iUT ORSAY
Département Informatique

DUT Informatique
Module Système S4 C
2009 / 2010

Travaux Pratiques n° 3 : Sockets Datagramme

Nom(s) :
Groupe :
Date :

Objectifs : manipuler les primitives relatives à la communication par sockets datagramme, être capable d'implanter des systèmes client-serveur sur ce mode de communication.

1 Avant propos

Les rappels nécessaires à la réalisation de ce TP sont fournis en notes de cours au début du TD n° 2. Bien sûr les pages de `man` sont comme toujours de précieuses alliées.

2 Commande `runame` : remote uname

`uname` est une primitive classique des unices permettant d'obtenir des informations sur le système où on exécute cette commande (faire `man uname` pour plus de détails). À l'IUT d'Orsay, de nombreux systèmes cohabitent qu'ils soient sous système d'exploitation AIX, Windows, Linux ou autre. De plus, en fonction des enseignements, une même machine peut être démarrée sous différents systèmes. L'administrateur du réseau souhaite disposer sur son ordinateur d'une commande `runame` («`remote uname`») qui permet d'afficher le nom du système qui fonctionne sur la machine dont le nom est passé en argument, au moment de l'exécution de la commande. Vous êtes chargés du développement du ou des programmes permettant le fonctionnement de la commande `runame`.

Le modèle client/serveur permet-il de modéliser l'application à réaliser ? Quel est le rôle du client ? Du serveur ? Où est le client : sur la machine de l'administrateur ou sur les autres machines ? Et le serveur ?

Les communications entre processus distants dans cette application utiliseront les sockets et donc la pile de protocoles TCP (non connecté, non sûr, non coûteux, correspond aux sockets datagrammes) ou UDP (mode connecté, sûr, coûteux, correspond aux sockets stream), et IP.

Quel protocole de la couche transport est le plus adapté à notre application : UDP ou TCP ? Pourquoi (en dehors du fait que ce TP soit dédié aux sockets datagrammes !) ?

2.1 Implantation : côté serveur

Le code exécutable de la commande `runame` et le squelette du programme C de *serveur.c* vous sont fournis [1]. La commande `runame` prend exactement deux arguments : l'adresse IP de la machine distante dont vous voulez connaître le système et le numéro de port où contacter le processus distant. Pour connaître l'adresse IP de votre machine, vous pouvez utiliser la commande `ifconfig` (voir la page de `man` correspondante pour plus d'informations).
L'algorithme du serveur est le suivant :

1. Obtenir le nom du système (primitive *uname*)
2. Obtenir une socket (primitive *socket*)
3. Attacher la socket à une adresse (primitive *bind*)

   tant_que vrai **faire**
4. | Attente de l'arrivée d'un datagramme (primitive *recvfrom*)
5. | Préparation de la réponse
6. | Émission d'un datagramme (primitive *sendto*)

Pour comprendre le fonctionnement de la primitive système `uname`, les pages de manuel (commande `man 2 uname`) et une visite dans le fichier d'entête */usr/include/sys/utsname.h* peuvent être d'une grande utilité.

Décrivez le fonctionnement de la primitive `uname`. Quels sont les différents renseignements indiqués dans une structure `utsname` ? Combien d'octets doit-on prévoir au maximum pour stocker le nom du système ?

1. Vous les trouverez sur la page du Système S4, `http://www.lri.fr/~bastoul/systeme`.

Complétez le fichier *serveur.c* en vous aidant des commentaires et des indications ci-présentes. **Vous joindrez le code source à votre compte-rendu**.

indications :
La programmation en C/C++ de programmes communicants *via* sockets est un exercice délicat où les erreurs sont vites arrivées et pénibles à corriger. Les conseils (à suivre) suivants vous faciliteront la tâche :

– Compilez votre programme régulièrement (après chaque étape de l'algorithme) à l'aide de la commande `gcc -Wall serveur.c -o serveur`, qui vous donnera le maximum d'avertissements.
– Testez systématiquement le code de retour des fonctions appelées pour détecter et comprendre les erreurs à l'exécution. En géneral une fonction retourne 0 si elle s'est bien déroulée mais -1 s'il y a eu un problème (voir les notes de cours du TD n° 2). En cas d'erreur, la variable `errno` est instanciée avec le code de l'erreur qui convient, il est possible de le visualiser grâce à la fonction standard suivante :

```
#include <errno.h>
void perror(const char * message);
```

L'appel à `perror` écrit sur la sortie standard d'erreur `stderr` la chaîne de caractères `message` passée en argument, suivie de la chaîne d'erreur correspondant au code d'erreur `errno`.
– Testez votre programme terminé à l'aide de l'exécutable `runame` fourni : lancez votre serveur `serveur` depuis un shell, et envoyez-lui des requêtes depuis une autre fenêtre par un appel à `runame`. Utilisez comme arguments pour `runame` le nom de votre machine renvoyé par la commande `hostname` (ou utilisez le nom `localhost` directement), et comme numéro de port celui que vous aurez défini dans *serveur.c*.

2.2 Implantation : côté client

Le programme `runame` est construit selon l'algorithme suivant :

1. Obtenir une socket (primitive *socket*)
2. Attacher la socket à une adresse (primitive *bind*)
3. Émettre un datagramme à l'adresse déduite du nom et du port en argument (primitive *sendto*)
4. Attente de l'arrivée du datagramme reponse (primitive *recvfrom*)
5. Affichage de la réponse

Complétez le fichier *runame.c* en vous aidant des commentaires et des indications ci-avant. **Vous joindrez le code source à votre compte-rendu.**

3 Sockets datagramme et fiabilité

À partir de l'application précédente, construisez une application client-serveur dont le client envoie en rafale 2000 messages de 200 octets au serveur (sans attendre une quelconque réponse du serveur), et dont le serveur ne fait qu'afficher le nombre de messages reçus.

Que constatez-vous ? Expliquez.

Commentaires personnels sur le TP (résultats attendus, difficultés, critiques etc.).