**ESIEE Paris**

13

**INF 4201B Informatique Distribuée**

**Rapport de TP : Sockets en UDP / TCP**

**Sébastien LE STUM**

Contenu

[INTRODUCTION 2](#_Toc375253812)

[CONNEXION EN MODE UDP 3](#_Toc375253813)

[EXERCICE 1 3](#_Toc375253814)

[COMMUNICATION EN MODE TCP 6](#_Toc375253815)

[EXERCICE 2 6](#_Toc375253816)

[EXERCICE 3 8](#_Toc375253817)

# INTRODUCTION

L’utilisation des « sockets » comme moyen de communication entre systèmes est une méthode très pratique et très utilisée dans le monde de l’informatique.

Dans le cadre de l’unité, nous nous sommes focalisé principalement sur l’utilisation des sockets sur des systèmes à noyau UNIX. Les sockets sur Windows ont, quant à eux, des méthodes d’initialisation différentes et n’utilisent pas exactement les mêmes structures, bien que le fonctionnement global soit très similaire avec son équivalent UNIX.

Sur un système UNIX, les sockets ont deux modes de fonctionnement :

* AF\_UNIX : Ce mode permet la communication entre deux processus d’un même système en utilisant un fichier tampon dans le système de fichier de la machine hôte. Il s’agit dans l’idée d’un fonctionnement similaire à un tube nommé.
* AF\_INET : Ce mode permet quant à lui la communication entre deux processus de systèmes différents à travers une connexion Ethernet / internet.

Dans ce rapport, nous allons constamment nous trouver dans le mode AF\_INET, qui nous permettra de contacter des machines distantes comme le proxy de l’ESIEE ou même un site extérieur.

Ensuite, l’ensemble de ce rapport concernera l’utilisation des sockets en mode UDP puis en mode TCP.

Le mode UDP représente une communication par blocs, sans aucune connexion persistante entre le client et le serveur. Cela implique qu’aucun contrôle n’est appliqué sur le flux transmis. En échange, ce protocole permet le transport de plus de données et de manière plus rapide.

Inversement, le mode TCP est un mode connecté : En effet, ce mode contient des structures et des éléments de contrôles (CRC, code d’erreur) afin de vérifier l’intégrité du message et de maintenir un lien temporaire entre le client et la machine distante.

Ainsi pour ce rapport, nous allons décrire explicitement chaque cas rencontré en indiquant le code source utilisé, appuyé par des traces d’exécution et conclure quant au fonctionnement des sockets et leurs utilisations pratiques dans un contexte de communication client / serveur.

# CONNEXION EN MODE UDP

## EXERCICE 1

Pour cet exercice simple, nous devions réaliser une communication UDP entre un client et un serveur. Le client doit envoyer son PID au serveur et un message passé en paramètre dans la ligne de commande. Dans un second temps, le serveur reçoit ces informations puis en envoie au client (PID du serveur + le message). Cette communication a pour but d’attester du fonctionnement des sockets en mode UDP et de le mettre en pratique.

Pour exécuter nos deux programmes il suffit :

* Pour le client : Fournir l’adresse du serveur (localhost ou 127.0.0.1), le port utilisé et enfin le message que l’on souhaite envoyer.
* Pour le serveur : Fournir uniquement le port sur lequel le serveur sera en écoute, prêt à recevoir les informations du client.

Pour pouvoir initier la communication entre les deux programmes, il faut tout d’abord initialiser les sockets.

Code « client » et « serveur » :

struct sockaddr\_in localAdd;

struct sockaddr\_in serverAdd;

struct hostent\* h\_s;

int cSocket = socket(AF\_INET, SOCK\_DGRAM, 0);

if(cSocket == -1){

perror("Erreur lors de la création de la socket");

return 0;

}

localAdd.sin\_family = AF\_INET;

localAdd.sin\_port = htons(5600);

localAdd.sin\_addr.s\_addr = INADDR\_ANY;

memset(localAdd.sin\_zero, 0, 8);

result = bind(cSocket, (struct sockaddr\*) &localAdd, sizeof(localAdd));

Le code ci-dessus est commun au serveur comme au client. En effet, il permet d’initialiser le socket et de le « *bind* », ce qui sous-entend de l’attacher au système pour le rendre prêt à être utilisé lors d’une communication.

Les deux structures contiennent les informations du socket « local » et du socket distant que l’on souhaite joindre.

L’argument SOCK\_DGRAM permet de passer le socket en mode Datagramme c’est-à-dire en mode UDP. Il est également créé pour fonctionner en mode réseau grâce à l’argument AF\_INET.

Une fois que le socket est créé et attachée, nous initialisons la structure contenant les informations du socket distant et ainsi commencer l’échange d’informations.

Code « Client » :

/\*Envoi des données au serveur\*/

fprintf(stdout,"Sent :\n");

sendto(cSocket, (void \*) message1, sizeof(message1), 0, (struct sockaddr\*) &serverAdd, sizeof(serverAdd));

sendto(cSocket, (void \*) pid, sizeof(pid), 0, (struct sockaddr\*) &serverAdd, sizeof(serverAdd));

fprintf(stdout,"\tMessage : %s\n",argv[3]);

fprintf(stdout,"\tClient PID : %s\n",pid);

/\*Reception de la réponse\*/

fprintf(stdout,"Recieved :\n");

recvfrom(cSocket, message1, MESSAGE\_SIZE, 0,(struct sockaddr\*) &serverAdd, &size);

recvfrom(cSocket, message2, MESSAGE\_SIZE, 0,(struct sockaddr\*) &serverAdd, &size);

printf("\tMessage : %s\n\tServer PID : %s\n", message1, message2);

Code « Serveur » :

/\*Reception des données du client\*/

fprintf(stdout,"Recieved :\n");

recvfrom(sSocket, message1, MESSAGE\_SIZE, 0,(struct sockaddr\*) &clientAdd, &taille);

recvfrom(sSocket, message2, MESSAGE\_SIZE, 0,(struct sockaddr\*) &clientAdd, &taille);

fprintf(stdout,"\tMessage : %s\n\tClient PID : %s\n", message1, message2);

sprintf(pid,"%d",getpid());

/\*Envoi des informations au client\*/

fprintf(stdout,"Sent :\n");

sprintf(message1,"%s%s",message1," (ACK from server)");

sendto(sSocket, (void \*) message1, sizeof(message1), 0, (struct sockaddr\*) &clientAdd, sizeof(clientAdd));

sendto(sSocket, (void \*) pid, sizeof(pid), 0, (struct sockaddr\*) &clientAdd, sizeof(clientAdd));

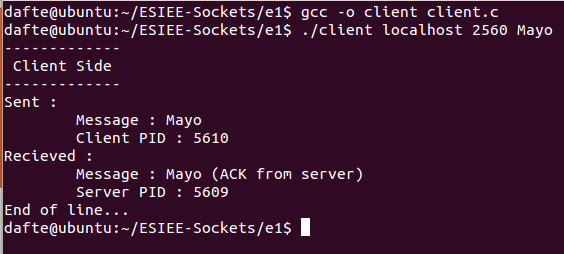
fprintf(stdout,"\tMessage : %s\n",message1);

fprintf(stdout,"\tServer PID : %s\n",pid);

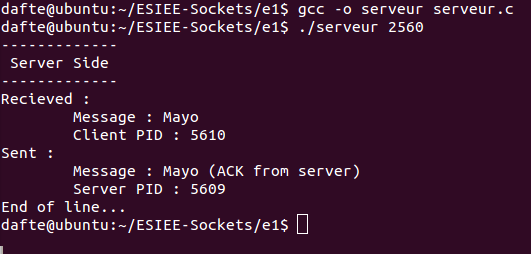
Du point de vue du client, nous envoyons donc le PID obtenu par la fonction getpid() et le message écrit sur la ligne de commande qui a été mis en buffer. Une fois que l’envoi a été effectué, le client se place en écoute sur le port de communication et attend les informations du serveur.

Du point de vue serveur, l’inverse se passe. Le serveur est tout d’abord en attente de recevoir les informations. Ces informations, une fois reçues, sont transférées dans des buffers. Pour prouver que l’échange du message à eu lieu sans aucune altération, nous avons ajouté dans la réponse serveur la chaine de caractère : « ACK from server ». De ce fait, nous pouvons confirmer que les informations envoyées l’ont été sans erreurs et que la réponse reçue par le client est bien celle provenant du serveur.

Ci-dessous se trouve une trace de l’exécution de ces deux programmes, dont le code source complet se trouve en annexe de ce rapport.



*Aperçu de l’exécution du programme « Client »*



*Aperçu de l’exécution du programme « Serveur »*

# COMMUNICATION EN MODE TCP

Durant les exercices suivants, nous initialiserons les sockets utilisés en mode TCP, c’est-à-dire en utilisant l’argument SOCK\_STREAM lors de la création du socket.

## EXERCICE 2

Le principe de l’exercice 2 est d’analysé la requête envoyé par un navigateur. En effet, lors d’une communication entre un navigateur Web et un serveur, le navigateur envoie une requête HTTP. Cette requête contient les informations utiles demandées par l’utilisateur pour accéder à une page stockée sur le serveur.

Si le serveur dispose de cette page dans le bon répertoire, alors il répond en envoyant directement les informations au navigateur sous forme de code HTML. Sinon une réponse automatisée (ou non) est envoyée au navigateur pour signaler l’erreur rencontrée (comme l’erreur 404 signifiant : Page Not Found)

Grâce à notre embryon de serveur, nous pourrons ainsi étudier comment est structurée une requête HTTP.

Pour lancer ce programme nous fournissons en paramètre uniquement le port sur lequel le serveur devra se placer en écoute.

Code « Serveur » :

if(listen(lSocket,1) == -1){

perror("Erreur lors du listen !\n");

return -1;

}

lService = accept(lSocket,(struct sockaddr\*) &clientAdd, &taille);

if(lService==-1){

perror("Erreur lors du accept !\n");

return -1;

}

read(lService, (void\*) requete, sizeof(requete));

fprintf(stdout,"Request from browser : %s", requete);

write(lService, (void\*) reponse, sizeof(reponse));

fprintf(stdout,"Answer from server : %s \n",reponse);

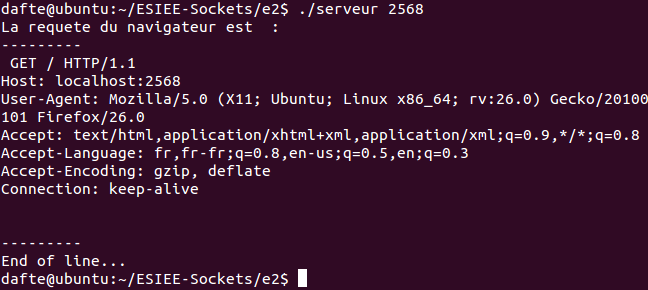
close(lService);

close(lSocket);

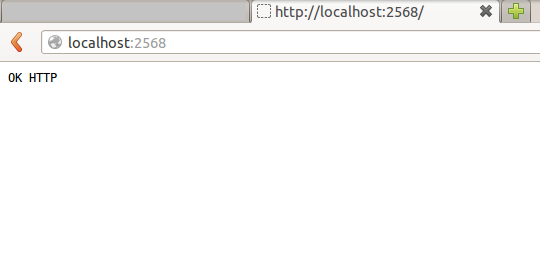
Pour placer le serveur en écoute, nous utilisons la fonction listen(). Une fois qu’un client s’est connecté au port d’écoute, nous utilisons la fonction accept() pour signifier que le serveur accepte la communication. A ce moment, le système effectue un fork() du programme pour pouvoir exécuter la requête indépendamment du programme principal.

Une fois fais, le programme lit la requête via read(), puis envoie une réponse au navigateur pour signifier que tout a fonctionné.

Ci-dessous se trouve une trace de l’exécution du programme et la réponse simple envoyée au navigateur utilisé dans cet exemple (Firefox). Le code source complet du programme se trouve en annexe de ce rapport.



*Aperçu de l’exécution du serveur*

**

*Aperçu de la réponse du serveur au navigateur*

## EXERCICE 3

Pour cet exercice, nous devions réaliser un embryon de navigateur. Pour se faire, nous disposions du squelette de la requête HTTP que nous avions récupéré précédemment. Il suffit ensuite, lors de l’exécution du programme de fournir la page que l’on souhaite atteindre puis le port à utiliser (port 80 pour un accès Internet).

Une fois fait, le client doit se connecter au port que l’on indique et demander grâce à notre requête HTTP la page souhaitée.

Code « client » :

if(connect(lSocket, (struct sockaddr\*) &serverAdd, sizeof(serverAdd)) == -1){

perror("Echec lors de la connexion !");

return -1;

}

sprintf(requete,"%s%s%s%s%s%s%s","GET /",argv[3]," HTTP/1.1\nHost: ",argv[1],":",argv[2],"\nUser-Agent: Mozilla/5.0 (X11; Linux x86\_64; rv:17.0) Gecko/20131029 Firefox/17.0\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8\nAccept-Language: en-US,en;q=0.5\nAccept-Encoding: gzip, deflate\nConnection: Close\n\n");

send(lSocket, (void\*) requete, sizeof(requete),0);

byteCount = recv(lSocket, (void\*) reponse, sizeof(reponse),0);

reponse[byteCount-1] = '\0';

printf("Reponse du serveur : \n%s\n",reponse);

close(lSocket);

Nous nous connectons dont au site voulu via notre socket. Ensuite nous écrivons notre requête en fonction des arguments fournis lors de l’exécution du programme. Enfin les données sont envoyées et le client attend la réponse du serveur à sa demande.

Pour éviter tout erreur de lecture du résultat nous plaçons un ‘\0’ à la fin des données reçues en réponse, ce symbole représentant la fin d’une chaine de caractère.

Ci-dessous se trouve la trace d’exécution de ce programme. Veuillez noter que le code source complet est disponible en annexe.