Systemy Operacyjne 1
Laboratorium 10
„Gniazda BSD - protokoły internetowe”
(dwa tygodnie)

dr inż. Arkadiusz Chrobot

1 stycznia 2021

Wstęp

System Linux, który jest kompatybilny z Uniksem, podobnie jak wszystkie inne systemy z nim zgodne posiada mechanizmy odpowiedzialne z obsługę komunikacji przy pomocy sieci komputerowej. Do nawiązywania połączeń oraz transmisji danych służą aplikacjom użytkowym w tym systemie gniazda BSD, które po raz pierwszy zostały wprowadzone w wersji `BSD` systemu Unix. Jest to na tyle udane rozwiązanie, że w chwili obecnej gniazda te stanowią standard dla komunikacji lokalnej i zdalnej w większości współczesnych systemów operacyjnych. W tej instrukcji zawarte są informacje na temat obsługi gniazd BSD. Rozdział 1 opisuje ogólnie możliwości obsługi komunikacji sieciowej w systemie Linux. Rozdział 2 opisuje różnice między protokołami transmisyjnymi sieci Internet. Rozdział 3 zawiera informacje o serwerach iteracyjnych i współbieżnych. Rozdział 4 zawiera opis API gniazd BSD oraz sposobu jego użycia. Rozdział 5 zawiera kody źródłowe dwóch programów (serwera i klienta) komunikujących się za pomocą gniazd BSD z użyciem protokołu UPD/IP. Instrukcję kończy zestaw zadań do samodzielnego wykonania na zajęciach laboratoryjnych.

1. Usługi sieciowe w Linuksie

Gniazda BSD (nazwa pochodzi od wersji BSD Uniksa, istnieją również gniazda TLI pochodzące z Systemu V, ale nie są one używane w Linuksie) stanowią API dla protokołów komunikacyjnych stosowane we wszystkich systemach operacyjnych, które umożliwiają pracę w sieci. W przypadku systemów uniksowych umożliwiają nie tylko pracę w środowisku rozproszonym, ale również lokalną komunikację między procesami stanowiąc uzupełnienie mechanizmów opisanych we wcześniejszych instrukcjach. Dzięki gniazdom można pracować z protokołami należącymi do różnych dziedzin np.: Uniksa, Internetu i Xerox NS. Dodatkowo możliwa jest praca zarówno z protokołami połączeniowymi, jak i bezpołączeniowymi.

2. Różnice między protokołem TCP i UDP

W dziedzinie Internetu aplikacje sieciowe wykorzystują najczęściej jeden z dwóch najpopularniejszych protokołów warstwy transportowej: TCP lub UDP. Za pomocą pierwszego dane są wysyłane w postaci strumienia. Ten protokół jest protokołem połączeniowym, zachowuje kolejność wysyłanych komunikatów po stronie odbiorcy oraz nadzoruje przebieg transmisji dbając o retransmisję zagubionych i zniekształconych pakietów. Nie ma w nim ograniczenia na rozmiar wysyłanych danych. W niektórych zastosowaniach może się jednak okazać zbyt powolny. Można wtedy zamiast niego zastosować protokół UDP. Jest on protokołem bezpołączeniowym. Jednorazowo można za pomocą tego protokołu wysłać dane o wielkości mniejszej od 64 KiB. Protokół ten nie zapewnia retransmisji danych, a przypadki zniekształcenia lub zagubienia pakietów należy obsługiwać samodzielnie. Jest on jednak zdecydowanie szybszy od protokołu TCP. Pakiety obu protokołów są „opakowywane" w komunikaty protokołu IP, stąd najczęściej w literaturze pojawiają się nazwy TCP/IP i UDP/IP.

3. Serwery iteracyjne i współbieżne

Aplikacje sieciowe można podzielić na dwie kategorie: klientów i serwery[1]. Zadaniem serwerów jest wykonywanie usług, o które proszą klienty[2]. Obsługa żądań klientów może przebiegać w sposób sekwencyjny (iteracyjny) lub współbieżny. W pierwszym przypadku serwer nawiązuje połączenie z klientem, realizuje jego prośbę, wysyła odpowiedź i wraca do oczekiwania na połączenia z innymi klientami. Podczas realizacji żądania klienta żaden inny klient nie jest w stanie połączyć się z serwerem. Serwer współbieżny po nawiązaniu połączenia z klientem tworzy proces potomny (lub nowy wątek), który obsługuje prośbę klienta, a proces macierzysty oczekuje na połączenia od innych klientów.

[1] W terminologii uniksowej nazywane demonami.

[2] To nie literówka. W języku polskim wprowadzono dwie odmiany liczby mnogiej wyrazu „klient", aby odróżnić ludzi od programów.

4. API gniazd BSD

Ta część instrukcji zawiera opis struktur danych i funkcji związanych z obsługą gniazd BSD.

4.1. Struktury danych

Strukturę wysyłanych przez gniazdo danych, czyli protokół wyższego rzędu, osadzony na protokole transmisji określa użytkownik. Aby jednak nawiązać połączenie należy zadeklarować i wypełnić odpowiednie pola zmiennej typu `struct sockaddr_in`, czyli pole `sin_family`, któremu w przypadku protokołów internetowych nadaje się wartość stałej `AF_INET` (skrót od *Address Family*), pole `sin_port`, któremu nadaje się numer portu[3] oraz pole `sin_addr`, któremu przypisuje się strukturę typu `struct in_addr` zawierającą adres internetowy drugiej strony połączenia. Jeśli struktura `struct sockaddr_in` jest używana przez serwer do nazwania jego własnego gniazda, przez które będzie nawiązywał połączenie z klientem, to pole `sin_addr` tej struktury może być zainicjowane poprzez nadanie mu wartości stałej o nazwie `INADDR_ANY`. Inicjacja tego pola będzie jeszcze opisywana w dalszej części instrukcji. Więcej informacji na temat tej struktury można uzyskać przy pomocy następującego polecenia:
`man 7 ip`

4.2. Opis funkcji

W tym podrozdziale opisane zostaną tylko funkcje niezbędne do wykonania większości zadań zawartych w instrukcji. Osoby, które chcą dokładniej zapoznać się z tematyką pisania oprogramowania dla sieci komputerowych powinny skorzystać z innych źródeł, jak np.: klasyczna już książka W. Richarda Stevensa „Programowanie zastosowań sieciowych w systemie Unix".

`socket()` - funkcja ta zwraca deskryptor gniazda, poprzez które będzie odbywała się komunikacja między stacjami roboczymi w sieci. Można o niej myśleć jako o funkcji `open()` przeznaczonej dla urządzeń sieciowych. Pierwszy pobierany przez nią argument oznacza rodzinę protokołów (`AF_INET` dla protokołów Internetu), drugi rodzaj gniazda (połączeniowe, inaczej strumieniowe - `SOCK_STREAM`, bezpołączeniowe, inaczej datagramowe - `SOCK_DGRAM`), natomiast ostatni określa, którego konkretnie protokołu będziemy używać (w wypadku protokołów internetowych jest zazwyczaj równy `0`). Funkcja wykorzystywana jest zarówno przez oprogramowanie serwera jak i klienta. W przypadku niepowodzenia utworzenia gniazda zwraca ona wartość `-1`.
Szczegóły: `man socket`

`bind()` - funkcja ta nadaje gniazdu nazwę. Zazwyczaj wywołuje ją serwer przed rozpoczęciem komunikacji z klientem, ale może jej również użyć klient celem zarezerwowania lub sprawdzenia adresu. Jako argumenty `bind()` pobiera deskryptor gniazda, strukturę zawierającą adres komputera (patrz: `man unix`, `man 7 ip` i podrozdział 4.1), oraz rozmiar tej struktury. Jeśli działanie funkcji się powiedzie, to zwróci ona wartość `0`, w przeciwnym przypadku `-1`.
Szczegóły: `man 2 bind`

`connect()` - funkcja ta jest wykorzystywana tylko przez klienta i służy do ustanowienia połączenia z serwerem. Zazwyczaj używana w komunikacji przy pomocy protokołu połączeniowego, ale może być także użyta w komunikacji za pomocą protokołu bezpołączeniowego. Przyjmuje ona jako argumenty: deskryptor gniazda, wskaźnik na strukturę z adresem serwera i rozmiar tej struktury. Jeśli nawiązanie połączenia się powiedzie, to zwraca wartość `0`, a jeśli nie to zwraca `-1`.
Szczegóły: `man connect`

`listen()` - ta funkcja jest używana przez serwer pracujący z protokołem połączeniowym do zgłoszenia, że będzie on nasłuchiwał żądań połączenia. Jeśli serwer odbierze takie żądanie, to umieści je w kolejce. Funkcja `listen()` przyjmuje dwa argumenty: deskryptor gniazda oraz liczbę żądań, które system może umieścić w kolejce zanim zostaną one zaakceptowane (szczegóły: `man tcp`). Jeśli funkcja wykona się prawidłowo, to zwróci `0`, a `-1` w przeciwnym przypadku.
Szczegóły: `man listen`

[3] Należy wiedzieć, że w Linuksie porty od `1` do `1023` są zarezerwowane dla użytkownika `root`. Porty o wyższych numerach mogą być używane przez inne demony. Aby sprawdzić które porty są zarezerwowane, a które nie można zajrzeć do pliku tekstowego `/etc/services`.

`accept()` - funkcja ta jest wywoływana przez oprogramowanie serwera pracującego z protokołem połączeniowym. Służy ona do przyjmowania połączeń. Jej wywołanie wymaga trzech argumentów. Pierwszym jest deskryptor gniazda, drugim wskaźnik do struktury, do której będzie zapisany adres klienta, a trzecim wskaźnik na zmienną, w której zostanie zapisany rozmiar tej struktury. Funkcja pobiera pierwsze żądanie z kolejki i tworzy dla niego gniazdo, o takich samych właściwościach jak gniazdo, do którego nadeszło żądanie. Jeśli kolejka jest pusta, to `accept()` wstrzymuje działanie do momentu, aż pojawi się w kolejce jakieś żądanie. W serwerach współbieżnych gniazdo, którego deskryptor zwraca `accept()` jest obsługiwane przez proces potomny (lub nowy wątek). W przypadku niepowodzenia działania funkcja zwraca wartość `-1`.
Szczegóły: `man 2 accept`

`read(), write()` - funkcje te służą do odbierania i wysyłania danych w protokole połączeniowym. Działają one trochę inaczej, niż w przypadku plików. Zamiast deskryptora pliku do ich wywołania przekazuje się deskryptor gniazda. Jeśli przez gniazdo połączeniowe są wysyłane dane o rozmiarze przekraczającym rozmiar bufora, to wprawdzie są wysyłane jako jeden strumień, ale mogą ulec segmentacji. Oznacza to, że funkcja `read()` może odebrać mniej danych, niż określiliśmy to w jej wywołaniu. Nie jest to błąd, należy po prostu powtórzyć jej działanie. Sposób ich wywoływania został opisany w poprzedniej instrukcji.

`close()` - funkcja ta służy do zamykania gniazda po zakończeniu komunikacji, niezależnie od tego jakim protokołem się posługujemy. Sposób jej wywołania został opisany w poprzedniej instrukcji. W przypadku gniazd jako argument jej wywołania przekazuje się deskryptor gniazda zamiast deskryptora pliku.

`sendto()` - funkcja ta służy do wysyłania informacji przez gniazdo zarówno w protokole bezpołączeniowymi, jak i połączeniowym, choć częściej jest stosowana w tym pierwszym. Przyjmuje ona sześć argumentów wywołania: deskryptor gniazda, wskaźnik na bufor wysyłanych danych, rozmiar bufora, flagę (najczęściej `0`), wskaźnik na strukturę w której zapisany jest adres przeznaczenia oraz rozmiar tej struktury. Funkcja zwraca liczbę przesłanych bajtów lub `-1` jeśli wystąpi wyjątek.
Szczegóły: `man sendto`

`recvfrom()` - funkcja ta służy do odbioru danych z gniazda zarówno w protokole bezpołączeniowym jaki i połączeniowym, choć częściej jest stosowana w tym pierwszym. Liczba i znaczenie argumentów jest podobne jak w przypadku funkcji `sendto()`. Różnica polega na tym, że do bufora na dane są zapisywane odebrane informacje, a w przedostatnim argumencie zapisywany jest adres strony połączenia, która te informacje nadała. Szósty argument funkcji jest wskaźnikiem na zmienną, w której będzie zapisana wielkość odebranej struktury adresu. Wartość początkowa tej zmiennej powinna być równa rozmiarowi zmiennej wskazywanej przez piąty argument. Funkcja `recvfrom()` domyślnie blokuje swoje działanie w oczekiwaniu na dane, jeśli nie zostały jeszcze wysłane. Zachowanie to można zmienić podając jako czwarty argument jej wywołania odpowiednią flagę. Jeśli ta flaga nie jest ustawiona, a funkcja zwróci `0`, to będzie to oznaczało, że komunikacja z drugą strona została zerwana. Po odebraniu danych funkcja zwraca ich rozmiar. W przypadku wyjątku zwraca wartość `-1`.
Szczegóły: `man recvfrom`

`htons()` - nazwa tej funkcji to skrót od angielskich słów *host to network short* zamienia w 16-bitowych liczbach naturalnych kolejność bajtów z tej, która obowiązuje lokalnie w urządzeniu sieciowym, na tę, która obowiązuje w całej sieci (big-endian). Funkcja ta przyjmuje jako argument liczbę 16--bitową i zwraca tę samą liczbę, przekształcając kolejność jej bajtów na big-endian. Można za jej pomocą np. przekształcić numer portu. W komputerach klasy PC ta funkcja nic nie robi, ponieważ w takim sprzęcie domyślnym porządkiem bajtów jest big-endian. Można zatem jej nie używać, pod warunkiem, że pisany program nie będzie używany na innych platformach sprzętowych.
Szczegóły: `man htons`

`htonl()` - funkcja działa podobnie jak `htons()`, ale przekształca 32-bitową liczbę naturalną, którą pobiera przez parametr.
Szczegóły: `man htonl`

`ntohs()` - nazwa funkcji pochodzi od angielskich słów *network to host short*. Zmienia ona w 16-bitowej liczbie naturalnej kolejność bajtów z tej, która obowiązuje w sieci, na tą, która obowiązuje w urządzeniu sieciowym. Liczbę do przekształcenia pobiera jako argument wywołania, a zwraca przekształconą liczbę.
Szczegóły: `man ntohs`

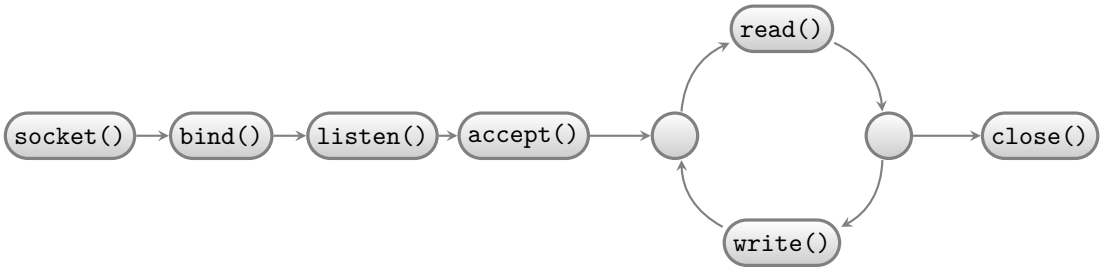
`ntohl()` - funkcja działa podobnie jak `ntohs`, ale przekształca 32-bitową liczbę naturalną.
Szczegóły: `man ntohl`

`inet_aton()` - funkcja ta przekształca ciąg znaków, zawierający adres IP w wersji 4 wyrażony w notacji „kropkowej" na ten sam rodzaj adresu wyrażony w postaci pojedynczej liczby 32-bitowej. Funkcja ta przyjmuje dwa argumenty wywołania. Pierwszym jest adres łańcucha zawierającego adres IP w notacji „kropkowej", a drugim wskaźnik na strukturę typu `struct in_addr`, w polu której zostanie zapisany adres w postaci pojedynczej liczby. Jeśli konwersja się powiedzie funkcja zwróci wartość różną od zera, a zero w przeciwnym przypadku.
Szczegóły: `man inet_aton`

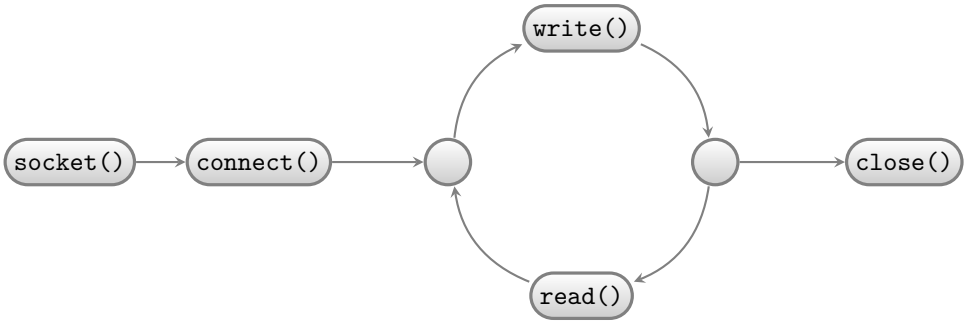
`select()` - ta funkcja służy do oczekiwania na zmianę stanu pewnej liczby deskryptorów plików lub gniazd. Przyjmuje ona pięć argumentów: pierwszym argumentem jest ogólna liczbą sprawdzanych deskryptorów, trzy środkowe argumenty są wskaźnikami na zbiory deskryptorów, a ostatni jest wskaźnikiem na strukturę typu `struct timeval`. Pierwszy wskazywany zbiór zawiera deskryptory, badane pod względem gotowości do odczytu, drugi - deskryptory badane pod względem gotowości do zapisu, a trzeci - deskryptory badane na ewentualność pojawienia się wyjątków. Zbiory te obsługiwane są z użyciem makr `FD_CLR`, `FD_SET`, `FD_ZERO` i `FD_ISSET`. Pierwsze makro usuwa podany deskryptor ze zbioru, drugie dodaje do zbioru, trzecie zeruje cały zbiór, a czwarte sprawdza, czy deskryptor należy do zbioru i jest wykorzystywane do sprawdzenia, czy zmienił się stan tego deskryptora. Ostatni argument funkcji `select()` pozwala określić czas, po jakim przerwane zostanie badanie gniazd, jeśli ich stan się nie zmienił. W Linuksie wartość struktury wskazywanej przez ten argument jest dodatkowo modyfikowana, jeśli pojawi się zmiana stanu któregoś z deskryptorów. Wartość pól tej struktury określa wówczas ile czasu upłynęło od wywołania `select()` do pojawienia się tej zmiany. Funkcja zwraca `-1` w przypadku błędu, `0` jeśli upłynął czas oczekiwania i nie pojawiło się żadne zdarzenie (żaden deskryptor nie zmienił stanu) oraz wartość większą od zera oznaczającą ile deskryptorów, spośród badanych, zmieniło stan, jeśli taka zmiana wystąpiła. Bardziej rozbudowaną funkcją, zbliżoną w działaniu do `select()` jest `pselect()`. Pozwala ona z większą precyzją określić czas zakończenia jej działania, przy czym nigdy nie modyfikuje argumentu, który określa ten czas. Dodatkowo pozwala ona określić, które sygnały podczas jej działania zostaną zablokowane.
Szczegóły: `man select`, `man pselect`

4.3. Kolejność wywołań funkcji

Opisane w poprzednim podrozdziale funkcje są wywoływane w określonej kolejności w programach serwerów i klientów, w zależności od tego, z jakich protokołów komunikacyjnych one korzystają. Diagramy 1 oraz 2 przedstawiają typową kolejność wywołań określonych funkcji, odpowiednio, w serwerze i kliencie, który komunikują się za pomocą protokołu TCP/IP.

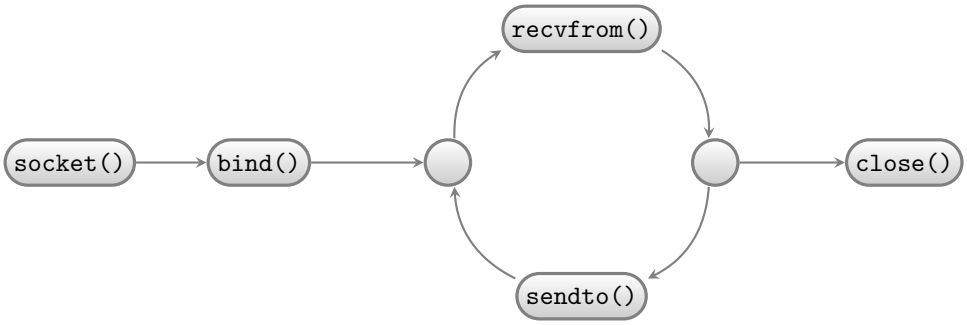


Rysunek 1: Typowa kolejność wywołań funkcji w serwerze korzystającym z protokołu TCP/IP.

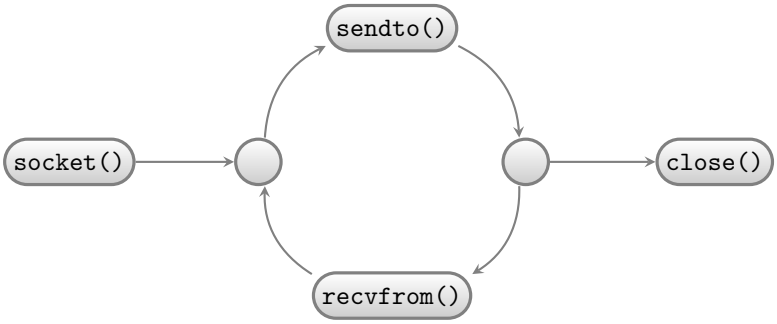


Rysunek 2: Typowa kolejność wywołań funkcji w kliencie korzystającym z protokołu TCP/IP.

Z kolei diagramy 3 i 4 obrazują typową kolejność wywołań określonych funkcji, odpowiednio, w serwerze i kliencie, którzy komunikują się z użyciem protokołu UDP/IP.



Rysunek 3: Typowa kolejność wywołań funkcji w serwerze korzystającym z protokołu UDP/IP.



Rysunek 4: Typowa kolejność wywołań funkcji w kliencie korzystającym z protokołu UDP/IP.

5. Przykład

W tym rozdziale zostaną przedstawione kody źródłowe dwóch programów (klienta i serwera), które komunikują się ze sobą za pomocą gniazd BSD z użyciem protokołu UDP/IP. Klient jednorazowo przesyła do serwera komunikat (łańcuch znaków), który ten wyświetla na ekranie.

Listing 1 zawiera kod źródłowy programu - klienta. Proszę zwrócić uwagę na liczbę włączonych plików nagłówkowych. Są one wszystkie niezbędne do prawidłowej kompilacji programu. W kodzie zdefiniowano również dwie stałe. Pierwsza (wiersz nr 9) określa numer portu, na którym serwer będzie nasłuchiwał połączenia od klienta. Jest to `1096`, który nie jest zajęty przez inne serwery. Drugim jest adres IP serwera w notacji „kropkowej". Zapis `127.0.0.1` oznacza adres lokalny komputera, który pozwala zrealizować tzw. pętlę zwrotną, czyli wszystkie komunikaty wysłane na ten adres z powrotem wracają do komputera.

Dzięki temu można m.in. lokalnie testować aplikacje sieciowe, bez konieczności używania prawdziwej sieci. W funkcji `main()` programu tworzone jest gniazdo do komunikacji za pomocą protokołu UDP/IP i wywoływana jest zdefiniowana w programie funkcja `send_message()`, a jako argument jej wywołania przekazywany jest deskryptor utworzonego gniazda. W tej funkcji inicjowane są struktury odpowiedzialne za zaadresowanie przesyłanej przez gniazdo informacji. Następnie tworzony jest (wiersz nr 25) bufor z komunikatem, który zostanie wysłany do serwera i wywoływana jest funkcja `sendto()`, która ten komunikat faktycznie przesyła. Po powrocie do funkcji `main()` zamykane jest gniazdo i kończone jest działanie programu.

```
1  #include<stdio.h>
2  #include<unistd.h>
3  #include<sys/socket.h>
4  #include<sys/types.h>
5  #include<string.h>
6  #include<netinet/ip.h>
7  #include<arpa/inet.h>
8
9  #define SERVER_PORT 1096
10 #define SERVER_IP_ADDRESS "127.0.0.1"
11
12 void send_message(int socket_descriptor)
13 {
14         struct in_addr network_address;
15
16         if(!inet_aton(SERVER_IP_ADDRESS,&network_address))
17                 perror("inet_aton");
18
19         struct sockaddr_in server_address = {
20                 .sin_family = AF_INET,
21                 .sin_port = SERVER_PORT,
22                 .sin_addr = network_address
23         };
24
25         const char *message = "Komunikat przesłany przez sieć.";
26         if(sendto(socket_descriptor,message,strlen(message),0,
27                                 (struct sockaddr *)&server_address,sizeof(server_address))<0)
28                 perror("sendto");
29 }
30
31 int main(void)
32 {
33         int socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
34         if(socket_descriptor<0)
35                 perror("socket");
36
37         send_message(socket_descriptor);
38
39         if(close(socket_descriptor)<0)
40                 perror("close");
41         return 0;
42 }
```

Listing 1: Przykładowy program klienta używającego protokołu UDP/IP.

Listing 2 zawiera kod źródłowy programu - serwera. Na jego początku włączane są wszystkie niezbędne do prawidłowej kompilacji i działania nagłówki, następnie (wiersz nr 7) definiowana jest stała określająca numer portu. Jej wartość jest taka sama jak w programie klienta. W funkcji `main()` najpierw tworzone jest gniazdo do komunikacji za pomocą protokołu UDP/IP, a potem wywoływana jest zdefiniowana w programie funkcja `name_socket()`, która przyjmuje jako argument wywołania deskryptor utworzonego gniazda. Zadaniem tej funkcji jest nadanie gniazdu nazwy, poprzez powiązanie go ze strukturą lokalnego adresu. Ta struktura jest inicjowana wewnątrz funkcji. Proszę zwrócić uwagę na sposób inicjacji jej pola `sin_addr` (wiersz nr 14). Następnie wywoływana jest funkcja `bind()`, która dokonu-

je opisywanego powiązania. Po powrocie z funkcji `name_socket()` w funkcji `main()` wywoływana jest również zdefiniowana w programie funkcja `get_and_print_message()`. Jej zadaniem jest odebranie od klienta wiadomości i wypisanie jej na ekranie. Funkcja ta przyjmuje tylko jeden argument wywołania, jakim jest deskryptor gniazda. W jej wnętrzu zadeklarowany jest (wiersz nr 24) 512-bajtowy bufor na nadesłaną przez klienta wiadomość. Deklarowana jest także zmienna na adres klienta (wiersz nr 26) i zmienna na rozmiar tej struktury (wiersz nr 27). Proszę zwrócić uwagę na sposób inicjacji tej ostatniej. Następnie wywoływana jest funkcja `recvfrom()`, która odbiera nadesłany komunikat i zapisuje go w buforze. W programie liczba bajtów zwrócona przez tę funkcję zapisywana jest w zmiennej `received_bytes`. Służy ona do sprawdzenia poprawności działania `recvfrom()`, a także do zakończenia nadesłanego ciągu znakiem końca łańcucha znaków, który nie jest przesyłany przez sieć. Po tym nadesłany komunikat jest po prostu wypisywany na ekranie (wiersz nr 34). W funkcji `main()` jest jeszcze zamykane gniazdo i kończy się wykonanie programu.

```c
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/ip.h>

#define PORT 1096

void name_socket(int socket_descriptor)
{
        struct sockaddr_in server_address = {
                .sin_family = AF_INET,
                .sin_port = PORT,
                .sin_addr = {INADDR_ANY}
        };

        if(bind(socket_descriptor,(struct sockaddr*)&server_address,sizeof(server_address))<0)
                perror("bind");
}

void get_and_print_message(int socket_descriptor)
{
        char buffer[512];

        struct sockaddr_in client_address;
        socklen_t address_length = sizeof(client_address);

        int received_bytes = recvfrom(socket_descriptor,(void *)buffer,sizeof(buffer),
                        0,(struct sockaddr*)&client_address,&address_length);
        if(received_bytes<0)
                perror("recvfrom");
        else {
                buffer[received_bytes]='\0';
                puts(buffer);
        }
}

int main(void)
{
        int socket_descriptor = socket(AF_INET, SOCK_DGRAM, 0);
        if(socket_descriptor<0)
                perror("socket");

        name_socket(socket_descriptor);
        get_and_print_message(socket_descriptor);

        if(close(socket_descriptor)<0)
                perror("close");
        return 0;
}
```

Listing 2: Przykładowy program serwera używającego protokołu UDP/IP.

Zadania

UWAGA: PROGRAMY MUSZĄ BYĆ NAPISANE Z PODZIAŁEM NA FUNKCJE Z PARAMETRAMI ORAZ MUSZĄ SPRAWDZAĆ, CZY WYWOŁYWANE PRZEZ NIE FUNKCJE Z API SYSTEMU OPERACYJNEGO NIE SYGNALIZUJĄ WYJĄTKÓW.

1. Zmodyfikuj przykładowe programy tak, aby serwer odsyłał do klienta komunikat potwierdzający odebranie komunikatu.

2. Napisz programy, które będą realizował polecenie zawarte w zadaniu pierwszym, ale w oparciu o protokół TCP/IP.

3. Napisz programy, które prześlą plik o rozmiarze większym od `1` MiB między dwoma komputerami, z użyciem protokołu TCP/IP. Sprawdź, co się stanie, jeśli plik będzie wysyłany w jednym komunikacie.

4. Napisz programy przesyłające plik o wielkości przekraczającej `1` MiB między dwoma komputerami, przy użyciu protokołu bezpołączeniowego.

5. Uzupełnij programy z pierwszego zadania tak, aby przesyłały między sobą po `10` komunikatów oraz dodatkowo wykrywały i retransmitowały zagubione pakiety. **Wskazówka:** można wykorzystać w rozwiązaniu obsługę sygnałów, w szczególności sygnał `SIGALRM`.

6. Protokół UDP/IP nie gwarantuje, że komunikaty dotrą do odbiorcy w kolejności, w jakiej zostały nadane. Napisz programy, które same o to zadbają.

7. Stwórz serwer współbieżny, który będzie obsługiwał połączenia od wielu klientów, również napisanych przez Ciebie - mogą one przesyłać np. pseudolosowe liczby do serwera, który będzie je wyświetlał na ekranie. Połączenia powinny być obsługiwane przez procesy potomne. Aby uniknąć tworzenia procesów zombie, proces macierzysty powinien ignorować sygnały o zakończeniu procesów potomnych. Użyj protokołu połączeniowego.

8. Wykonaj polecenie z poprzedniego zadania, używając tym razem wątków zamiast procesów.

9. Stwórz serwer iteracyjny, o takim samym działaniu jak serwer w zadaniu siódmym. Skorzystaj z funkcji `select()`.

10. W praktyce dosyć często tworzy się serwery, które mają charakterystykę pośrednią między współbieżnym a iteracyjnym. Taki serwer utrzymuje pewną stałą liczbę wątków, które są odpowiedzialne za obsługę połączeń. Konieczność nawiązania nowego połączenia sprawdza przy pomocy funkcji `select()`. Po nawiązaniu komunikacji jej obsługę powierza się pierwszemu wątkowi z puli, który nie jest zajęty obsługiwaniem innego połączenia. Napisz taki serwer i klientów, którzy będą do niego wysyłać komunikaty tekstowe, będące kolejnymi wierszami plików tekstowych.

11. Napisz programy, które będą podawały czasy przesyłania kolejnych pakietów przez sieć. Wielkość pakietu będzie określał użytkownik jako argument wywołania programu klienckiego. Użyj protokołu UDP/IP.

12. Napisz programy, które będą podawały czasy przesyłania kolejnych pakietów przez sieć. Wielkość pakietu będzie określał użytkownik jako argument wywołania programu klienckiego. Użyj protokołu TCP/IP.

13. Stwórz programy, które będą tworzyły strukturę *farmer-worker*. *Farmer* będzie rozsyłał liczby naturalne z przedziału od `2` do `302` do procesów typu *worker*, z których każdy będzie sprawdzał, czy otrzymana przez niego liczba jest pierwsza i odsyłał wynik do *farmera*. *Farmer* powinien współpracować z trzema procesami tego typu. Użyj protokołu bezpołączeniowego.

14. Napisz programy, o strukturze klient-serwer, które pozwolą użytkownikom wysyłać do siebie wiadomości asynchronicznie (tak jak e-mail). Serwer będzie jedynym programem działającym w trybie ciągłym. Do jego zadań będzie należało rejestrowanie nowych użytkowników, wysyłanie wiadomości do adresata, jeśli ten nawiąże połączenie oraz odbieranie wiadomości do innych użytkowników. Klient po uruchomieniu powinien wysłać serwerowi nazwę użytkownika, następnie odebrać wiadomości, które są dla niego przeznaczone i umożliwić użytkownikowi wysłanie komunikatów do innych użytkowników.