Rshell pentru UNIX

Dimitriu Gabriel ISC

Cuprins

Capitolul 1

Introducere

Acesta este implementarea programului rshell (Remote Shell) pentru masinile UNIX. El ne permite sa executam comenzi pe masina target, aceste sunt comenzi Bash, returnind apoi rezultatul pe masina host. Este un program mai simplu decit telnet-ul.

Limitarile sunt:

- Nu suporta login, su si alte schimbari de user.
- Nesuportind optiunile spuse mai inainte se poatevedea ca userul care porneste rshell-ul are accesul ingradit sau are complet accesul in functie de userul care a lansat serverul pe masina target.
- Nu suprota apelul de programe care lucreaza grafic sau cu consola, de exemplu nu se poate lansa un program X, el se lanseaza dar ruleaza pe masina target si pierdem controlul lui.
- Nu suporta aducerea de fisiere, deoarece a fost gindit ca un program care ruleaza numai pe masina target.

In toata lucrarea se numeste **masina target** masina care ruleaza servarul si **masina host** masina de pe care se lanseaza rshell-ul adica clientul.

Programul este implementat in ANSI C, pentru masinile UNIX, avind o portabilitate maxima. Cerinta este ca masina target sa aiba instalat interpretorul **bash** pe care se bazeaza programul si sa aiba un compilator compatibil gcc. Ca metoda de comunicare cu reteaua s-a am stabilit ca se va lucra pe socketuri cu conexiune adica TCP, deoarece este mai sigur, programul neavind confirmari pe parcurs.

Comenzile se introduc dupa apelarea programului cu **rshell "IP_adress"** si trebuie neaparat ca intre comanda si argument sa se lase un spatiu. Pentru iesiere se tasteaza exit. Daca se observa ca listingul rezultat este prea lung se poate face un pipe catre orice pager dar se incurajeaza pagerul less.

Servarul din constructie nu accepta decit un singur client.

Capitolul 2

Implementare

2.1 Protocolul de comunicatie

Se transmite dupa cum am mai amintit in introducere pe protocolul TCP, dar am mai creat si un protocol aplicatie pentru a putea realiza sincronizarea paginilor transmise cind se face cerea de pager.

Protocolul aplicatie este:

- Se trasmite in total 515 caractere imparite astfel: 512 caractere datele si 3 caractere sincronizare.
- Caracterele de sincronizare sunt intotdeauna ultimile trei caractare dupa date.
- Daca nu este ultimul pachet din mesaj se trasmite "000" iar daca este ultimul se trasmite "END".
- Sau utilizat trei caractere pentru posibilitati de dezvoltare ulterioare.
- Clientul nu transmite sincronizare.
- Pentru iesire se transmite "exit" in zona de date.
- Comunicarea se face pe portul 12345.

Numarul de caractere de cod este stocat in variabila preprocesor SIZE, iar datele sunt receptionate si transmise din bufferul de mesaje numit buff care are dimensiunea SIZE+3, deoarece trebuie sa aiba loc tot mesajul transmis de client sau receptionat de la server.

Mesajele de continuare si sfirsit sunt memorate in sirurile statice noend[3]="000" si end[3]="END".

2.2 Implementarea serverului

2.2.1 Prezentarea generala

Initializam conexiunea:

- Cream socketul cu comanda socket.
- Initializam adresa serverului adica variabila server_addr cu valoarea portului, tipul de conexiune AF_INET si se permite accesul de la orice adresa.
- Legam socketul la adresa cu comanda bind.
- Initializa coada de mesaje cu comanda listen.
- Initializam adresa clientului :client_len.

Din acest moment toate operatiile se executa intr-o bucla infinita while(1).

Accept conexiunea daca clientul cere vreo conexiune prin comanda accept.

Creez un proces nou prin functia fork() si daca sunt in procesul client (adica noul proces) realizez din nou o blucla infinita (while(1)) in care se vor executa toate prelucrarile procesului client, dina cest bucla nu se iese decit cu exit. In tot acest timp procesul parinte asteapta fiul datorita functiei waitpid(), dupa care revine in bucla infinita in asteptare.

2.2.2 Prezentarea prelucrarilor asupra datelor

Dupa ce aceptam conexiunea realizam un recv (o receptie a unei comenzi), datele aflindu-se in buff. Daca se receptioneza comanda **exit**, de remarcat ca clientul nu pune in cei trei biti de sincronizare deci clientul va putea transmite 515 caractere, am pastrat lungimea constanta a pachetului. Cu toate ca clientul nu va transmite toata lungimea decit in cazuri exceptionale deoarece se transmite decit sirul de caractere nu si zerourile terminale deci avind lungimea strlen(buff). Cind se primeste comanda exit, atunci se transmite clientului exit, bineninteles fara caractere de control, adica un pachet de 4 caractere care este interpretat diferit de client, deoarece si el asteapta un pachet de patru caractere, dupa care serverul iese. Aceasta comunicatie suplimentara a fost utilizata pentru a ne asigura ca si clientul si serverul au inceiat conversatia.

In primul rind se testeaza daca acesta comanda venita este un **cd**, daca da se trateaza separat, executind o comanda de sistem **chdir()** deoarece daca am apela bash sau popen (in cazul de fata) nu am putea realiza un chdir deoarece cind se termima pipe-ul sau shell-ul se revine la starea initiala acesta fiind un o comanda care influenteaza numai shell-ul respectiv. Catre client se transmite codul de eroare returnat de functia chdir(), bineinteles cu terminatorul **END**.

Daca nu este o comanda **cd** se trece la tratarea normala a apelului. Si anume se deschide un pipe cu functia popen() catre un fisier de unde se va citi rezultatul catre client.

Raspunsul catre client se realizeaza intr-o bucla while() cu conditia de iesire de sfirsit de fisier, deoarece asupra pipe-ului se pot fece orice operatie se facea asupra unui fisier fizic. Din acest pipe se citesc binar cu ajutorul functiei fread() 512 caractere, la care se adauga terminatorul in functie daca este sau nu sfirsitul fisierului, rezultatul se paseaza functiei de transmitere send().

Dupa confirmarea transmiterii se trece din nou in pozitia de asteptare.

Sursa serverului este prezentata in sectiunea 3.1

2.3 Implementarea clientului

2.3.1 Prezentare generala

Daca nu am furnizat adresa masinii terget, adica a serverului, ni se cere acesta adresa si se iese din program.

Se fac initializarile si se initilizeaza conexiunea:

- Se seteaza variabila server_addr cu adresa serverului, portul care este acelasi ca la server 12345 si tipul de conexiune care este tot AF_INET.
- Se creeza socketul cu ajutorul comenzi socket().
- Se face conexiune cu connect().

Din acest moment se intra intr-o bucla while() din care se iese stabilind valoarea varibilei exit1 diferita de 0. Cind se iese se inchide client_socket.

2.3.2 Prezentarea prelucrarilor

In primul rind se citesc datele de la tastatura (comanzile) si se verifica daca exista cerere de pager, adica daca exista carcaterul —. Daca da se seteaza variabila pipe=1 si in stringul pager se stocheaza numele pagerului. Se pot utiliza doua tipuri de pager **less** si **more** adica cele mai utilizate

pageruri din lumea masinilor UNIX. Se deschide pipe de scriere cu argumentul pager, astfel incit datele nu se transmit catre stdout cum est modul normal de lucru ci spre acel pager.

Daca comanda este exit se transmite acesta comanda serverului si se asteapta primirea de la server a unui mesaj care se afiseaza pe ecran si se iese afara. Obs: Se ignora sincronizarea pe care a transmis-o serverul deoarece se stie ce tip comanda este.

Altfel se intra in procesarea normala:

Se transmite comanda serverului cu comanda send() si imediat se intra intr-o bucla while() cu conditia de terminare exit2 diferit de zero.

In acesta bucla se asteapta raspunsul de la server si se afiseaza zona de date de raspuns. Se fac testarile de sincronizare si daca se intilneste setul de caractere **END** in cele trei caractere de sincronizare de la final se seteaza varibila exit2=1 sau se lasa variabila neschimbata si se afiseaza la stdout daca pipe=0 sau catre pipe daca pipe=1. Scrierea catre pipe se face cu fuctia fwrite(), iar catre stdout cu functia printf(%s).

Codul sursa este prezentat in sectiunea 3.2.

Capitolul 3

Listingul surselor

3.1 Programul server

```
/* Programul SERVER pentru remote shell */
   #include<stdio.h>
   #include<stdlib.h>
   #include<string.h>
   #include<unistd.h>
   #include<sys/socket.h>
   #include<sys/wait.h>
   #include<sys/types.h>
   #include<netinet/in.h>
   #include<netdb.h>
   #define SIZE 512
   #define PORT 12345
   static char buff[SIZE+3];            //bufferul de mesaj
   static char noend[3]="000";         //mesaj de neterminare pipe
   static char end[3]="END";             //mesaj de terminare
   static char argument[SIZE];                 //argumentul
   char *eroare;
   FILE *fpipe;                        //pipe
   pid_t client;                     //pid pentru procese
   int status;
   int main()
   {
   int listen_socket,    //socket pentru ascultare
      server_socket,        //socket pentru server
      client_len;           //lungimea adresei client
   struct sockaddr_in
      server_addr,        //adresa serverului
      client_addr;        //adresa clientului
   int i;
      //creare socket
      if((listen_socket=socket(AF_INET,SOCK_STREAM,0))<0)
      {
         perror("listen error");
         exit(1);
      }
      memset((char *)&server_addr,0,sizeof(server_addr));
      server_addr.sin_family=AF_INET;
```

```
server_addr.sin_port=htons(PORT);
server_addr.sin_addr.s_addr=htonl(INADDR_ANY);
//legare (bind)
if(bind(listen_socket,(struct sockaddr *)&server_addr,sizeof(server_addr))<0)
{
    perror("bind error");
    close(listen_socket);
    exit(2);
}
if(listen(listen_socket,5)<0)
{
    perror("listen socket error");
    close(listen_socket);
    exit(3);
}
client_len=sizeof(client_addr);
while(1)
{
    //acceptare conexiune
    if((server_socket=accept(listen_socket,(struct sockaddr *)&client_addr,&client_len))<0)
    {
        perror("accept error");
        close(listen_socket);
        exit(4);
    }
    if((client=fork())<0)
    {
        perror("fork client error");
        close(server_socket);
        close(listen_socket);
        exit(5);
    }
    if(client==0)    //proces client
    {
        while(1)
        {
            memset((char *)&buff,0,sizeof(buff));
            memset((char *)&argument,0,sizeof(argument));
            recv(server_socket,buff,SIZE,0);
            if(strcmp(buff,"exit")==0)
            {
                send(server_socket,"exit",4,0);
                close(server_socket);
                exit(6);
            }
            for(i=0;i<2;i++)
            argument[i]=buff[i];
            if(strcmp(argument,"cd")==0)
            {
                for(i=3;i<strlen(buff);i++)
                argument[i-3]=buff[i];
                i=chdir(argument);
                memset((char *)&buff,0,sizeof(buff));
                memset((char *)&argument,0,sizeof(argument));
```

```
                    eroare=strerror(i);
                    strcpy(buff,eroare);
                    for(i=0;i<3;i++) buff[SIZE+i]=end[i];
                    send(server_socket,buff,SIZE+3,0);
                }
                else
                {
                    strcpy(argument,buff);
                    if((fpipe=(FILE *)popen(argument,"r"))==(FILE *)NULL)
                    {
                        perror("error open pipe");
                        exit(7);
                    }
                    while(!feof(fpipe))
                    {
                        memset((char *)&buff,0,sizeof(buff));
                        fread(buff,sizeof(char),SIZE,fpipe);
                        if(!feof(fpipe)) for(i=0;i<3;i++) buff[SIZE+i]=noend[i];
                        else for(i=0;i<3;i++) buff[SIZE+i]=end[i];
                        send(server_socket,buff,SIZE+3,0);
                    }
                    pclose(fpipe);
                }
            }
        }
        if((client=waitpid(client,&status,0))<0)
        {
            perror("error waitpid client");
            close(listen_socket);
            exit(7);
        }
    }
}
```

3.2 Programul client

```
/* Programul client pentru remote shell */
    #include<stdio.h>
    #include<stdlib.h>
    #include<string.h>
    #include<unistd.h>
    #include<sys/socket.h>
    #include<sys/types.h>
    #include<netinet/in.h>
    #include<arpa/inet.h>
    #include<netdb.h>
    #define SIZE 512
    #define PORT 12345
    static char consola[SIZE];          //buffer de consola
    static char buff[SIZE+3];           //bufferul de date
    static char data[SIZE];             //buffer pentru afisare
    static char pager[SIZE];            //buffer pentru pager
    static char sincro[3];              //buffer pentru sincronizare
```

```
int main(int argc, char **argv)
{
int client_socket;        //socketul client
struct sockaddr_in server_addr;
int i,j;                  //contoare
int exit1,exit2,pipe;   //sincronizari
FILE *fpipe;              //pipe de iesire
    if(argc!=2)
    {
        printf("rshell_client adresa");
        exit(1);
    }
    memset((char *)&server_addr,0,sizeof(server_addr));
    server_addr.sin_family=AF_INET;
    server_addr.sin_port=htons(PORT);
    server_addr.sin_addr.s_addr=inet_addr(argv[1]);
    //creare socket
    if((client_socket=socket(AF_INET,SOCK_STREAM,0))<0)
    {
        perror("error socket");
        exit(2);
    }
    //conectare
    if(connect(client_socket,(struct sockaddr *)&server_addr,sizeof(server_addr))<0)
    {
        perror("error connect");
        close(client_socket);
        exit(3);
    }
    exit1=0;
    while(exit1==0)
    {
        memset((char *)&consola,0,sizeof(consola));
        memset((char *)&data,0,sizeof(data));
        memset((char *)&pager,0,sizeof(pager));
        gets(consola);
        i=0;
        while((consola[i]!='—')&&(i<SIZE))
        {
            data[i]=consola[i];
            i++;
            if(consola[i]=='—') pipe=1;
        }
        if(pipe==1)
        {
            j=0;i++;
            while(i<SIZE)
            {
                pager[j]=consola[i];
                i++;
                j++;
            }
        }
        if((strcmp(pager,"less")==0)——(strcmp(pager,"more")==0))
```

```
        {
            if((fpipe=(FILE *)popen(pager,"w"))==(FILE *)NULL)
            {
                perror("nu se poate deschide pipe");
                strcpy(data,"exit");
                pipe=0;
            }
            else pipe=1;
        }
        else pipe=0;
        if(strcmp(data,"exit")==0)
        {
            send(client_socket,data,strlen(data),0);
            memset((char *)&buff,0,sizeof(buff));
            recv(client_socket,buff,SIZE+3,0);
            for(i=0;i<SIZE;i++) data[i]=buff[i];
            printf("%s\n",data);
            close(client_socket);
            exit(0);
        }
        else
        {
            send(client_socket,data,strlen(data),0);
            exit2=0;
            while(exit2==0)
            {
                memset((char *)&buff,0,sizeof(buff));
                recv(client_socket,buff,SIZE+3,0);
                for(i=0;i<SIZE;i++) data[i]=buff[i];
                for(i=0;i<3;i++)    sincro[i]=buff[SIZE+i];
                if(strcmp(sincro,"END")==0) exit2=1;
                if(strcmp(sincro,"000")==0) exit2=0;
                if(pipe==0)         printf("%s ",data);
                if(pipe==1)         fprintf(fpipe,"%s",data);
                fflush(stdin);
            }
            if(pipe==0)    printf("\n");
            if(pipe==1)
            {
                fprintf(fpipe,"\n");
                pclose(fpipe);
            }
        }
    }
    close(client_socket);
    exit(0);
}
```