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 3](#_Toc99454114)

[2. Kod źródłowy pliku nagłówkowego telnet.h wygenerowany przez rpcgen na podstawie specyfikacji 4](#_Toc99454115)

[3. Kod źródłowy pliku nagłówkowego telnet\_xdr.c wygenerowany przez rpcgen na podstawie specyfikacji 6](#_Toc99454116)

[4. Kod źródłowy klienta telnet telnet\_client.c 7](#_Toc99454117)

[5. Kod źródłowy serwera telnet\_server.c 11](#_Toc99454118)

[6. Działanie aplikacji 18](#_Toc99454119)

# 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\_VERSION \*/

        MyData EXECUTE(MyData) = 1; /\* numer procedury \*/

    } = 1; /\* numer wersji \*/

} = 0x31230000; /\* numer programu\*/

# 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 \*/

# 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;

}

# 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 zmiennej przechowującej timeout oczekiwania na wiadomość zwrotną z wywołanej procedury zdalnej

MyData \* execute\_1(MyData \*argp, CLIENT \*clnt) // Deklarcja i defnicja funkcji 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 klawiatury

    CLIENT \*clnt = NULL; // Deklaracja uchwytu klienta

    MyData  inPut; // Struktura przechowująca argumenty przekazywane do procedury zdalnej

    MyData  \*outPut; // Struktura przechowująca wyniki otrzymane dzięki wywołaniu procedury zdalnej

    // Inicjalizacja struktury zawierającej argumenty przekazywane do procedury 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 wejściowych podczas uruchomienia programu

    {

        printf ("|Error|: Błąd utworzenia uchwytu dla klienta, nie podano adresu ip hosta.\n");

        exit(1);

    }

    host = argv[1]; // Pobranie adres hosta z argumentów wejściowych programu

    clnt = clnt\_create (host, MY\_PROGRAM, MY\_VERSION, "tcp"); // Utworzenie uchwytu klienta

    if (clnt == NULL)

    {

        printf ("|Error|: Błąd utworzenia uchwytu dla klienta, brak połączenia 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 zwrotną od wywołanej procedury zdalnej

    {

        printf ("|Error|: Brak odpowiedzi od serwera, połączenie zostało zerwane.\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 msg

        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 wiadomość zwrotną od wywołanej procedury zdalnej

        {

            printf ("|Client|: Brak odpowiedzi od serwera, połączenie zostało 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 zerwane.\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 wywołanie 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 zwrotną od wywołanej procedury zdalnej

    {

        printf ("|Error|: Brak odpowiedzi od serwera, połączenie zostało zerwane.\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

}

# Kod źródłowy serwera telnet\_server.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 klienta = %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 procedury 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 teraz klient

        for(int i = 0; i < MAX\_HOST; i++) // Sprawdzenie czy jest wolne miejsce 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, NULL, NULL); // Utworzenie PTY

                pipe(clients[i].shellFdRW);

                if ((clients[i].shellPid = fork()) == 0) // Utworzenie procesu 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 nieużywanej strony odczytu pipe w procesie potomnym

                    execl("/bin/sh", "-i", (const char\*)NULL); // Zastąpienie obrazu aktualnego programu w przestrzeni adresowej procesu obrazem programu shella. Wykonanie programu /bin/sh z parametrm -i (powłoka interaktywna) oraz standardowym środowiskiem z zmiennej \*\*environ

                }

                close(clients[i].shellFdRW[1]); // Zamknięcie nieużywanej strony 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 teraz klient

        if(argp->client\_id > -1 && argp->client\_id < MAX\_HOST) // Sprawdzenie czy id klienta znajduje się w zakresie

        {

            if(!clients[argp->client\_id].active) // Sprawdzenie czy dane połączenie jest nie aktywne

            {

                result.state = 3; // Ustawienie stanu połączenia, aby poinformować klienta o błędzie

            }

            else // Jeśli połączenie jest aktywne serwer wykonuje komende od klienta

            {

                /\* Odebranie wiadomości od klienta \*/

                int commandLen = argp->message.message\_len;

                char command[commandLen];

                bzero(command, commandLen); // wyczyszczenie obszaru pamięci

                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 potomnego (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 shella

                    {

                        dup2(clients[argp->client\_id].slave, 0); // Podłączenie strony slave PTY do standardowego wejścia procesu potomnego

                        dup2(clients[argp->client\_id].shellFdRW[1], 1); // Podłączenie strony zapisu pipe do standardowego wyjścia procesu potomnego

                        dup2(clients[argp->client\_id].shellFdRW[1], 2); // Podłączenie strony zapisu pipe do standardowego 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 przestrzeni adresowej procesu obrazem programu shella. Wykonanie programu /bin/sh z parametrm -i (powłoka interaktywna) oraz standardowym środowiskiem z zmiennej \*\*environ

                    }

                    close(clients[argp->client\_id].shellFdRW[1]); // Zamknięcie nieużywanej strony zapisu pipe w procesie głównym

                    printf("|Server-%d|: exit\n", argp->client\_id + 1);

                }

                else

                {

                    char msg[4095]; // Zmienna przechowująca wiadomośc odczytaną 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 klienta = %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) &argument)) {

        svcerr\_decode (transp);

        return;

    }

    result = (\*local)((char \*)&argument, rqstp);

    if (result != NULL && !svc\_sendreply(transp, (xdrproc\_t) \_xdr\_result, result)) {

        svcerr\_systemerr (transp);

    }

    if (!svc\_freeargs (transp, (xdrproc\_t) \_xdr\_argument, (caddr\_t) &argument)) {

        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\_PROGRAM, MY\_VERSION, tcp).\n");

        exit(1);

    }

    svc\_run();

    printf("|Error|: Wystąpił błąd, serwer przestał funkcjonować. (Powrót z funkcji svc\_run()).\n");

    exit(1);

}

# Działanie aplikacji

