Michał Dybaś

Temat projektu:

Projekt i implementacja serwera współbieżnego oraz klienta programu telnet w środowisku RPC (język C).

Spis treści

1. Kod źródłowy definicji danych w formacie XDR telnet.x

```
struct MyData { /* dane wejściowe i wyjściowe) */
    opaque message<>;
    int state;
    int client_id;
};

program MY_PROGRAM { /* definicja programu RPC o nazwie MY_PROGRAM */
    version MY_VERSION { /* program składający się z jednej wersji MY_VERSI
ON */
        MyData EXECUTE(MyData) = 1; /* numer procedury */
    } = 1; /* numer wersji */
} = 0x31230000; /* numer programu*/
```

2. Kod źródłowy pliku nagłówkowego telnet.h wygenerowany przez rpcgen na podstawie specyfikacji

```
#ifndef _TELNET_H_RPCGEN
#define _TELNET_H_RPCGEN

#include <rpc/rpc.h>


#ifdef __cplusplus
extern "C" {
#endif


struct MyData {
    struct {
        u_int message_len;
        char *message_val;
    } message;
    int state;
    int client_id;
};
typedef struct MyData MyData;

#define MY_PROGRAM 0x31230000
#define MY_VERSION 1

#if defined(__STDC__) || defined(__cplusplus)
#define EXECUTE 1
extern  MyData * execute_1(MyData *, CLIENT *);
extern  MyData * execute_1_svc(MyData *, struct svc_req *);
extern int my_program_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define EXECUTE 1
extern  MyData * execute_1();
extern  MyData * execute_1_svc();
extern int my_program_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_MyData (XDR *, MyData*);

#else /* K&R C */
extern bool_t xdr_MyData ();

#endif /* K&R C */
```

```
#ifdef __cplusplus
}
#endif

#endif /* !_TELNET_H_RPCGEN */
```

3. Kod źródłowy pliku nagłówkowego telnet_xdr.c wygenerowany przez rpcgen na podstawie specyfikacji

```
#include "telnet.h"

bool_t
xdr_MyData (XDR *xdrs, MyData *objp)
{
    register int32_t *buf;

     if (!xdr_bytes (xdrs, (char **)&objp-
>message.message_val, (u_int *) &objp->message.message_len, ~0))
         return FALSE;
     if (!xdr_int (xdrs, &objp->state))
         return FALSE;
     if (!xdr_int (xdrs, &objp->client_id))
         return FALSE;
    return TRUE;
}
```

4. Kod źródłowy klienta telnet telnet_client.c

```
#include <stdio.h>
#include <string.h>
#include "telnet.h"
#include <memory.h>

static struct timeval TIMEOUT = { 25, 0 }; // Deklaracja i inicjalizacja zm
iennej przechowującej timeout oczekiwania na wiadomość zwrotną z wywołanej
procedury zdalnej
MyData * execute_1(MyData *argp, CLIENT *clnt) // Deklarcja i defnicja funk
cji wywołującej procedurę zdalną
{
    static MyData clnt_res;

    memset((char *)&clnt_res, 0, sizeof(clnt_res));
    if (clnt_call (clnt, EXECUTE,
        (xdrproc_t) xdr_MyData, (caddr_t) argp,
        (xdrproc_t) xdr_MyData, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
    }
    return (&clnt_res);
}

void main(int argc, char* argv[])
{
    // Deklaracja i inicjalizacja zmiennych
    char *host; // Zmienna przechowująca adres hosta
    char msg[4095]; // Zmienna przechowująca wiadomośc odczytaną z klawiatu
ry
    CLIENT *clnt = NULL; // Deklaracja uchwytu klienta
    MyData  inPut; // Struktura przechowująca argumenty przekazywane do pro
cedury zdalnej
    MyData  *outPut; // Struktura przechowująca wyniki otrzymane dzięki wyw
ołaniu procedury zdalnej

    // Inicjalizacja struktury zawierającej argumenty przekazywane do proce
dury zdalnej
    inPut.message.message_val = "";
    inPut.message.message_len = 0;
    inPut.state = 0;
    inPut.client_id = -
1; // Id mniejsze od 0 oznacza, że nie przyznano jeszcze id klienta

    if (argc < 2) // Sprawdzenie czy podano odpowiednią ilość argumentów we
jściowych podczas uruchomienia programu
    {
```

```
        printf ("|Error|: Błąd utworzenia uchwytu dla klienta, nie podano a
dresu ip hosta.\n");
        exit(1);
    }
    host = argv[1]; // Pobranie adres hosta z argumentów wejściowych progra
mu

    clnt = clnt_create (host, MY_PROGRAM, MY_VERSION, "tcp"); // Utworzenie
 uchwytu klienta
    if (clnt == NULL)
    {
        printf ("|Error|: Błąd utworzenia uchwytu dla klienta, brak połącze
nia lub nieprawidłowy adres hosta.\n");
        exit(1);
    }

    outPut = execute_1(&inPut, clnt); // Wywołanie procedury zdalnej w celu
 otrzymania id klienta
    if (outPut == (MyData *) NULL) // Sprawdzenie czy otrzymano wiadomość z
wrotną od wywołanej procedury zdalnej
    {
        printf ("|Error|: Brak odpowiedzi od serwera, połączenie zostało ze
rwane.\n");
        if(clnt != NULL)
            clnt_destroy(clnt); // Usunięcie uchwytu klienta
        exit(1); // Nieudane wyjście z programu
    }

    if(outPut->client_id < 0)
    {
        printf ("|Client|: Serwer odmówił połącznia z powodu przekroczonia
liczby hostów. Spróbuj później.\n");
        if(clnt != NULL)
            clnt_destroy(clnt); // Usunięcie uchwytu klienta
        exit(0); // Nieudane wyjście z programu
    }
    else
    {
        inPut.state = 1;
        inPut.client_id = outPut->client_id;
        // Wiadomość powitalna
        printf("|Client|: Nawiązano połączenie z serwerem (id klienta = %d)
.\n", outPut->client_id + 1);
        printf("|Client|: Podaj komendę i potwierdź klawiszem enter. Pusty
ciąg znaków kończy działanie programu.\n");
    }

    while (1)
    {
```

```
        printf("|Client|: ");
        bzero(msg, sizeof(msg)); // Wyczyszczenie obszaru pamięci tablicy m
sg
        memset(msg, '\0', sizeof(msg));
        fgets(msg, sizeof msg, stdin); // Oczekiwanie na polecenie
        if(strlen(msg) == 1 && msg[0] == '\n') // Dla pustego ciągu znaków,
 kończy działanie programu
            break; // Udane wyjście z programu
        msg[strlen(msg) - 1] = '\0';

        inPut.message.message_val= msg;
        inPut.message.message_len = strlen(msg);

        outPut = execute_1(&inPut, clnt); // Wywołanie procedury zdalnej
        if (outPut == (MyData *) NULL) // Sprawdzenie czy otrzymano wiadomo
ść zwrotną od wywołanej procedury zdalnej
        {
            printf ("|Client|: Brak odpowiedzi od serwera, połączenie zosta
ło zerwane.\n");
            if(clnt != NULL)
                clnt_destroy(clnt); // Usunięcie uchwytu klienta
            exit(1); // Nieudane wyjście z programu
        }

        if(outPut->state == 3)
        {
            printf ("|Client|: Serwer zgłosił błąd, połączenie zostało zerw
ane.\n");
            if(clnt != NULL)
                clnt_destroy(clnt); // Usunięcie uchwytu klienta
            exit(1); // Nieudane wyjście z programu
        }

        if(outPut->message.message_len > 0)
            printf("%s", outPut-
>message.message_val); // Wyświetlenie wiadomości otrzymanej poprzez wywoła
nie metody zdalnej
    }

    inPut.state = 2;
    outPut = execute_1(&inPut, clnt); // Wywołanie procedury zdalnej w celu
 poinformowania serwera o zakończeniu połączenia
    if (outPut == (MyData *) NULL) // Sprawdzenie czy otrzymano wiadomość z
wrotną od wywołanej procedury zdalnej
    {
        printf ("|Error|: Brak odpowiedzi od serwera, połączenie zostało ze
rwane.\n");
        if(clnt != NULL)
            clnt_destroy(clnt); // Usunięcie uchwytu klienta
```

```
        exit(1); // Nieudane wyjście z programu
    }

    if(clnt != NULL) {
        clnt_destroy(clnt); // Usunięcie uchwytu klienta
        printf("|Client|: Zakończono połączenie z serwerem.\n");
    }
    printf("|Client|: Działanie aplikacji zostało zakończone.\n");
    exit(0); // Udane wyjście z programu
}
```

5. Kod źródłowy serwera telnet_server.c

```c
#include "telnet.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pty.h>
#include <stdbool.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>

// KOMPILACJA Z FLAGĄ -lutil -lpthread
#define MAX_HOST 20 // Deklaracja i incjalizacja stałej przechowujacej ilość maksymalną liczbe klientów
#define TIMEOUT_SECONDS 5 * 60// Deklaracja i incjalizacja stałej przechowujacej maskymalny czas bezczynności klienta

struct telnet_client // Definicja struktury przechowujacej dane klienta
{
    int master, slave; // Deskryptory PTY używane do wysłania polecenia do shella
    bool active; // Flaga sprawdzająca czy struktura jest wykorzystywana przez jakiegoś klienta
    int shellFdRW[2]; // Deskryptory pipe używane do odczytu danych z shella
    pid_t shellPid; // Identyfikator procesu shella
    time_t lastTimeUsed; // Zmienna przechowująca w sekundach czas ostatniego żądania klienta
};
typedef struct telnet_client telnet_client; // Definicja typu

void* drop_unused_connections(void* pclient) // Definicja funkcji wątku zwalniająca zbyt długo nie używane struktury klientów
{
    telnet_client *clients = pclient;
    time_t current_time = time(NULL);
    for(int i = 0; i < MAX_HOST; i++)
    {
        if(clients[i].active)
            if(current_time - clients[i].lastTimeUsed >= TIMEOUT_SECONDS)
            {
                clients[i].active = false;
                kill(clients[i].shellPid, SIGKILL); // Wysyłamy sygnał SIGKILL do procesu potomnego (shella)
```

```
                close(clients[i].shellFdRW[0]);
                printf("|Server|: Zakończono połączenie z klientem (id klie
nta = %d) z powodu braku aktywności.\n", i + 1);
            }
    }
    return NULL;
}

MyData * execute_1_svc(MyData *argp, struct svc_req *rqstp)
{
    static MyData result;
    static telnet_client clients[MAX_HOST];

    pthread_t thread_id;
    pthread_create(&thread_id, NULL, drop_unused_connections, clients); //
Wywołanie wątku sprawdzającego, czy przekroczono TIMEOUT

    // Inicjalizacja struktury zawierającej argumenty przekazywane do proce
dury zdalnej
    result.message.message_val = "";
    result.message.message_len = 0;
    result.client_id = -
1; // Id mniejsze od 0 oznacza, że nie przyznano jeszcze id klienta

    if(argp-
>state == 0) // Odebrano prośbę od klienta o próbie nawiązania połączenia z
 serwerem
    {
        result.state = 0; // Ustawienie stanu połączenia na jaki oczekuje t
eraz klient
        for(int i = 0; i < MAX_HOST; i++) // Sprawdzenie czy jest wolne mie
jsce w tablicy klientów
        {
            if(!clients[i].active)
            {
                clients[i].active = true;
                clients[i].lastTimeUsed = time(NULL);
                result.client_id = i;
                if(clients[i].master == 0 || clients[i].slave == 0)
                    openpty(&clients[i].master, &clients[i].slave, NULL, NU
LL, NULL); // Utworzenie PTY
                pipe(clients[i].shellFdRW);
                if ((clients[i].shellPid = fork()) == 0) // Utworzenie proc
esu potomnego dla shella
                {
                    dup2(clients[i].slave, 0); // Podłączenie strony slave
PTY do standardowego wejścia procesu potomnego
                    dup2(clients[i].shellFdRW[1], 1); // Podłączenie strony
 zapisu pipe do standardowego wyjścia procesu potomnego
```

```
                    dup2(clients[i].shellFdRW[1], 2); // Podłączenie strony
 zapisu pipe do standardowego wyjścia błędów procesu potomnego
                    close(clients[i].shellFdRW[0]); // Zamknięcie nieużywan
ej strony odczytu pipe w procesie potomnym
                    execl("/bin/sh", "-
i", (const char*)NULL); // Zastąpienie obrazu aktualnego programu w przestr
zeni adresowej procesu obrazem programu shella. Wykonanie programu /bin/sh
z parametrm -
i (powłoka interaktywna) oraz standardowym środowiskiem z zmiennej **enviro
n
                }
                close(clients[i].shellFdRW[1]); // Zamknięcie nieużywanej s
trony zapisu pipe w procesie głównym
                printf("|Server|: Nawiązano połączenie z nowym klientem (id
 klienta = %d).\n", i + 1);
                break;
            }
        }
    }
    else if(argp-
>state == 1) // Odebranie komendy od klienta do wykonania w powłoce
    {
        result.state = 1; // Ustawienie stanu połączenia na jaki oczekuje t
eraz klient
        if(argp->client_id > -1 && argp-
>client_id < MAX_HOST) // Sprawdzenie czy id klienta znajduje się w zakresi
e
        {
            if(!clients[argp-
>client_id].active) // Sprawdzenie czy dane połączenie jest nie aktywne
            {
                result.state = 3; // Ustawienie stanu połączenia, aby poinf
ormować klienta o błędzie
            }
            else // Jeśli połączenie jest aktywne serwer wykonuje komende o
d klienta
            {
                /* Odebranie wiadomości od klienta */
                int commandLen = argp->message.message_len;
                char command[commandLen];
                bzero(command, commandLen); // wyczyszczenie obszaru pamięc
i
                for(int i = 0; i < commandLen + 1; i++)
                    command[i] = argp->message.message_val[i];
                command[commandLen] = '\0';

                if(strcmp(command, "exit") == 0)
                {
```

```
                    kill(clients[argp-
>client_id].shellPid, SIGKILL); // Wysyłamy sygnał SIGKILL do procesu potom
nego (shella)
                    close(clients[argp->client_id].shellFdRW[0]);

                    pipe(clients[argp->client_id].shellFdRW);
                    if ((clients[argp-
>client_id].shellPid = fork()) == 0) // Utworzenie procesu potomnego dla sh
ella
                    {
                        dup2(clients[argp-
>client_id].slave, 0); // Podłączenie strony slave PTY do standardowego wej
ścia procesu potomnego
                        dup2(clients[argp-
>client_id].shellFdRW[1], 1); // Podłączenie strony zapisu pipe do standard
owego wyjścia procesu potomnego
                        dup2(clients[argp-
>client_id].shellFdRW[1], 2); // Podłączenie strony zapisu pipe do standard
owego wyjścia błędów procesu potomnego
                        close(clients[argp-
>client_id].shellFdRW[0]); // Zamknięcie nieużywanej strony odczytu pipe w
procesie potomnym
                        execl("/bin/sh", "-
i", (const char*)NULL); // Zastąpienie obrazu aktualnego programu w przestr
zeni adresowej procesu obrazem programu shella. Wykonanie programu /bin/sh
z parametrm -
i (powłoka interaktywna) oraz standardowym środowiskiem z zmiennej **enviro
n
                    }
                    close(clients[argp-
>client_id].shellFdRW[1]); // Zamknięcie nieużywanej strony zapisu pipe w p
rocesie głównym
                    printf("|Server-%d|: exit\n", argp->client_id + 1);
                }
                else
                {
                    char msg[4095]; // Zmienna przechowująca wiadomośc odcz
ytaną z shella
                    fd_set rfds; // Struktura opisująca zestaw deskryptorów
                    struct timeval tv; // Struktura opisująca czas użyta do
 ustawienia opóźnienia dla select
                    tv.tv_sec = 0; // 0s
                    tv.tv_usec = 0; // 0ms

                    bzero(msg, 4095); // Wyzerowanie zmiennych
                    printf("|Server-%d|: %s\n", argp-
>client_id + 1, command);
```

```
                    write(clients[argp->client_id].master, command, commandLen); // Wysyłamy polecenie do shella
                    write(clients[argp->client_id].master, "\r\n", 2); // Symulacja Enter

                    FD_ZERO(&rfds); // Wyzerowanie struktury zestawu deskryptorów
                    FD_SET(clients[argp->client_id].shellFdRW[0], &rfds); // Dodanie deskryptora do zestawu deskryptorów

                    sleep(5);
                    if(select(clients[argp->client_id].shellFdRW[0] + 1, &rfds, NULL, NULL, &tv)) // Monitorowanie deskryptora pod kątem możliwości wykonania operacji I/O
                    {
                        result.message.message_len = read(clients[argp->client_id].shellFdRW[0], msg, 4095 - 1); // Odczytanie wyniku działania polecenia
                        msg[result.message.message_len] = '\0'; // Dodanie znaku końca ciągu znaków
                        result.message.message_val = msg;
                    }
                }
            }
        }
        else
            result.state = 3; // Ustawienie stanu połączenia, aby poinformować klienta o błędzie
    }
    else if(argp->state == 2) // Odebrano informację od klienta o zakończeniu połączenia z serwerem
    {
        result.state = 2; // Ustawienie stanu połączenia na jaki oczekuje teraz klient
        if(argp->client_id >= 0 && argp->client_id < MAX_HOST) // Sprawdzenie czy id klienta znajduje się w zakresie
            if(clients[argp->client_id].active) // Sprawdzenie czy dane połączenie jest aktywne
            {
                clients[argp->client_id].active = false; // Ustawienie danej struktury klienta na wolną
                kill(clients[argp->client_id].shellPid, SIGKILL); // Wysyłamy sygnał SIGKILL do procesu potomnego (shella)
                close(clients[argp->client_id].shellFdRW[0]);
```

```
                printf("|Server|: Zakończono połączenie z klientem (id klie
nta = %d).\n", argp->client_id + 1);
            }
    }
    else // Jeśli klient podan stanu połączenia który nie istnieje
    {
        result.state = 3; // Ustawienie stanu połączenia, aby poinformować
klienta o błędzie
        printf("|Server|: Otrzymano nieprawidłowe żądanie od klienta, które
 zostanie zignorowane (id klienta = %d).\n", argp->client_id + 1);
    }

    return &result; // Zwrócenie struktury klientowi
}

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void my_program_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
    union {
        MyData execute_1_arg;
    } argument;
    char *result;
    xdrproc_t _xdr_argument, _xdr_result;
    char *(*local)(char *, struct svc_req *);

    switch (rqstp->rq_proc) {
    case NULLPROC:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void, (char *)NULL);
        return;

    case EXECUTE:
        _xdr_argument = (xdrproc_t) xdr_MyData;
        _xdr_result = (xdrproc_t) xdr_MyData;
        local = (char *(*)(char *, struct svc_req *)) execute_1_svc;
        break;

    default:
        svcerr_noproc (transp);
        return;
    }
    memset ((char *)&argument, 0, sizeof (argument));
    if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argumen
t)) {
        svcerr_decode (transp);
        return;
    }
```

```
    result = (*local)((char *)&argument, rqstp);
    if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, r
esult)) {
        svcerr_systemerr (transp);
    }
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argume
nt)) {
        printf("|Error|: Wystąpił błąd, nie udało się zwolnić argumentów.\n
");
        exit(1);
    }
    return;
}

int main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset (MY_PROGRAM, MY_VERSION);

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
        printf("|Error|: Wystąpił błąd, nie można utworzyć usługi tcp.\n");
        exit(1);
    }
    if (!svc_register(transp, MY_PROGRAM, MY_VERSION, my_program_1, IPPROTO
_TCP)) {
        printf("|Error|: Wystąpił błąd, nie udało się zarejestrować (MY_PRO
GRAM, MY_VERSION, tcp).\n");
        exit(1);
    }

    svc_run();
    printf("|Error|: Wystąpił błąd, serwer przestał funkcjonować. (Powrót z
 funkcji svc_run()).\n");
    exit(1);
}
```

6. Działanie aplikacji