserver.cpp : fichier avec l’implémentation de la classe.

Vous pouvez télécharger les modèles des deux fichiers depuis le site Web du cours.

/////////////////////////////////////////////////////////////////

// File: tcpserver.h

// Author: Juergen Ehrensberger, HEIG-VD

// Created: 28.03.2008

// Copyright: (c) 2008 HEIG-VD, all rights reserved

//

// Course PSE - Chapter Network Programming

//

// Implemention of the chat server over TCP connections

/////////////////////////////////////////////////////////////////

#ifndef \_\_tcpserver\_H\_\_

#define \_\_tcpserver\_H\_\_

#define MAX\_MSG\_LEN 500

class ChatServer {

 public:

 bool Listen(unsigned int localPort);

 bool Accept();

 bool MainLoop();

 bool Close();

 private: // Internal methods

 private: // Data members

 int activeSock;

 int listenSock;

};

#endif

Ce fichier header définit plusieurs méthodes :

* Listen : crée un socket, le met en mode passive et l’attache au port reçu comme argument.
* Accept : attend l’arrive d’une nouvelle connexion et l’accepte.
* MainLoop : lit des messages du socket et les renvoie au client.
* Close : ferme le socket établit (non pas le socket passif).

Nous n’aurons pas besoin de modifier ce fichier header. La réalisation sera faite dans le fichier tcpserver.cpp, montré ci-dessous.

/////////////////////////////////////////////////////////////////

// File: tcpserver.cpp

// Author: Juergen Ehrensberger, HEIG-VD

// Created: 28.03.2008

// Copyright: (c) 2008 HEIG-VD, all rights reserved

//

// Course PSE - Chapter Network Programming

//

// Implemention of the chat server over TCP connections.

/////////////////////////////////////////////////////////////////

#include <iostream>

#include <errno.h>

#include <sys/socket.h>

#include <arpa/inet.h>

#include <resolv.h>

#include "tcpserver.h"

using namespace std;

//---------------------------------------------------------------

// Listen

// This method creates a listening socket

//---------------------------------------------------------------

bool **ChatServer::Listen**(unsigned int localPort) {

 struct sockaddr\_in localAddr; // Local address of socket

 // 1. Create socket as data member ChatServer::listenSock

 // Todo: create socket of type STREAM

 // Todo: test if socket creation has been successful

 // if not: print error message and return false

 // 2. Bind socket to address INADDR\_ANY and the localPort

 // Todo: initialize localAddr with zeros.

 // Todo: specify address family

 // Todo: specify address INADDR\_ANY

 // Todo: specify localPort as port (don't forget conversion)

 // Todo: call bind function

 // Todo: test if call was successful.
 // If not: print error message and return false

 // 3. Call listen on the socket with a queue for 20 conn.

 // Todo: put socket into passive mode with listen.

 // Todo: test is call was successful.

 // If not, print error and return false

 cout << "Server: Listen succeeded!\n";

 return true;

}

//---------------------------------------------------------------

// Accept

// This method calls accept to wait for incoming connections.

// Aften accept, it closes the listening socket and returns.

//---------------------------------------------------------------

bool ChatServer::Accept() {

 struct sockaddr\_in clientAddr; // Address of client

 socklen\_t clientAddrLen;

 // 1. Accept incoming connections and store the accepted

 // socket as data member ChatServer::activeSock

 // Todo: call accept for listenSock

 // Todo: test if accept was successful.

 // If not, print error and return false

 cout << "Server: Connection accepted\n";

 return true;

}

//---------------------------------------------------------------

// MainLoop

// Enter main loop: read message from activeSock and echo it back

//---------------------------------------------------------------

bool ChatServer::MainLoop() {

 ssize\_t msgLen = 0;

 char msg[MAX\_MSG\_LEN];

 // Loop forever

 while(1) {

 // 1. Read a message from the socket

 // Todo: initialize msg with zeros

 // Todo: read on activeSock

 // Todo: test if read was successful.

 // If not print error and return false

 // 2. If 0 octets received, call ChatServer::Close

 // Todo: test if End-of-file has been received.

 // Todo: if yes: call ChatServer::Close.

 // Todo: return true

 // 3. Sent message back on socket

 // Todo: write message back to socket

 // Todo: test if write was successful.

 // If not, print error and return false

 // Todo: print received message

 }

 return true;

}

//---------------------------------------------------------------

// Close

// Close socket

//---------------------------------------------------------------

bool ChatServer::Close() {

 // Close socket

 // Don't forget to test if result is OK

 // Todo: call close

 // Todo: test if call was successful.

 // If not, error and return false

 return true;

}

//---------------------------------------------------------------

// Main function

// Create a chat server, call listen, accept and then mainloop.

// Usage:

// ./chatserver <localport>

//---------------------------------------------------------------

main(int argc, char \*argv[]){

 int localport = 0;

 // Check if there is 1 arg and if it is a port number>=1024.

 if (argc != 2) {

 printf("Usage: ./chatserver <localport>\n");

 exit(-1);

 }

 localport = (unsigned int) strtoul(argv[1],(char\*\*)NULL,10);

 if (localport < 1024 or localport > 65535) {

 printf("Error: port must be between 1024 and 65535\n");

 exit(-1);

 }

 // Create server object, call listen, accept, mainloop

 ChatServer chatserver = ChatServer();

 if (not chatserver.Listen(localport))

 exit(-1);

 if (not chatserver.Accept())

 exit(-1);

 chatserver.MainLoop();

 cout << "Server shutdown\n";

}

Regardons d’abord la fonction main à la fin du fichier. Quand le programme démarre il lit un argument de la ligne de commande et l’interprète comme numéro de port local sur lequel le serveur doit écouter. Il stocke ce numéro de port dans la variable localport. Ensuite le programme crée un objet du type ChatServer qui réalise la fonctionnalité du serveur.

Puis le programme appelle la méthode Listen du serveur qui crée un socket, le lie au port local et le met en mode passif.

Ensuite la méthode Accept est appelée qui attend une nouvelle connexion.

Quand cette méthode retourne, le programme appelle la méthode MainLoop du serveur est appelé qui lit des données du socket et les renvoie simplement. La boucle fait ceci jusqu’à ce que le client libère la connexion.

# Manipulations

Nous allons maintenant réaliser les différentes fonctionnalités du serveur en modifiant le fichier tcpserver.cpp. Tous les commentaires qui commencent par

// Todo:

doivent être remplacés par le code C++ correspondant.

## Méthode ChatServer::Listen

Manipulations à faire :

* Modifier la méthode ChatServer::Listen comme indiqué par les commentaires *Todo*.
* Compiler le programme avec la commande

g++ tcpserver.cpp –o chatserver

Ceci génère un exécutable chatserver

* Exécuter le serveur sur le port local 1234 avec la commande

./chatserver 1234

* Le serveur doit s’exécuter sans erreurs.

## Méthode ChatServer::Accept

Manipulations à faire :

* Modifier la méthode ChatServer::Accept comme indiqué par les commentaires *Todo*.
* Compiler le programme et l’exécuter. Le programme doit se bloquer sur l’appel de la fonction accept.

* Pour tester le serveur, lancer l’outil nc sur l’adresse 127.0.0.1 et le port 1234

nc 127.0.0.1 1234

Nc établit une connexion TCP avec le serveur indiqué par l’adresse IP et le port de destination. Il accepte ensuite des caractères de l’entrée standard et les envoie au serveur. Les données reçues par le serveur sont affichées sur la sortie standard. Nc nous permet donc de facilement tester l’établissement de connexion TCP avec le serveur.

Le serveur doit accepter la connexion et ensuite se terminer sans erreurs. Nc ne doit pas afficher d’erreurs.

## Méthode ChatServer::MainLoop

Manipulations à faire :

* Modifier la méthode ChatServer::MainLoop comme indiqué par les commentaires *Todo*.
* Compiler le programme et l’exécuter.
* Etablir une connexion avec nc. Les caractères de l’entrée standard de nc doivent être reçues correctement par le serveur et l’écho envoyé par le serveur doit être affiché par nc :

Client nc :

~ $ ***nc 127.0.0.1 1234***

Ceci est un message.

Ceci est un message.

Je peux terminer nc avec Ctrl-C

Je peux terminer nc avec Ctrl-C

^C punt!

~ $

Serveur :

~ $ ***./chatserver 1234***

Server: Listen succeeded!

Server: Connection accepted

Server: Received: Ceci est un message.

Server: Received: Je peux terminer nc avec Ctrl-C

Server: EOF received. Closing socket

Server shutdown

~ $

## Méthode ChatServer::Close

Manipulations à faire :

* Modifier la méthode ChatServer::MainLoop comme indiqué par les commentaires *Todo*.
* Compiler le programme et l’exécuter.
* Etablir une connexion avec nc. Libérer la connexion avec Ctrl-C dans nc. Le serveur doit correctement libérer la connexion.

# Rapport

Comme rapport, fournir le code source complet du serveur développé.