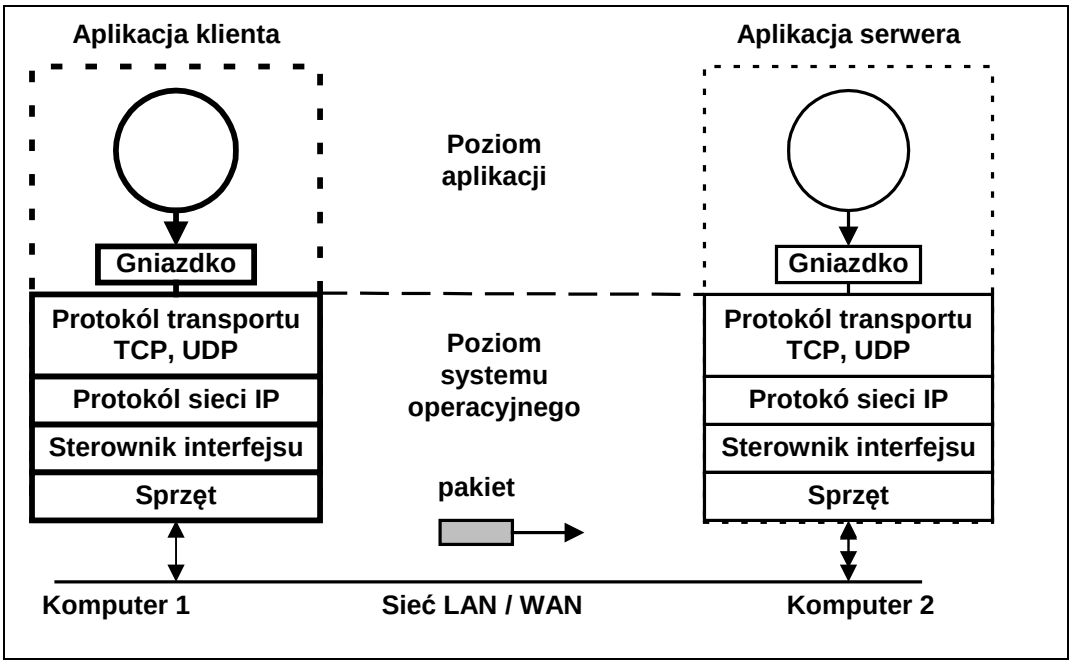
Programowanie sieciowe z użyciem interfejsu gniazdek

Spis treści:

1. Interfejs gniazdek - wprowadzenie

Jednolity interfejs API (Application Program Interface) do mechanizmów komunikacji sieciowej a także lokalnej. Wprowadzony w wersji Unixa BSD 4.2



Rys. 1-1 Ogólny schemat komunikacji sieciowej z użyciem gniazd

Główna idea gniazdek polega na użyciu do komunikacji (lokalnej i zdalnej) tego samego mechanizmu, co dostępu do plików. Jest to mechanizm oparty o deskryptory plików i funkcje **`read, write`** .

Termin gniazdko ma dwa znaczenia:

1. Biblioteka + funkcje interfejsowe (API).
2. Końcowy punkt komunikacji

Biblioteka gniazdek maskuje mechanizmy transportu sieci.

Własności gniazd:

- Gniazdo jest identyfikowane przez liczbę całkowitą nazywaną deskryptorem gniazda
- Gniazdo można nazwać i wykorzystywać do komunikacji z innymi gniazdami w tej samej domenie komunikacyjnej

Opis interfejsu gniazdek zawarty jest w dokumentacji biblioteki gnu libc: https://www.gnu.org/software/libc/manual/pdf/libc.pdf

2. Domeny komunikacji, style komunikacji i protokoły

Kiedy tworzone jest gniazdko następujące dane muszą być określone:
- Domena komunikacji
- Styl komunikacji
- Protokół

Tworzenie gniazdka

```
int socket(int domain, int typ, int protcol)
```

domain Specyfikacja domeny komunikacyjnej. Podstawowe domeny: AF_UNIX, AF_INET, AF_INET6 ,
Typ Semantyka komunikacji. Podstawowe style: SOCK_STREAM, SOCK_DGRAM, SOCK_SEQPACKET, SOCK_RAW
protcol Specyfikacja protokołu. Zwykle dla domeny i stylu jest implementowany tylko jeden protokół.

Funkcja zwraca:
> 0 Uchwyt gniazdka
-1 błąd

```
main() {
 int sock;
 sock = socked(AF_INET,SOCK_STREAM,0);
 if(sock < 0) {
   perror(”gniazdko”);
   exit(0);
 }
 . . .
}
```

Przykład 2-1 Tworzenie gniazda strumieniowego w domenie internetu

Zamykanie gniazdka

Zamknięcie gniazdka może być wykonane przez funkcje:

- **`close(int socket)`**
- **`shutdown(int socket, int how)`**

Gdzie:

`socket`	Uchwyt gniazdka
`how`	SHUT_WR – zamknięcie gniazdka do zapisu SHUT_RD – zamknięcie gniazdka do odczytu SHUT_RDWR – zamknięcie gniazdka do odczytu i zapisu

```
int res;
int s; // gniazdko
res = shutdown(s, SHUT_WR);
if (res == -1 ) perror("shutdown");
```

Listing 2-1 Przykład zamknięcia gniazdka

Działanie:

- Wymuszenie zapisu buforowanych danych
- Wysłanie znaku EOF do połączonego gniazdka
- Gdy gniazdko pozostaje niezamkniete do odczytu można odebrać potwierdzenie

Uwaga:

Gdy gniazdko s zostanie zduplikowane poleceniem **`d = dup(s)`** polecenie **`close(d)`** zamknie tylko jedno gniazdko i trzeba ponownie wykonać **`close(s)`**. Problem rozwiązuje użycie polecenia:

`shutdown(s,SHUT_RDWR)`

2.1. Domeny

Komunikacja odbywa się w pewnej domenie. Od domeny zależy sposób adresowania w sieci.

Są trzy podstawowe domeny:
- Domena internetu wersja IPv4 - AF_INET
- Domena internetu wersja IPv6 - AF_INET6
- Domena Unixa - AF_UNIX

Inne domeny:
AF_IPX – Protokół IPX Novell
AF_NETLINK – Protokół komunikacji z jądrem
AF_AX25 - Protokół komunikacji amatorskiej
AF_PACKET - Komunikacja z driverem sieciowym

Rodzina adresów AF_INET
Ta rodzina adresów umożliwia komunikację między procesami działającymi w tym samym systemie lub w różnych systemach. Używa protokołu IP w wersji 4. Adres w dziedzinie AF_INET składa się z:

- Numeru Portu
- Adresu IP maszyny

Adres IP maszyny jest 32 bitowy.

Rodzina adresów AF_INET6
Ta rodzina adresów zapewnia obsługę protokołu IP w wersji 6 (IPv6). Adres gniazda składa się z:

- Numeru Portu
- Adresu IP maszyny

Rodzina adresów AF_INET6 używa 128-bitowych (16-bajtowych) adresów maszyny.

Rodzina adresów AF_UNIX
Ta rodzina adresów umożliwia komunikację między procesami w ramach jednego systemu. Adres jest nazwą ścieżki do pozycji systemu plików.

2.2. Styl komunikacji

Interfejs realizuje następujące podstawowe style komunikacji:

- Strumienie (*ang. stream*) – SOCK_STREAM
- Datagramy (*ang. datagram*) – SOCK_DGRAM
- Protokół surowy (*ang. raw*) – SOCK_RAW

Strumienie

- Metoda strumieni zapewnia połączenie pomiędzy gniazdkami. Korekcja i detekcja błędów zapewniana jest przez system.
- Pojedynczy odczyt instrukcją **`read`** może dostarczać danych zapisanych wieloma instrukcjami **`write`** lub tylko część danych zapisanych instrukcją **`write`** po drugiej stronie połączenia.
- Aplikacja jest zawiadamiana gdy połączenie zostanie zerwane.

Datagramy

- W komunikacji datagramowej nie są używane połączenia. Każda porcja danych (datagram) adresowany jest indywidualnie. Gdy adres jest prawidłowy a połączenie sprawne, datagram jest dostarczany do adresata, ale nie jest to gwarantowane.
- Aplikacja sama musi zadbać o sprawdzenie czy dane dotarły (np. poprzez potwierdzenia).
- Granice datagramów są zachowane.

Protokół surowy

Umożliwia dostęp do protokołów niższych warstw np. ICMP.

2.3. Protokoły

Protokół jest zestawem reguł, formatów danych i konwencji które są wymagane do przesłania danych. Zadaniem kodu implementującego protokół jest:

- Zamiana adresów symbolicznych na fizyczne
- Ustanowienie połączeń
- Przesyłanie danych przez sieć

Ten sam styl komunikacji może być implementowany przez wiele protokołów.

Domena	Styl komunikacji	Protokół gniazda
AF_UNIX	SOCK_STREAM	-
	SOCK_DGRAM	-
	SOCK_SEQPACKET	Komunikacja połączeniowa, zachowuje granice pakietów
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_SEQPACKET	SCTP
	SOCK_RAW	IP, ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_SEQPACKET	SCTP
	SOCK_RAW	IP6, ICMP6

Tab. 0-1 Zestawienie parametrów gniazd

2.4. Kolejność bajtów w adresach

Sposób zapisywania danych w różnych typach maszyn może być odmienny. Dotyczy to w szczególności kolejności bajtów składających się na zmienne **`int`**.

Rys. 0-1 Sposoby wewntrznej reprezentacji liczb

Mniejsze niżej	Intel 80x86, DEC VAX
Mniejsze wyżej	Motorola 68000, Power PC

Tab. 0-2 Sposoby reprezentacji liczb w zależności od typu maszyny

Dla protokołów TCP/IP przyjęto konwencję mniejsze wyżej. Jest to tzw. Format sieciowy. Funkcje konwersji formatów sieciowego na lokalny:

```
unsigned long ntohl(unsigned long netlong)

unsigned short ntohs(unsigned short netshort)

unsigned long htonl(unsigned long hostlong)

unsigned short htons(unsigned short hostshort)
```

2.5. Adresy gniazd

Nowo utworzone gniazdo nie posiada jeszcze adresu. Aby mogło uczestniczyć w komunikacji musi mu być nadany adres. Definicja adresu zawarta jest w pliku nagłówkowym **<sys.socket.h**>.

```
stuct sockaddr {
    u_short sa_family;  // Określenie domeny komunikacji
    char sa_data[14];   // Bajty adresu
}
```

Wartościami pola **sa_family** są: AF_UNIX, AF_INET, AF_INET6

Powyższy format jest formatem ogólnym – jest on słuszny dla różnych domen (danych powyżej). W poszczególnych domenach stosowane są odmienne metody adresowania.

Nazywanie gniazdka

Gdy gniazdko jest utworzone istnieje ono w przestrzeni nazw danej domeny, ale nie ma adresu. Przypisywanie adresu odbywa się za pomocą funkcji **bind**.

```
int bind(int sock, struct sockaddr * name, int
namelen)
```

sock - Uchwyt gniazdka
name - Adres przypisany gniazdku
namelen - Długość nazwy

Funkcja zwraca:
0 Sukces
-1 Błąd

2.5.1. Dziedzina internetu AF_INET

Adres w dziedzinie AF_INET składa się z:

Adresu IP maszyny	Liczba 32 bit
Numeru Portu	Liczba 16 bit

Adres gniazda ma postać struktury **sockadr_in**.
Format adresu **sockadr_in** określony w pliku nagłówkowym **<netinet/in.h>**

```
struct sockaddr_in {
  sa_family_t sin_family;  /* AF_INET */
  in_port_t   sin_port;    /* port-format sieciowy */
  struct in_addr sin_addr; /* adres internetowy */
 };

 /* adres internetowy */
 struct in_addr {
   uint32_t s_addr; /* adres-format sieciowy  */
 };
```

Tworzenie takich gniazd odbywa się jak poniżej:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/ip.h>

tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
udp_socket = socket(AF_INET, SOCK_DGRAM, 0);
raw_socket = socket(AF_INET, SOCK_RAW, protocol);
```

Adresy internetowe są zwykle zapisywane jako czwórki rozdzielone kropkami. Każda czwórka odpowiada jednemu bajtowi.

Na przykład:

Zapis z kropką	156.17.24.42
Szesnastkowo	0x7D11182A
Dziesiętnie	2098272298

Zajętość portów można sprawdzić oglądając plik: **/etc/services**

```
echo        7/tcp
echo        7/udp
daytime     13/tcp
daytime     13/udp
netstat     15/tcp
ftp-data    20/tcp
ftp         21/tcp
ssh         22/tcp                  # SSH Remote Login Protocol
ssh         22/udp
telnet      23/tcp
smtp        25/tcp    mail
time        37/tcp    timserver
time        37/udp    timserver
nameserver  42/tcp    name          # IEN 116
tftp        69/udp
www         80/tcp    http          # WorldWideWeb HTTP
www         80/udp                  # HyperText Transfer Protocol
rtelnet     107/tcp                 # Remote Telnet
rtelnet     107/udp
pop3        110/tcp   pop-3         # POP version 3
pop3        110/udp   pop-3
sunrpc      111/tcp   portmapper    # RPC 4.0 portmapper
sunrpc      111/udp   portmapper
ntp         123/udp                 # Network Time Protocol
snmp        161/tcp                 # Simple Net Mgmt Protocol
snmp        161/udp                 # Simple Net Mgmt Protocol
```

Tab. 0-3 Fragment pliku **/etc/services**

Zakres numerów portów	Przeznaczenie
0 -1023	Porty których używać może tylko root
1024 - 5000	“Dobrze znane” porty
5001-64K	Porty efemeryczne

Tab. 0-4 Zakresy portów i ich przeznaczenie

```
 struct sockaddr_in adr_moj, adr_cli;
  int s;
  . . .
  s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
  if(s < 0) blad("socket");
  printf("Gniazdko %d utworzone\n",s);
  // Ustalenie adresu IP nadawcy
  memset((char *) &adr_moj, 0, sizeof(adr_moj));
  adr_moj.sin_family = AF_INET;
  adr_moj.sin_port = htons(PORT);
  adr_moj.sin_addr.s_addr = htonl(INADDR_ANY);
  if (bind(s, &adr_moj, sizeof(adr_moj))==-1)
      blad("bind");
```

Przykład 0-1 Przypisanie adresu do gniazda dziedziny AF_INET

2.5.2. Dziedzina internetu AF_INET6

W dziedzinie internetu AF_INET6 adres gniazda ma postać struktury **sockadr_in6**.

```
struct sockaddr_in6 {
 sa_family_t sin6_family;   /* AF_INET6     */
 in_port_t   sin6_port;     /* port number  */
 uint32_t    sin6_flowinfo; /* IPv6 flow information */
 struct in6_addr sin6_addr; /* IPv6 address */
 uint32_t   sin6_scope_id;  /* Scope ID     */
};

struct in6_addr {
  unsigned char   s6_addr[16];   /* IPv6 address */
};
```

Pole **sin6_flowinfo** jest obecnie nieużywane i ma wartość 0.
Pole **sin6_scope_id** identyfikuje zestaw interfejsów odpowiednich dla zakresu adresów określonych w polu **sin6_addr**.

Adres IP komputera (pole **s6_addr[16]**) jest 16 bajtowy (128 bitów).

2.5.3. Dziedzina Unixa AF_UNIX

Ta rodzina adresów umożliwia komunikację między procesami w ramach jednego systemu. Adres jest nazwą ścieżki do pozycji systemu plików. Występuje także jako **AF_LOCAL**.

```
#include <sys/socket.h>
#include <sys/un.h>

unix_socket = socket(AF_UNIX, type, 0);

error = socketpair(AF_UNIX, type, 0, int fd[2]);
```

Adres dziedziny **AF_UNIX** jest reprezentowany przez następującą strukturę:

```
#define UNIX_PATH_MAX    108
struct sockaddr_un {
   sa_family_t sun_family;         /* AF_UNIX */
   char sun_path[UNIX_PATH_MAX];  /* pathname */
}
```

Pole **sun_family** zawsze zawiera stałą **AF_UNIX**.

```
#include <stddef.h>
...
int make_named_socket (const char *filename) {
   struct sockaddr_un name;
   int sock;size_t size;/*Create the socket.*/
   sock = socket (PF_LOCAL, SOCK_DGRAM, 0);
   if (sock < 0){
      perror ("socket");exit (EXIT_FAILURE);

   /*Bind a name to the socket.*/
   name.sun_family = AF_LOCAL;
   strncpy (name.sun_path, filename, sizeof (name.sun_path));
   name.sun_path[sizeof (name.sun_path) - 1] = '\0';
   size = (offsetof(struct sockaddr_un, sun_path)+
            strlen (name.sun_path));
   if (bind (sock, (struct sockaddr *) &name, size) < 0) {
       perror ("bind");exit (EXIT_FAILURE);
   }
   return sock;
 }
```

Przykład 0-1 Tworzenie nazwy w domenie AF_LOCAL

Funkcja **socketpair** tworzy dwa połączone gniazdka. Ich deskryptory zawarte są w elementach tablicy **fd[0]** i **fd[1]**. Funkcja stosowana gdy procesy są w relacji macierzysty potomny (deskryptory są dziedziczone). Nie ma różnicy do którego gniazdka **fd[0]** czy **fd[1]** się pisze.

```
/* Przyklad uzycia funkcji socketpair                    */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>

int main(int argc,char *argv[]) {
    int z,n;
    int fd[2];  /* Para gniazdek */
    const char hello[] = "witam - pisze do gniazdka";
    char buf[80];
    z = socketpair(AF_UNIX,SOCK_STREAM,0,fd);
    if ( z < 0 ) {
          perror("socket");
          return 1;
    }
    printf("s[0] = %d;\n",fd[0]);
    printf("s[1] = %d;\n",fd[1]);
    if(fork() == 0) {
        close(fd[0]);
        n = write(fd[1], hello, sizeof(hello));
        printf("Proces potomny wyslal: %d znakow\n", n);
    } else { /* 2.2 ... you are the parent */
        close(fd[1]); /* Close the child file descriptor */
        n = read(fd[0], buf, sizeof(buf));
        printf("Proces macierzysty odebral: %d %s\n", n, buf);
    }
    n = wait(NULL);
    return 0;
}
```

Przykład 0-2 Użycie funkcji **socketpair**.

2.6. Funkcje dotyczące adresów

inet_addr	Konwersja z zapisu kropkowego na binarny
inet_aton	Konwersja z zapisu kropkowego na binarny
inet_ntoa	Konwersja z zapisu binarnego na kropkowy
gethostbyname	Ustalanie adresu sieciowego na podstawie nazwy
gethostname	Pobieranie nazwy komputera

Pobieranie adresu komputera

```
int gethostname(char *name, size_t len)
```

Gdzie:
name Nazwa komputera w postaci łańcucha zakończonego 0
len Maksymalna długość łancucha

Funkcja zwraca:
0 sukces
-1 błąd

Funkcja zwraca nazwę komputera na którym wykonywany jest program. Do ustawiania adresu służy funkcja:

```
int sethostname(const char *name, size_t len);
```

Wykonać ją może tylko użytkownik z przywilejami administratora.

Konwersja z zapisu kropkowego na binarny

Funkcje systemowe nie obsługują zapisu z kropką ale zapis binarny 32 bitowy zdefiniowany jako **in_addr_t**. Konwersji z adresu „kropkowego” na binarny dokonują funkcje:

- **inet_addr**
- **inet_aton**

```
in_addr_t inet_addr(char * ip_addres)
```

Gdzie:
ip_addres Zapis adresu IP z kropką w postaci łańcucha

Funkcja zwraca pole **sin_addr** części adresowej struktury **in_addr.**

```
int inet_aton(char * ip_addres, struct in_addr * inp)
```

Gdzie:

ip_addres	Zapis adresu IP z kropką w postaci łańcucha
inp	Pole **sin_addr** części adresowej struktury **in_addr**

```
int main(int argc, char * argv[]) {
   struct sockaddr_in adr_moj, adr_serw;

   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   memset((char *) &adr_serw, 0, sizeof(adr_serw));
   adr_serw.sin_family = AF_INET;
   adr_serw.sin_port = htons(PORT);
   if (inet_aton(argv[1], &adr_serw.sin_addr)==0) {
        . . .
   }
```

Przykład 0-2 Ustalanie adresu klienta za pomocą funkcji **inet_aton**

Konwersja z zapisu kropkowego na binarny dla AF_INET i AF_INET6

```
int inet_pton(int af, const char *src, void *dst);
```

Gdzie:

af	Domena AF_INET lub AF_INET6
src	Zapis adresu IP z kropką w postaci łańcucha
dst	Pole **sin_addr** części adresowej struktury **in_addr**

Konwersja z zapisu binarnego na kropkowy

Konwersji z adresu binarnego na „kropkowy” dokonuje funkcja:

```
char *inet_ntoa( struct in_addr in )
```

Gdzie:

in	Binarny zapis adresu IP

Funkcja przekształca zapis binarny adresu IP na zapis kropkowy w postaci łańcucha.

Przykład:

```
 rec = recvfrom(s, &msg, blen, 0, &adr_cli, &slen);
 if(rec < 0) blad("recvfrom()");
 printf("Odebrano kom. z %s:%d \n",
 inet_ntoa(adr_cli.sin_addr),ntohs(adr_cli.sin_port)
);
```

Przykład 0-3 Uzyskiwanie adresu kropkowego z binarnego

Ustalanie adresu sieciowego na podstawie nazwy

Aby ustalić adres IP komputera na podstawie jego nazwy należy użyć funkcji **gethostbyname**.

```
struct hostend *gethostbyname(char * hostname)
```

hostname - Nazwa komputera

Funkcja zwraca wskaźnik na strukturę której elementem jest adres IP komputera.

```
struct hostend {
     char *name;              // Oficjalna nazwa komputera
     char **h_aliases;        // Lista pseudonimów komputera
     int h_addrtype;          // Typ dziedziny (AF_INET)
     int h_length;            // Dlugość adresu (4)
     char **h_addr_list;      // Lista adresów IP komputera
}
```

Dla celów kompatybilności definiuje się h_addr jako h_addr_list[0]. Informacja otrzymywana jest z serwera nazw NIS lub lokalnego pliku **/etc/hosts.**

Funkcja zwraca:
NULL Gdy błąd
wskaźnik Gdy znaleziono adres

```
// Wywołanie – prog nazwa_komp
#include <netdb.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    unsigned int i=0;
    struct hostent *hp;
    if (argc < 2) {
        printf("Uzycie: %s hostname", argv[0]);
        exit(-1);
    }
    hp = gethostbyname(argv[1]);
    if (hp == NULL) {
       printf("gethostbyname() -blad \n");
    } else {
       printf("%s = ", hp->h_name);
       while ( hp -> h_addr_list[i] != NULL) {
          printf("%s ",
          inet_ntoa(*( struct in_addr*)
          (hp->h_addr_list[i])));
          i++;
       }
       printf("\n");
    }
}
```

Przykład 0-4 Uzyskanie adresu komputera z linii poleceń

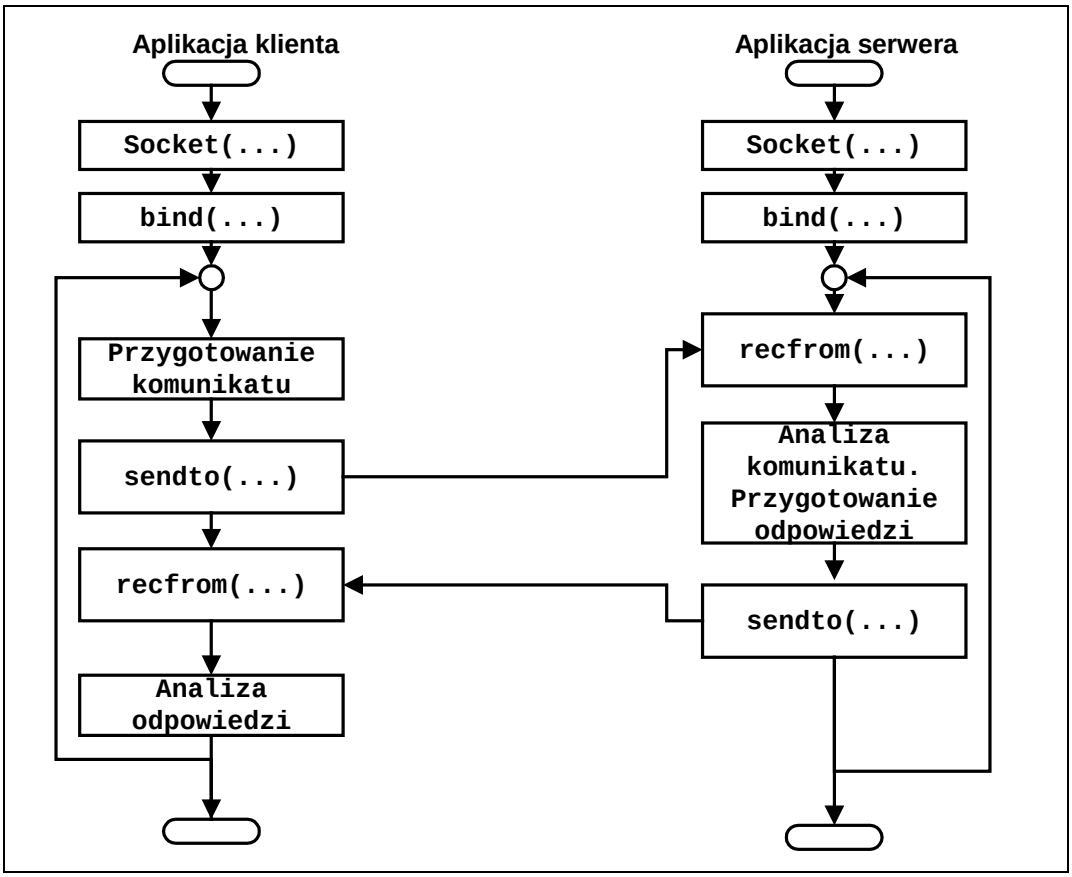
W pliku nagłówkowym **<netinet/in.h>** definiuje się adres lokalnego komputera jako INADDR_ANY.

3. Komunikacja bez kontroli połączenia

3.1. Przebieg komunikacji

Klient:
Tworzy gniazdko - socket
Nadaje gniazdku adres - bind (konieczne przy odbiorze)
Nadaje lub odbiera dane - sendto, recfrom, write, read, recv, send

Serwer:
Tworzy gniazdko - socket
Nadaje gniazdku adres - bind (konieczne przy odbiorze)
Nadaje lub odbiera dane - sendto, recfrom, write, read, recv, send



Rys. 3-1 Przebieg komunikacji bezpołączeniowej

3.2. Funkcje sendto i recfrom

Odbiór danych z gniazdka - funkcja recfrom

Funkcja **recfrom** umożliwia odczyt bajtów z gniazdka znajdującego się w stanie nie połączonym jak i połączonym.

```
int recfrom(int sock, void *buf, int nbytes,int
flags, struct sockaddr *from, int *fromlen )
```

sock	Identyfikator gniadka
buf	Bufor w którym umieszczane są odczytane bajty
nbytes	Długość bufora odbiorczego
flags	Np. MSG_OOB (dane pilne), MSG_PEEK (odbiór bez usuwania)
from	Adres skąd przyszły dane (wartość nadawana przez funkcję).
fromlen	Długość adresu (wartość nadawana przez funkcję).

Funkcja zwraca:

>0	liczbę odebranych bajtów
-1	Gdy błąd

Zapis do gniazdka - funkcja sendto

Funkcja **sendto** umożliwia wysłanie bajtów do gniazdka znajdującego się w stanie nie połączonym jak i połączonym.

```
int sendto(int sock, void *buf, int nbytes,int flags,
struct sockaddr *to, int tolen )
```

sock	Identyfikator gniadka
buf	Bufor w którym umieszczane są bajty przeznaczone do zapisu
nbytes	Liczba bajtów którą chcemy zapisać
flags	Np. MSG_OOB (dane pilne)
to	Adres docelowy
tolen	Dlugość adresu

Funkcja zwraca:

>0	Liczbę wysłanych bajtów
-1	Gdy błąd

3.3. Przykład – komunikacja w domenie internetu AF_INET

```
// Uruchomienie:  udp_serw adres
// Proces odbierajacy komunikaty - wysyla udp_cli
 #include <arpa/inet.h>
 #include <netinet/in.h>
 #include <stdio.h>
 #include <sys/types.h>
 #include <sys/socket.h>
 #include <unistd.h>
 #define BUFLEN 80
 #define KROKI 10
 #define PORT 9950

 typedef struct {
  int typ;
  char buf[BUFLEN];
} msgt;

  void blad(char *s) {
     perror(s);
     exit(1);
   }

int main(void) {
   struct sockaddr_in adr_moj, adr_cli;
   int s, i, slen=sizeof(adr_cli),snd, rec;
   int blen=sizeof(msgt);
   char buf[BUFLEN];
   msgt msg;

   // Utworzenie gniazdka
   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);
   // Ustalenie adresu IP odbiorcy
   memset((char *) &adr_moj, 0, sizeof(adr_moj));
   adr_moj.sin_family = AF_INET;
   adr_moj.sin_port = htons(PORT);
   adr_moj.sin_addr.s_addr = htonl(INADDR_ANY);
```

```
   if (bind(s, &adr_moj, sizeof(adr_moj))==-1)
       blad("bind");

   // Odbior komunikatow ------------
   for (i=0; i<KROKI; i++) {
     rec = recvfrom(s,&msg,blen,0,&adr_cli, &slen);
     if(rec < 0) blad("recvfrom()");
     printf("Odebrano z %s:%d res %d\n Typ: %d %s\n",
            inet_ntoa(adr_cli.sin_addr),
     ntohs(adr_cli.sin_port), rec,msg.typ,msg.buf);
     // Odpowiedz -----
     sprintf(msg.buf,"Odpowiedz %d",i);
     snd = sendto(s, &msg, blen, 0, &adr_cli, slen);
     if(snd < 0) blad("sendto()");
   }
   close(s);
   return 0;
}
```

Przykład 3-1 Transmisja bezpolaczeniowa serwer

```
// Uruchomienie udp_cli adres_serwera
#include <netinet/in.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#define BUFLEN 80
#define KROKI 10
#define PORT 9950

typedef struct {
  int typ;
  char buf[BUFLEN];
} msgt;

void blad(char *s) {
     perror(s);
     exit(1);
}

int main(int argc, char * argv[]) {
   struct sockaddr_in adr_moj, adr_serw, adr_x;
   int s, i, slen=sizeof(adr_serw), snd;
   int blen=sizeof(msgt),rec;
   char buf[BUFLEN];
   msgt msg;

   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);
   memset((char *) &adr_serw, 0, sizeof(adr_serw));
   adr_serw.sin_family = AF_INET;
   adr_serw.sin_port = htons(PORT);
   if (inet_aton(argv[1], &adr_serw.sin_addr)==0) {
        fprintf(stderr, "inet_aton() failed\n");
        exit(1);
   }

   for (i=0; i<KROKI; i++) {
      msg.typ = 1;
      sprintf(msg.buf, "Komunikat %d", i);
      snd = sendto(s,&msg,blen,0,&adr_serw, slen);
      if(snd < 0) blad("sendto()");
```

```
      printf("Wyslano komunikat do %s:%d   %s\n",
            inet_ntoa(adr_serw.sin_addr),
      ntohs(adr_serw.sin_port), msg.buf);
      rec=recvfrom(s,&msg,blen,0,&adr_x, &slen);
      if(rec < 0) blad("recvfrom()");
      sleep(1);
   }
   close(s);
   return 0;
}
```

Przykład 3-2 Transmisja bezpołączeniowa - klient

3.4. Przykład – komunikacja w domenie Unix'a AF_UNIX

```
 // Proces odbierajacy komunikaty - wysyla udp_cli
 #define BUFLEN 80
 #define KROKI 10
 #define PORT 9951
 #define MY_SOCK_PATH "gniazdko1"
...

int main(void) {
   struct sockaddr_un my_addr, peer_addr;
   socklen_t peer_addr_size,rlen,slen;
   int s, i, snd, rec, blen=sizeof(msgt);
   char buf[BUFLEN];
   msgt msg;
   s = socket(AF_UNIX, SOCK_DGRAM, 0);
   if (s == -1)
        blad("socket");

    memset(&my_addr, 0, sizeof(struct sockaddr_un));
                        /* Clear structure */
    my_addr.sun_family = AF_UNIX;
    strncpy(my_addr.sun_path, MY_SOCK_PATH,
            sizeof(my_addr.sun_path) - 1);

    if (bind(s, (struct sockaddr *) &my_addr,
            sizeof(struct sockaddr_un)) == -1)
        blad("bind");

   // Odbior komunikatow ------------
   for (i=0; i<KROKI; i++) {
      rec = recvfrom(s, &msg, blen, 0, &peer_addr, &rlen);
      if(rec < 0) blad("recvfrom()");
      printf("Odebrano komunikat z :%d res %d\n  Typ: %d
             %s\n",msg.typ,msg.buf);
      // Odpowiedz -----
      sprintf(msg.buf,"Odpowiedz %d",i);
      snd = sendto(s, &msg, blen, 0, &peer_addr, slen);
      if(snd < 0) blad("sendto()");
   }
   close(s);
   return 0;
}
```

Przykład 3-1 Komunikacja w dziedzinie AF_UNIX – proces serwera

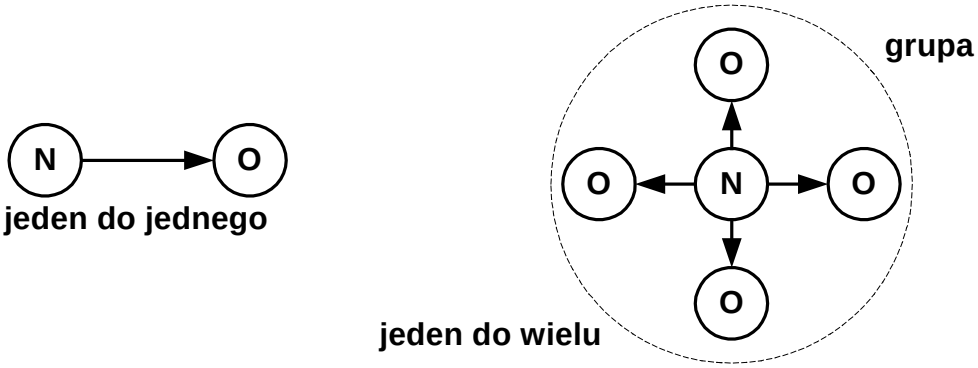
```
int main(int argc, char * argv[]) {
   struct sockaddr_in adr_moj, adr_serw, adr_x;
   int s, i, slen=sizeof(adr_serw), snd,
blen=sizeof(msgt),rec;
   char buf[BUFLEN];
   msgt msg;

   s=socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if(s < 0) blad("socket");
   printf("Gniazdko %d utworzone\n",s);
   memset((char *) &adr_serw, 0, sizeof(adr_serw));
   adr_serw.sin_family = AF_INET;
   adr_serw.sin_port = htons(PORT);
   if (inet_aton(argv[1], &adr_serw.sin_addr)==0) {
        fprintf(stderr, "inet_aton() failed\n");
        exit(1);
   }

   for (i=0; i<KROKI; i++) {
      // printf("Sending packet %d\n", i);
      msg.typ = 1;
      sprintf(msg.buf, "Komunikat %d", i);
      snd = sendto(s, &msg, blen, 0, &adr_serw, slen);
      if(snd < 0) blad("sendto()");
      printf("Wyslano komunikat do %s:%d   %s\n",
            inet_ntoa(adr_serw.sin_addr),
            ntohs(adr_serw.sin_port), msg.buf);
      rec = recvfrom(s, &msg, blen, 0, &adr_x, &slen);
      if(rec < 0) blad("recvfrom()");
      sleep(1);
   }
   close(s);
   return 0;
}
```

Przykład 3-2 Komunikacja w dziedzinie AF_UNIX – proces klienta

3.5. Komunikacja grupowa i rozgłoszenia

Dotychczas omawiana komunikacja była typu jeden do jeden.
W komunikacji klient / serwer komunikaty wymieniane były pomiędzy dwoma procesami. Jest to typ komunikacji jeden do jednego. Zdarzają się sytuacje że komunikacja obejmuje grupę procesów. Jest to typ komunikacji jeden do wielu.

Grupa – zbiór procesów działających wspólnie w sposób określony poprzez system lub użytkownika.

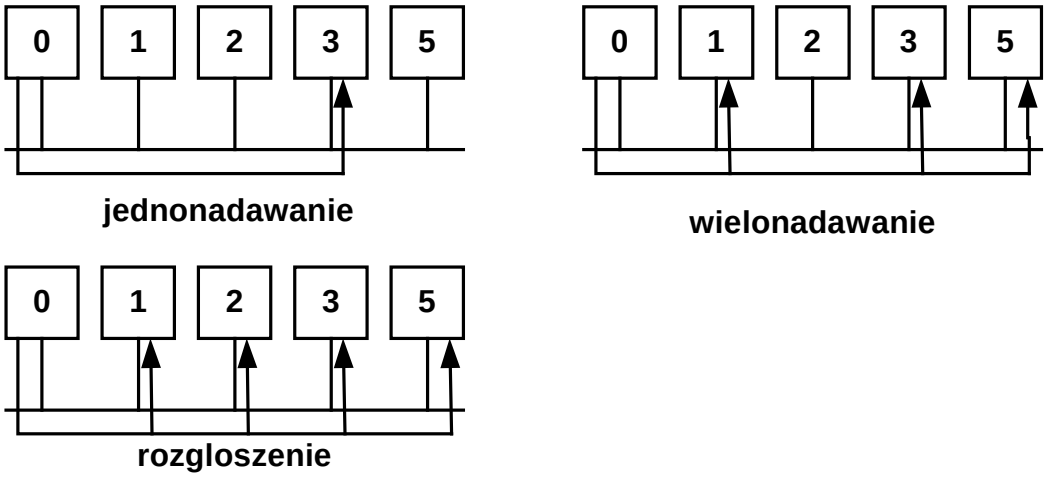


Grupy wprowadza się po to aby procesy mogły działać na zbiorach procesów traktowanych jako jeden obiekt. Proces może wysłać komunikat do grupy jako całości nie orientując się w jej składzie i położeniu pojedynczych procesów. W sieciach istnieje wsparcie dla komunikacji grupowej.

Rodzaje transmisji w sieciach:

- Jednonadawanie (*ang. Unicasting*) – komunikat otrzymuje jedna stacje w grupie
- Wielonadawanie (*ang. Multicasting*) – komunikaty otrzymują wybrane stacje w grupie
- Rozgłaszanie (*ang. Broadcasting*) – komunikaty otrzymują wszystkie stacje w grupie

Datagram rozsyłania grupowego powinien być odbierany przez te interfejsy które ich potrzebują.



Rysunek 3-1 Rodzaje komunikacji w sieciach

Rodzaj	Ipv4	Ipv6	TCP	UDP	Liczba określanych interfejsów	Liczba używanych interfejsów
Jednostkowe	+	+	+	+	jeden	jeden
Grupowe	Opcja	+	-	+	zbiór	wszystkie ze zbioru
Rozgłaszanie	+	-	-	+	zbiór	wszystkie

Tab. 3-1 Wsparcie dla różnych rodzajów komunikacji dla protokołów rodziny IP

Użycie rozgłaszania w sieciach standardu IP4 wymaga znajomości adresacji rozgłaszania. Adres IP 32 bitowy dzieli się na adres sieci (ang. network) i adres komputera (ang. host). Sposób podziału podaje poniższa tabela.

Klasa	Adres najniższy	Adres najwyższy	Bitów sieci	Bitów hosta
A	**0.0.0.0**	**127.255.255.255**	**7**	**24**
B	**128.0.0.0**	**191.255.255.255**	**14**	**16**
C	**192.0.0.0**	**223.255.255.255**	**21**	**8**
D	**224.0.0.0**	**239.255.255.255**	**28**	**brak**
E	**240.0.0.0**	**247.255.255.255**	**27**	**brak**

Tab. 3-2 Podział adresu IP4 na klasy

Klasa D używana jest do wielonadawania. 28 bitów określa adres grupy.

1	1	1	0	28 bitów – adres rozsyłania grupowego

Przyjeto konwencję że adres rozgłaszania jest adresem który posiada same jedynki w części dotyczącej hosta.

Część sieciowa może być poszerzona przez maskę sieciową (ang. netmask). W prawidłowo skonfigurowanym interfejsie sieciowym adres rozgłaszania można uzyskac przez polecenie ifconfig (jak poniżej).

```
$/sbin/ifconfig
eth0   Link encap:Ethernet  HWaddr 00:0c:29:3e:ce:3f
       inet addr:192.168.141.132  Bcast:192.168.141.255
       Mask:255.255.255.0
       inet6 addr: fe80::20c:29ff:fe3e:ce3f/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
```

Listing 3-1 Uzyskanie adresu rozgłaszania przez polecenie ifconfig

Do rozgłaszania może być też użyty specjalny adres: 255.255.255.255. Nie jest on jednak polecany, gdyż różne systemy mogą go odmiennie interpretować, w przypadku gdy komputer ma więcej interfejsów sieciowych.

```
// Uzycie: ./bclient <port_serwera>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MAXBUF 65536

int main(int argc, char*argv[]){
  int sock, status, buflen, sinlen;
  char buffer[MAXBUF];
  struct sockaddr_in sock_in;
  int yes = 1;

  sinlen = sizeof(struct sockaddr_in);
  memset(&sock_in, 0, sinlen);
  buflen = MAXBUF;

  // Utworzenie gniazdka UDP
  sock = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP);

  sock_in.sin_addr.s_addr = htonl(INADDR_ANY);
  sock_in.sin_port = htons(0);
  sock_in.sin_family = PF_INET;

  status = bind(sock, (struct sockaddr *)&sock_in, sinlen);
  printf("Bind Status = %d\n", status);

  status = setsockopt(sock,SOL_SOCKET,SO_BROADCAST,&yes,
           sizeof(int));
  printf("Setsockopt Status = %d\n", status);
  sock_in.sin_addr.s_addr=htonl(-1);

  // Wysylamy komunikat na adres rozgl. 255.255.255.255
  sock_in.sin_port = htons(atoi(argv[1])); /* port number */
  sock_in.sin_family = PF_INET;

  sprintf(buffer, "To jest rozgloszenie !!!");
  buflen = strlen(buffer);
  status = sendto(sock, buffer, buflen, 0,
          (struct sockaddr *)&sock_in, sinlen);
  printf("Sendto Status = %d\n", status);

  shutdown(sock, 2);
  close(sock);
}
```

Przykład 3-3 Rozgłoszenia – proces wysyłający (na podstawie [5])

```
// Uzycie ./bserver

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#define MAXBUF 65536
#define PORT  55555

int main() {
  int sock, status, buflen;
  unsigned sinlen;
  char buffer[MAXBUF];
  struct sockaddr_in sock_in;
  int i,yes = 1;
  sinlen = sizeof(struct sockaddr_in);
  memset(&sock_in, 0, sinlen);

  sock = socket (PF_INET, SOCK_DGRAM, IPPROTO_UDP);

  sock_in.sin_addr.s_addr = htonl(INADDR_ANY);
  sock_in.sin_port = htons(PORT);
  sock_in.sin_family = AF_INET;

  status = bind(sock, (struct sockaddr *)&sock_in, sinlen);
  printf("Bind Status = %d\n", status);

  status=getsockname(sock,(struct sockaddr*)&sock_in,
                     &sinlen);
  printf("Sock port %d\n",htons(sock_in.sin_port));

  buflen = MAXBUF;
  for(i=0;i<10;i++) {
    memset(buffer, 0, buflen);
    status = recvfrom(sock, buffer, buflen, 0,
            (struct sockaddr *)&sock_in, &sinlen);
    printf("recfrom Status = %d\n", status);
    printf("%s\n",buffer);
  }
  shutdown(sock, 2);
  close(sock);
}
```

Przykład 3-4 Rozgłoszenia – proces odbierający (na podstawie [5])

4. Komunikacja z kontrolą połączenia

Klient:

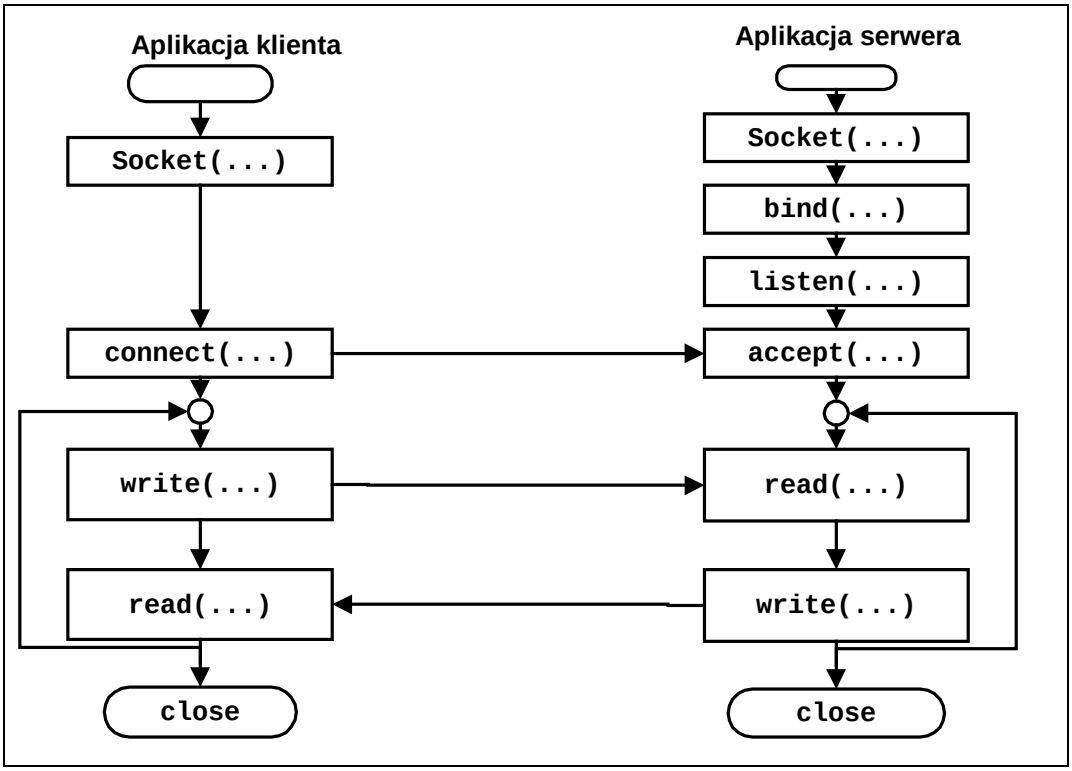
1. Tworzy gniazdko **`socket`**
2. Nadaje gniazdku adres **`bind`** (konieczne przy odbiorze)
3. Łączy się z serwerem **`connect`**
4. Nadaje lub odbiera dane **`write, read, recv, send`**

Serwer:

1. Tworzy gniazdko **`socket`**
2. Nadaje gniazdku adres **`bind`** (konieczne przy odbiorze)
3. Wchodzi w tryb akceptacji połączeń **`listen`**
4. Oczekuje na połączenia **`accept`**

Gdy połączenie zostanie nawiązane:

1. Tworzy dla tego połączenia nowe gniazdko
2. Nadaje lub odbiera dane - **`write`**, **`read`**, **`recv`**, **`send`**
3. Zamyka gniazdko



Rys. 4-1 Przebieg komunikacji z kontrolą połączenia

Połączenie ze zdalnym gniazdkiem

```
int connect(int sock, struct sockaddr *name,
            int namelen)
```

sock	Numer gniazdka
name	Nazwa (adres) komputera
len	Długość adresu

Funkcja powoduje próbę nawiązania połączenie ze zdalnym gniazdkiem wyspecyfikowanym jako adres.

Funkcja zwraca:

-1	Gdy błąd
0	Gdy nawiązano połączenie

Wprowadzenie serwera w stan gotowości do nawiązania połączenia

```
int listen(int sock, int len)
```

sock	Numer gniazdka
len	Długość kolejki oczekujących połączeń

Funkcja zwraca:

-1	Błąd
0	Sukces

Nawiązanie połączenia przez serwer

```
int accept(int sock, struct sockaddr * name,
           int *namelen)
```

sock Identyfikator gniazdka
name Adres skąd przyszło połączenie (wartość nadana przez system po wykonaniu)
namelen Długość adresu (wykonanie funkcji nadaje zmiennej wartość)

Działanie funkcji **accept**:

Wywołanie **accept** może być blokujące. Gdy przychodzi nowe połączenie następuje odblokowanie procesu bieżącego i wykonanie następujących czynności:

1. Pobranie pierwszego połączenie z kolejki oczekujących połączeń.
2. Utworzenie nowego gniazdka o identycznych własnościach jak gniazdko utworzone poleceniem **socket**.
3. Alokacja nowego deskryptora pliku dla gniazdka.
4. Nadanie wartości parametrom **name** i **namelen**.

Funkcja zwraca:
>0 Identyfikator nowego gniazdka
-1 Błąd

Odczyt z gniazdka – funkcja read

Funkcja jest używana do odbioru danych z gniazdka w trybie połączeniowym.

```
int read(int sock, void *bufor, int nbytes)
```

sock Uchwyt gniadka
bufor Bufor w którym umieszczane są przeczytane bajty
nbytes Liczba bajtów którą chcemy przeczytać.

Funkcja powoduje odczyt z gniazdka identyfikowanego przez **sock** nbytes bajtów i umieszczenie ich w buforze.
Funkcja zwraca:
> 0 Liczbę rzeczywiście odczytanych bajtów
-1 Gdy błąd

Nie ma gwarancji, że pojedyncze wywołanie funkcji odbierze dane wysłane za pomocą pojedynczego wywołania funkcji **`write`**.

Zapis do gniazdka - funkcja write

```
int write(int sock, void *bufor, int nbytes)
```

`sock`	Uchwyt gniazdka
`bufor`	Bufor w którym umieszczane są bajty przeznaczone do zapisu
`nbytes`	Liczba bajtów którą chcemy zapisać

Funkcja powoduje zapis do gniazdka identyfikowanego przez **`sock`** **`nbytes`** bajtów znajdujących buforze.

Funkcja zwraca:

`>0`	liczbę rzeczywiście wysłanych bajtów
`-1`	Gdy błąd

Odczyt z gniazdka – funkcja recv

Funkcja jest używana do odbioru danych z gniazdka w trybie połączeniowym lub bezpołączeniowym.

```
int recv(int sock, void *bufor, int nbytes, int flags)
```

sock	Identyfikator gniadka
bufor	Bufor w którym umieszczane są przeczytane bajty
nbytes	Liczba bajtów którą chcemy przeczytać.
flags	Flagi modyfikujące działanie funkcji: **MSG_OOB, MSG_PEEK, MSG_WAITALL**

Funkcja powoduje odczyt z gniazdka identyfikowanego przez **sock** **nbytes** bajtów i umieszczenie ich w buforze.

Funkcja zwraca:
> 0 – liczbę rzeczywiście przeczytanych bajtów,
- 1 – gdy błąd.

MSG_WAITALL	Funkcja czeka na tyle bajtów ile wymieniono w wywołaniu
MSG_OOB	Odbiór danych poza pasmem – znaczenie zależy od protokołu
MSG_PEEK	Dane odczytane na próbę, nie znikają z bufora

Zapis do gniazdka - funkcja send

Funkcja jest używana do zapisu danych do gniazdka w trybie połączeniowym.

```
int send(int sock, void *bufor, int nbytes, int flags)
```

sock	Identyfikator gniazdka
bufor	Bufor w którym umieszczane są bajty przeznaczone do zapisu
nbytes	Liczba bajtów którą chcemy zapisać
flags	Flagi modyfikujące działanie funkcji: **MSG_OOB, MSG_DONTROUTE, MSG_EOR**

Funkcja powoduje zapis do gniazdka identyfikowanego przez **sock** nbytes bajtów znajdujących buforze.

Funkcja zwraca:

>0	liczbę rzeczywiście wysłanych bajtów
-1	Gdy błąd

MSG_OOB	Wysyłanie danych pilnych (*ang. out of band*)
MSG_DONTROUTE	Cel diagnostyczny
MSG_EOR	Koniec rekordu

Kto się ze mną połączył?

Funkcja jest używana do testowania adresu partnera połączenia.

```
int getpeername(int sock, struct sockaddr *addr, socklent *len)
```

sock	Numer gniazdka
***addr**	Adres struktury w której jest adres partnera
len	Długość adresu

Funkcja nadaje wartość zmiennej *addr zawierającej adres partnera połączenie gniazdka sock ze zdalnym gniazdkiem.

Funkcja zwraca:

-1	Gdy błąd
0	Gdy nawiązano połączenie

```
// Gniazdka - przyklad trybu polaczeniowego
// Uzywany port 2000
// Uruchomienie: ./tcp_serw2,
// Testowanie z tcp_cli2
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#define MY_PORT 2000
#define TSIZE 32

typedef struct { // Komunikat
            int  typ;
            char tekst[TSIZE];
        } komunikat_t;

int main() {
  int sock, length;
  struct sockaddr_in server;
  int msgsock;
  int rval, res,i , cnt;
  komunikat_t msg;

  // Tworzenie gniazdka
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0) { perror("Blad gniazdka"); exit(1); }

  // Adres gniazdka
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = INADDR_ANY;
  server.sin_port = ntohs(MY_PORT);
  if(bind(sock,(struct sockaddr *)&server,
     sizeof(server))) {
       perror("Tworzenie gniazdka");  exit(1);
  }

  // Uzyskanie danych poloczenia
  length = sizeof(server);
  if (getsockname(sock, (struct sockaddr *)&server,
       &length))
```

```
  {      perror("getting socket name"); exit(1); }
  printf("Numer portu %d\n", ntohs(server.sin_port));
  // Start przyjmowania polaczen
  listen(sock, 5);
  do {
       printf("Czekam na polaczenie \n");
       msgsock = accept(sock, 0, 0);
       cnt = 0;
       if (msgsock == -1) perror("accept");
       else {
         printf("Polaczenie %d \n",msgsock);
         do { /* przesylanie bajtow ------------*/
           cnt++;
           // Odbior -----------------------------
           res=recv(msgsock,&msg,sizeof(msg),
                    MSG_WAITALL);
           if(res < 0){
              perror("Blad odczytu");
              break;
           }
           if(res == 0) {
              printf("Polaczenie zamkniete\n");
              break;
           }
           printf("Odebrano: Msg %d %s\n",
                 cnt,msg.tekst);
           msg.typ = 1;
           sprintf(msg.tekst,"Odpowiedz %d",cnt);
           printf("Wysylam: %s\n",msg.tekst);
           res =
           send(msgsock,&msg,sizeof(msg),MSG_EOR);
           // printf("Wyslano %d bajtow\n",res);
           sleep(1);
         } while(res != 0);
         close(msgsock);
       }
  } while (1);
  printf("Koniec\n");
  return(0);
}
```

Przykład 4-1 Serwer tcp_serw2.c działający w trybie z kontrolą połączenia

```
// Program wysyla i odbiera dane z/do programu
tcp_serw2
// Uzywany port 2000
// Uruchomienie: ./tcp_cli2 addres_serwera

#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <netdb.h>
#include <string.h>


#define MY_PORT 2000
#define TSIZE 32

typedef struct { // Komunikat
                 int  typ;
                 char tekst[TSIZE];
        } komunikat_t;

int main(int argc, char *argv[]){
   int sock, cnt,res;
   struct sockaddr_in server;
   struct hostent *hp, *gethostbyname();
   komunikat_t msg;
   if(argc < 2) {
    printf("Uzycie: ./tcp_cli2 adres_IP_serwera \n");
    return 1;
   }
   // Tworzenie gniazdka
   sock = socket(AF_INET, SOCK_STREAM, 0);
   if (sock < 0) {
       perror("Blad gniazdka");
       exit(1);
   }

   // Uzyskanie adresu maszyny z linii polecen
   server.sin_family = AF_INET;
   hp = gethostbyname(argv[1]);
```

```
   if (hp == 0) {
        printf("%s nieznany\n",argv[1]);
        exit(2);
   }
   memcpy(&server.sin_addr, hp->h_addr,
          hp->h_length);
   server.sin_port = htons(MY_PORT);

   // Proba polaczenia
   if (connect(sock,(struct sockaddr *) &server,
       sizeof(server)) < 0) {
       perror("Polaczenie"); exit(1);
   }
   printf("Polaczenie nawiazane\n");

   // Petla odczytu -----------------------
   cnt = 0;
   do {
    cnt++;
    // Wyslanie komunikatu ----------------------
    msg.typ = 1;
    sprintf(msg.tekst,"Komunikat %d",cnt);
    printf("Wysylam: %s\n",msg.tekst);
    res = send(sock,&msg,sizeof(msg), MSG_EOR);

    // Odbior komunikatu -------------------------
    res = recv(sock,&msg,sizeof(msg),MSG_WAITALL);
    if(res < 0) { perror("Blad odczytu"); break; }
    if(res == 0) {
      printf("Polaczenie zamkniete");   break;
    }
    printf("Odebramo %s\n",msg.tekst);
   } while( cnt < 10  );
   return 0;
}
```

Przykład 4-2 Klient tcp_cli2.c w trybie z kontrolą połączenia

4.1. Obsługa sygnałów

Pewne istotne zdarzenia powodują generowanie sygnałów.

SIGIO - W gniazdku znalazły się nowe gotowe do czytania dane
SIGURG - Do gniazdka przybyła wiadomość pilna
SIGPIPE - Połączenie zostało przerwane i niemożliwe jest pisanie do gniazdka.

4.2. Konfigurowanie gniazdek

Do konfigurowania gniazdek używa się następujących funkcji:

Testowanie bieżących opcji:

```
int getsockopt(int s, int level, int optname, void *optval, int *optlen);
```

Ustawianie bieżących opcji:

```
int setsokopt(int s, int level, int optname, void *optval, int optlen);
```

Gdzie:

s	Uchwyt gniazdka
level	Poziom na którym opcja ma działać, • dla poziomu gniazdek: SOL_SOCKET • dla poziomu TCP: IPPROTO_TCP
optname	Identyfikator opcji – zdefiniowane w pliku **sys/socket.h**
optval	Nazwa opcji
optlen	Długość opcji

Przykłady opcji:

SO_BRODCAST	Ustawienie trybu rozgłaszania
SO_RCVBUF	Ustawienie wielkości bufora odbiorczego
SO_RCVLOWAT	Minimalna liczba bajtów przy której funkcja odbioru może się zakończyć (domyślnie 1)
SO_SNDBUF	Ustawienie wielkości bufora nadawczego
SO_SNDLOWAT	Minimalna liczba bajtów przy której funkcja wysyłania może się zakończyć
SO_KEEPALIVE	Wysyłaj pakiety kontrolne
SO_RCVTIMEO	Timeout odbioru
SO_SNDTIMEO	Timeout nadawania
SO_DEBUG	Specyfikuje czy gromadzona informacja debug
SO_REUSEADDR	Specyfikuje czy w operacji bind można przypisać dwóm gniazdkom takie same adresy
SO_OOBINLINE	Dane pilne umieszczone w zwykłym strumieniu danych

Tabela 1 Niektóre opcje gniazdek

4.3. Wykorzystanie standardowej biblioteki wejścia / wyjścia

Standardowa biblioteka wejścia oferuje szerokie możliwości przetwarzania i formatowania plików. Korzysta ona ze strumieni - struktury FILE zdefiniowanej w pliku nagłówkowym **stdio.h**. Strumień otwiera się za pomocą funkcji fopen.

```
FILE *we;
we = fopen(pathname,"r");
if ( we == NULL ) {
    perror(“fopen”);
    exit(1);
}
```

Do połączenia gniazdka ze strumieniem może być użyta funkcja fdopen.

```
#include <stdio.h>

FILE *fdopen(int fildes,const char *mode);
```

Gdzie:

fildes	Identyfikator pliku – w tym przypadku gniazdka
mode	Tryb dostępu, tak sam jak w funkcji fopen

Funkcja zwraca:

!=NULL	Identyfikator strumienia połączonego z gniazdkiem
NULL	Błąd

```
int s;       /* socket */
FILE *strm;  /* stream */
s = socket(PF_INET,SOCK_STREAM,0);
…
strm = fdopen(s,"r+");
if ( strm == NULL ) {
     perror(fdopen);
     exit(1);
}
```

Listing 4-1 Połączenie gniazdka s ze strumieniem

Aby przerwać połączenie gniazdka ze strumieniem należy użyć funkcji close - zamyka ona gniazdko.

Oddzielne strumienie dla zapisu i odczytu

W powyższym przykładzie strumień **strm** może być użyty do zapisu i odczytu. Często wygodnie jest te funkcje rozdzielić (buforowanie może powodować trudności). Wykorzystana będzie funkcja dup.

```
int dup(int oldfd)
```

Funkcja **dup** tworzy kopię uchwytu **oldfd**. Nowy uchwyt, będący wolnym uchwytem o najmniejszym numerze (pierwszy wolny) zwracany jest przez funkcję **dup**.

```
int s;     /* socket */
FILE *rx;  /* read stream */
FILE *tx   /* write stream */

s = socket(PF_INET,SOCK_STREAM,0);
…
rx = fdopen(s,"r");
if ( rx == NULL ) {
    perror(fdopen1); exit(1);
}

tx = fdopen(dup(s),"w");
if ( tx == NULL ) {
    perror(fdopen2); exit(1);
}
```

Listing 4-2 Utworzenie oddzielnych strumieni do odczytu i zapisu

Strumienie mogą być zamykane niezależnie:

fclose(rx) – zamyka strumień wejściowy
fclose(tx) – zamyka strumień wejściowy
Przed zamknięciem strony do zapisu należy spowodować zapis buforów przez wykorzystanie funkcji fflush.

```
fflush(tx);  /* Flush buffer out */
```

Buforowanie

Strumienie umożliwiają buforowanie. Możliwe są następujące tryby buforowania:

- Pełne bufowanie (blokowe)
- Buforowanie linii
- Brak buforowania

```
#include <stdio.h>

int setbuf(FILE *stream,char *buf);

int setbuffer(FILE *stream, char *buf, size_t size);

int setlinebuf(FILE *stream);

int setvbuf(FILE *stream, char *buf, int mode, size_t
size);
```

Gdzie:

stream	Identyfikator strumienia
buf	Adres bufora, gdy NULL użyty będzie bufor wewnętrzny
size	Wielkość bufora
mode	Flaga – patrz tabela niżej

Flaga	Opis
_IOFBF	Pełne buforowanie
_IOLBF	Buforowanie liniami
_IONBF	Brak buforowania.

Tabela 2 Znaczenie parametru **mode** funkcji **setvbuf**

```
#include <unistd.h>
…
#define PORT 40003
#define BUFSIZE 2000

int main(void) {
     int server_sock_fd;
     int rc;
     char buf[BUFSIZE];
     int done = 0;
     struct sockaddr_in server_addr;
     int client_socket_fd;
     struct sockaddr_in client_addr;
     socklen_t client_addr_size = sizeof(client_addr);

     s_sock= socket(AF_INET, SOCK_STREAM, 0);
     if (s_sock< 0) {
          perror("Couldn't create socket");
          exit(1);
     }

     server_addr.sin_family = AF_INET;
     server_addr.sin_port = htons(PORT);
     inet_pton(AF_INET, "0.0.0.0", &server_addr.sin_addr);

     rc = bind(server_sock_fd,(struct sockaddr *) &server_addr,
          sizeof(server_addr));
     if (rc < 0) {
          perror("Couldn't bind server socket");
          exit(1);
     }

     rc = listen(server_sock_fd, 5);
     if (rc < 0) {
          perror("Couldn't listen on server socket");
          exit(1);
     }
```

```
    while (1) {
      // Oczekiwanie na polaczenie
      c_sock= accept( s_sock,(struct sockaddr*)
                  &client_addr,&client_addr_size);
      if (c_sock< 0) {
              perror("Couldn't accept connection");
              exit(1);
      }
      // Otwarcie gniazdka jako strumienia
      FILE *client_socket_fh = fdopen(c_sock,"r");
      do {
         // Odczyt linii ze strumienia
         fgets(buf, BUFSIZE, client_socket_fh);
         printf("%s", buf);
         if(strcmp(buf, "stop\r\n") == 0) done = 1;
      } while(!done);
      fclose(c_sock);
    }
    close(s_sock);
    return 0;
}
```

Przykład 4-1 Przekształcenie gniazdka w strumień – program serwera (na podstawie [4])

```
...
#define PORT 40003

int main(int argc, char *argv[]) {
     int rc;
     FILE *fh;
     int s;
     if (argc < 2) {
          fprintf(stderr, "Uzycie: client <message>\n");
          exit(1);
     }
     s = socket(AF_INET, SOCK_STREAM, 0);
     if (s < 0) {
          perror("socket"); exit(1);
     }

     struct sockaddr_in server_addr;
     server_addr.sin_family = AF_INET;
     server_addr.sin_port = htons(PORT);
     inet_pton(AF_INET, "127.0.0.1", &server_addr.sin_addr);

     rc = connect(s,(struct sockaddr *) &server_addr,
                  sizeof(server_addr));
     if (rc < 0) {
          perror("connect"); exit(1);
     }

     // Przeksztalcenie gniazdka  w strumien FILE * dla zapisu
     fh = fdopen(s, "w");

     // Wyslanie komunikatu do serwera
     fprintf(fh, "%s\r\n", argv[1]);

     // Zamkniecie gniazdk
     fclose(fh);

     return 0;
}
```

Przykład 4-2 Przekształcenie gniazdka w strumień – program klienta (na podstawie [4])

Testowanie przykładu

Na jednym terminalu uruchamiamy program serwer, na drugim uruchamiamy polecenie netstat:

```
$netstat -lt
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 0.0.0.0:sybaseanywhere  0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:aminet          0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:ssh             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:40003           0.0.0.0:*               LISTEN
tcp        0      0 localhost:ipp           0.0.0.0:*               LISTEN
tcp        0      0 localhost:smtp          0.0.0.0:*               LISTEN
```

Listing 4-3 Wynik działania polecenia netstat – widoczny proces nasłuchujący na porcie 40003

Następnie uruchamiamy polecenie telnet:

```
$telnet 127.0.0.1 40003
```

Tekst wpisany w oknie telneta będzie widoczny w terminalu serwera.

5. Serwery współbieżne

Typową sytuacją jest taka, gdy do serwera łączy się wielu klientów. Aby mogli być oni obsłużeni współbieżnie także serwer musi działać współbieżnie.

Aby zwiększyć przepustowość serwera stosuje się następujące rozwiązania:

1. Serwer wieloprocesowy - Dla każdego klienta tworzony jest proces który go obsługuje
2. Serwer wielowątkowy - Dla każdego klienta tworzony jest wątek który go obsługuje
3. Serwer asynchroniczny – Obsługuje się zdarzenia gotowości na gniazdach, wykorzystanie funkcji **`select`** lub **`poll`**

Gdzie szukamy możliwości zwiększenia przepustowości?

1. Podział na procesy lub wątki by umożliwić wykorzystanie równoległości sprzętowej (wiele rdzeni lub procesorów).
2. Eliminacja oczekiwania na gotowość klienta i urządzeń wejścia / wyjścia przez obsługę zdarzeń asynchronicznych.

5.1. Serwer wieloprocesowy

Pierwsze rozwiązanie jest najprostsze, jednak powoduje trudności gdy procesy musza się między sobą komunikować. Do komunikacji wykorzystuje się:

- Łącza
- Kolejki FIFO
- Kolejki POSIX
- Pamięć dzielona i semafory

Zaletą jest możliwość wykonywania procesów obsługi klienta na oddzielnych rdzeniach procesora lub oddzielnych procesorach. Schemat działania serwera wieloprocesowego podano poniżej:

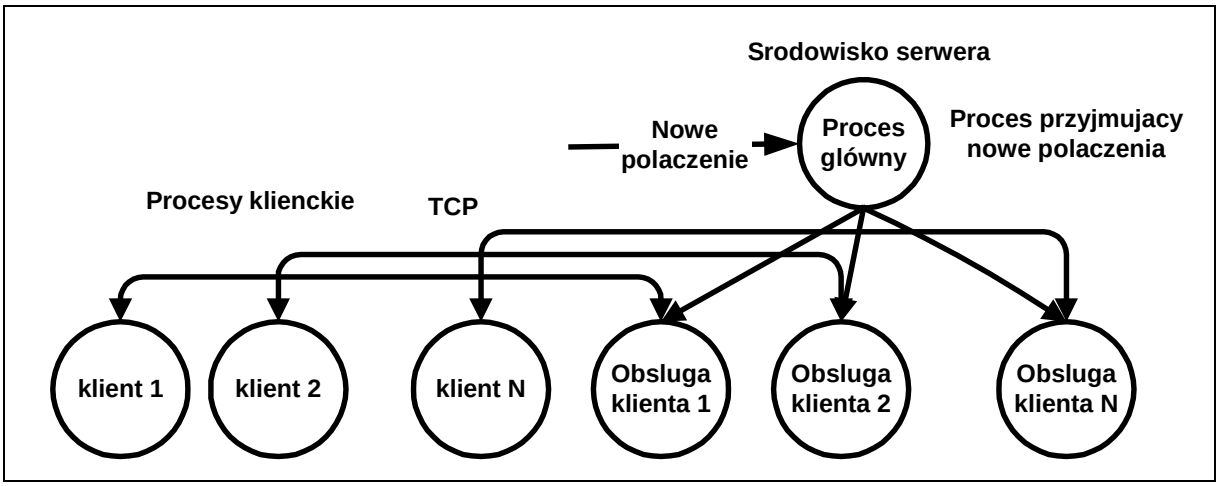
Proces główny:

1. Proces główny tworzy gniazdko - **`socket`**
2. Nadaje gniazdku adres - **`bind`** (konieczne przy odbiorze)
3. Wchodzi w tryb akceptacji połączeń - **`listen`**
4. Oczekuje na połączenia - **`accept`**
5. Gdy przychodzi nowe połączenie funkcja **`accept`** zwraca identyfikator nowego gniazdka. To gniazdko będzie używane w połączeniu z klientem. Dla połączenia tworzy się nowy proces potomny i przechodzi się do 4.

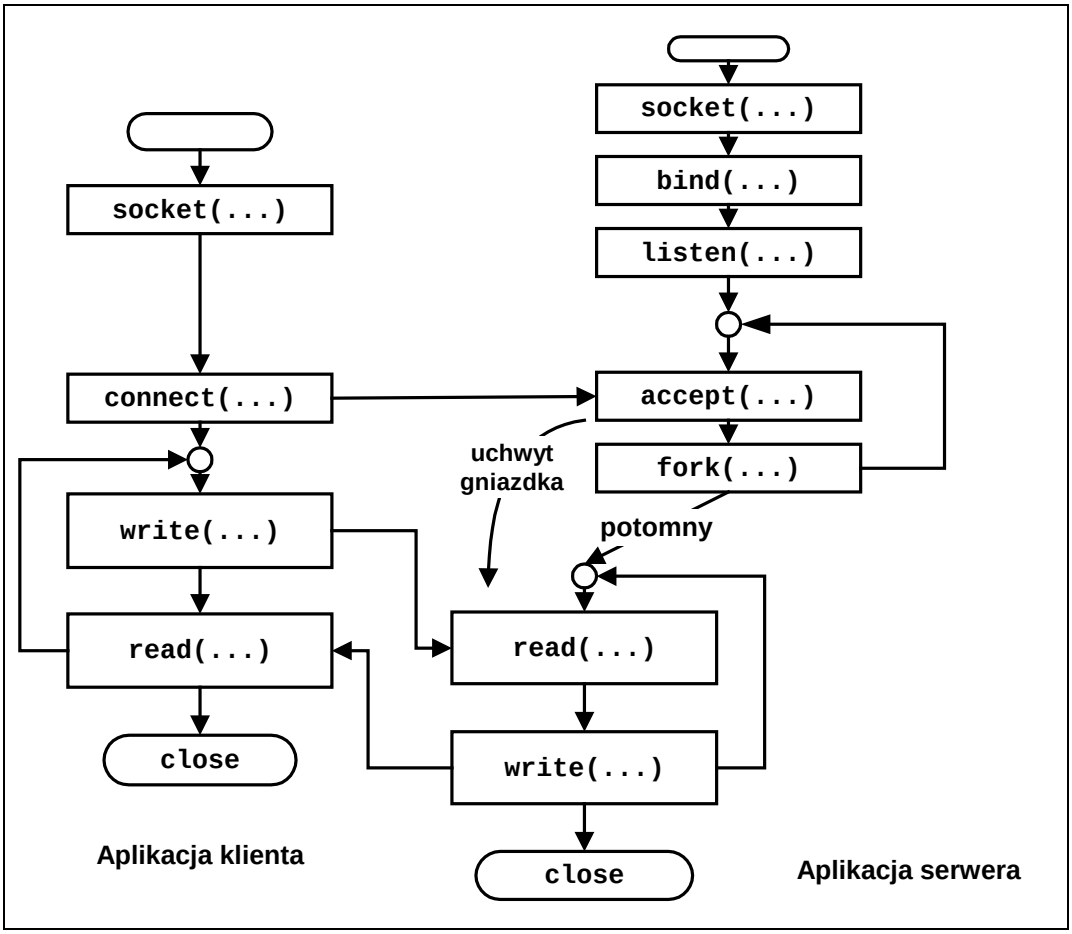
Proces obsługujący połączenie korzysta z nowego gniazdka którego numer jest przekazany jako parametr.

Proces potomny:

1. Nadaje lub odbiera dane **`- write, read, recv, send`**
2. Zamyka gniazdko



Rys. 5-1 Serwer współbieżny – każdy z klientów obsługiwany przez oddzielny proces



Rys. 5-2 Serwer współbieżny

Gdy kończone są procesy obsługujące połączenia przebywają one w stanie zombie. Proces macierzysty powinien usuwać te procesy.

Może się to odbywać w następujący sposób:

1. Obsługiwać sygnał SIGCHLD
2. W procedurze obsługi tego sygnału wykonać funkcję **`wait`**.

```
// Gniazdka - przyklad trybu polaczeniowego
// Serwer wspolbiezny
// Uzywany port 2000
// Uruchomienie: tcp_serw_wsp
#include <sys/socket.h>
#include <netinet/in.h>
#include <stdlib.h>
#define MY_PORT 2000
#define TSIZE 32

typedef struct { // Komunikat
            int  typ;
            char tekst[TSIZE];
        } komunikat_t;

main() {
  int sock, length;
  struct sockaddr_in server;
  int msgsock;
  int rval, res,i , cnt;
  komunikat_t msg;

  // Tworzenie gniazdka
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock < 0) { perror("Blad gniazdka"); exit(1); }

  // Adres gniazdka
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = INADDR_ANY;
  server.sin_port = ntohs(MY_PORT);
  if (bind(sock, &server, sizeof(server))) {
       perror("Tworzenie gniazdka");  exit(1);
  }

  // Uzyskanie danych poloczenia
  length = sizeof(server);
  if (getsockname(sock, &server, &length)) {
         perror("getting socket name"); exit(1);
  }
  printf("Numer portu %d\n", ntohs(server.sin_port));
  cnt = 0;
```

```
  // Start przyjmowania polaczen
  listen(sock, 5);
  do {
       printf("Czekam na polaczenie \n");
       msgsock = accept(sock, 0, 0);
       if (msgsock == -1) perror("accept");
       cnt++;
       printf("Polaczenie %d cnt: %d\n",msgsock,cnt);
       if(fork() == 0) { // Nowy proces ----------
         i = 0;
         do {
           // Odbior ---------------------------
           i++;
           res = read(msgsock,&msg,sizeof(msg));
           if(res < 0) { perror("Bl odcz"); break; }
           if(res == 0) {
              printf("Pol zamkn\n"); break;
           }
           printf("Pol. %d Od: Msg %d %s\n",
                  cnt,i,msg.tekst);
           msg.typ = 1;
           sprintf(msg.tekst,"Pol %d odpowiedz
                   %d",cnt,i);
           printf("Wysylam: %s\n",msg.tekst);
           res = write(msgsock,&msg,sizeof(msg));
           sleep(10);
         } while (res != 0);
         close(msgsock);
         exit(cnt);
       }
  } while (1);
  printf("Koniec\n");
}
```

Przykład 5-1 Serwer współbieżny w trybie połączeniowym. Dla każdego połączenia tworzony nowy proces.

5.2. Serwer asynchroniczny

Inną metodą zwiększenia szybkości działania serwera jest wykorzystanie schematu serwera asynchronicznego.
Rozwiązanie polega wykorzystaniu funkcji select (lub poll) która blokuje proces wykonawczy w oczekiwaniu na zdarzenia wystąpienia gotowości na którymś z nadzorowanych deskryptorów a następnie na obsłudze tych zdarzeń.

Schemat postępowania jest następujący:

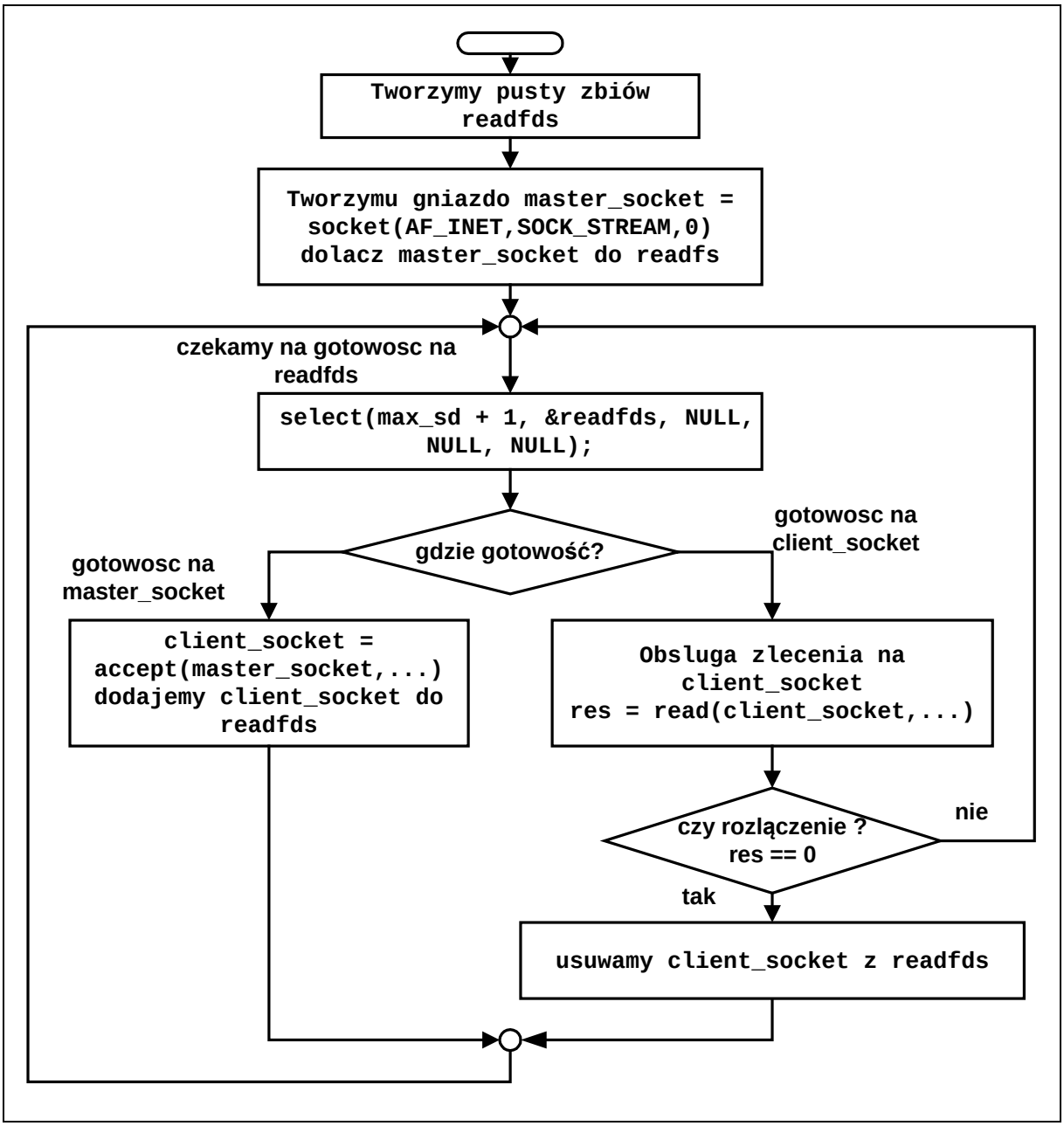
1. Tworzony jest pusty zbiór gniazd **`readfds`**
2. Tworzone jest gniazdko **`master_socket`** na którym będą przyjmowane połączenia. Do **`readfds`** dodajemy **`master_socket`**
3. Funkcja **`select`** blokuje process w oczekiwaniu na gotowość na którymś z gniazd ze zbioru S.

   ```
   select(max_sd + 1, &readfds, NULL, NULL, NULL);
   ```

4. Gdy wystąpiła gotowość na **`master_socket`** znaczy to że jest nowe połączenie.

   ```
   client_socket=accept(master_socket,...).
   ```

   Dodajemy nowe gniazdo **`client_socket`** do **`readfds`**
5. Gdy wystąpiła gotowość na którymś z gniazd klienta **`client_socket`** obsługujemy żądanie.
6. Gdy okaże się że klient się rozłączył (np. funkcja **`read(client_socket,...)`** zwróci 0) usuwamy gniazdko **`client_socket`** ze zbioru **`readfds`**
7. Przechodzimy do punktu 3



Rysunek 5-1 Schemat działania serwera asynchronicznego

```
/*
      Handle multiple socket connections with select and fd_set on Linux

      Silver Moon ( m00n.silv3r@gmail.com)
*/

...
#define TRUE   1
#define FALSE  0
#define PORT 8888

int main(int argc , char *argv[]) {
    int opt = TRUE;
    int master_socket, addrlen, new_socket, client_socket[30],
        max_clients = 30, activity, i, valread, sd, res;
    int max_sd;
    struct sockaddr_in address;

    char buffer[1025];  //data buffer of 1K

    //set of socket descriptors
    fd_set readfds;

    //a message
    char *message = "ECHO Daemon v1.0 \r\n";

    //initialise all client_socket[] to 0 so not checked
    for (i = 0; i < max_clients; i++) {
        client_socket[i] = 0;
    }

    //create a master socket
    if( (master_socket = socket(AF_INET , SOCK_STREAM , 0)) == 0)  {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }


    //type of socket created
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( PORT );

    //bind the socket to localhost port 8888
    if(bind(master_socket, (struct sockaddr *)&address,sizeof(address))<0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
      printf("Listener on port %d \n", PORT);

    //try to specify maximum of 3 pending connections for the master socket
    if (listen(master_socket, 3) < 0) {
        perror("listen");
        exit(EXIT_FAILURE);
    }

    //accept the incoming connection
    addrlen = sizeof(address);
```

```
    puts("Waiting for connections ...");

    while(TRUE)  {
        //clear the socket set
        FD_ZERO(&readfds);

        //add master socket to set
        FD_SET(master_socket, &readfds);
        max_sd = master_socket;

        // Przygotowanie zestawu obserwowanych deskryptorow readfds
        for ( i = 0 ; i < max_clients ; i++)  {
            //socket descriptor
            sd = client_socket[i];

            //if valid socket descriptor then add to read list
            if(sd > 0)  FD_SET( sd , &readfds);

            //highest file descriptor number, need it for the select
            // function
            if(sd > max_sd)   max_sd = sd;
        }

        // Oczekiwanie na gotowosc na jednym z deskryptorow
        activity = select( max_sd + 1 , &readfds , NULL , NULL , NULL);

        if ((activity < 0) && (errno!=EINTR)){ printf("select error\n");}

        // Jeżeli jest zdarzenie na glownym deskryptorze jest to nowe
        // polaczenie
        if (FD_ISSET(master_socket, &readfds))
        {
            if((new_socket = accept(master_socket, (struct sockaddr *)
                            &address, (socklen_t*)&addrlen))<0) {
                perror("accept");
                exit(EXIT_FAILURE);
            }

            // Komunikat o polaczeniu
            printf("New connection, fd is %d , ip is : %s , port : %d \n" ,
                   new_socket , inet_ntoa(address.sin_addr),
                   ntohs(address.sin_port));

            // Wyslij komunikat do nowego polaczenia
            res = send(new_socket, message, strlen(message), 0);
            if(res != strlen(message)) {
                perror("send");
            }
            puts("Komunikat wyslany");
```

```
            // Dodaj nowe gniazdko do tablicy gniazdek
            for (i = 0; i < max_clients; i++) {
                //if position is empty
                if( client_socket[i] == 0 ) {
                    client_socket[i] = new_socket;
                    printf("Adding to list of sockets as %d\n" , i);
                    break;
                }
            }
        }

        // Operacja we/wy na innym gniazdku
        for (i = 0; i < max_clients; i++)  {
            sd = client_socket[i];
            if (FD_ISSET( sd , &readfds))  {
                // Sprawdz czy nie zamnieto gniazdka
                // Jak nie czytaj komunikat
                if ((valread = read( sd , buffer, 1024)) == 0)   {
                    // Partner sie rozlaczyl, sprawdz kto
                    getpeername(sd , (struct sockaddr*)&address,
                                    (socklen_t*)&addrlen);
                    printf("Host disconnected , ip %s , port %d \n",
                    inet_ntoa(address.sin_addr), ntohs(address.sin_port));

                    //Close the socket and mark as 0 in list for reuse
                    close( sd );
                    client_socket[i] = 0;
                }

                //Echo back the message that came in
                else  {
                    // Wstaw znak konca lancucha w odebranym buforze
                    buffer[valread] = '\0';
                    // Przetwarzaj komunikat
                    . . .
                    send(sd , buffer , strlen(buffer) , 0 );
                }
            }
        }
    }
    return 0;
}
```

Przykład 5-1 Serwer asynchroniczny z wykorzystaniem funkcji select (na podstawie [4])

5.3. Serwer wielowątkowy

Serwer wielowątkowy obsługuje współbieżnie wiele połączeń od klientów. Tworzone są trzy rodzaje wątków:

1. Główny
2. Serwera
3. Wątki wykonawcze – po jednym na każde połączenie

Uwaga:
Należy zwrócić uwagę by wywołanie blokujące nie zatrzymało całego procesu w skład którego wchodzą wątki. Może się to zdarzyć na wywołaniu funkcji accept(**`server_sock_fd,...`**). Dlatego gniazdko **`server_sock_fd`** ustawione jest jako nieblokujące:

```
fcntl(server_sock_fd, F_SETFL, O_NONBLOCK);
```

Do synchronizacji zakończenia wątków zastosowano:
Muteks i zmienną warunkową.

Wątek główny:

1. Tworzy strukturę **`Server`** opisu serwera
2. Inicjuje muteks i zmienną warunkową
3. Tworzy watek serwera
4. Czeka na zakończenie wątków klientów
5. Czeka na zakończenie wątku serwera

Wątek serwera:

1. Tworzy gniazdko serwera **server_sock_fd**
2. Wiąże gniazdko do portu i ustawia je jako nieblokujące
3. Tworzy zbiór deskryptorów **readyfds** oczekujących na dane (włącza doń server_sock_fd)
4. Czeka na gotowość na dowolnym deskryptorze ze zbioru fds

   select(server_sock_fd+1, &readyfds,...)

5. Gdy jest gotowość na deskryptorze wykonujemy funkcję

   client_socket_fd = accept(server_sock_fd,...)

   Tworzymy wątek wykonawczy client i przekazujemy mu nowe gniazdko client_socket_fd
6. Sprawdzamy warunek zakończenia i przechodzimy do 4

Wątek wykonawczy:

1. Duplikuj deskryptor gniazdka **client_socket_fd**
2. Tworzy z gniazdka dwa strumienie: do odczytu i do zapisu
3. Czyta dane klienta z gniazdka
4. Odsyła dane z powrotem do klienta

```
// Kompilacja:
// $gcc –D_REENTRANT server.c –o server -lpthread
// Uruchomienie:
// $./server port
// Testowanie:
// $telnet port
#define BUFSIZE 2000

// Struktura opisu serwera: liczy ile jest watkow
// wykonawczych i czy konieczne zatrzymanie
typedef struct {
     int port;
     int num_workers;
     int shutdown_requested;

     pthread_mutex_t lock;
     pthread_cond_t cond;
} Server;
```

```
// Struktura opisu klienta
typedef struct {
     Server *s;       // Dane serwera
     int client_fd;  // gniazdko
} Worker;

// Serwer - czeka na polaczenia i startuje watki wykonawcze
void *server(void *arg);

// Watek wykonawczy - obsluguje klienta
void *worker(void *arg);

int main(int argc, char *argv[]) {
     if (argc != 2) {
          fprintf(stderr, "Uzycie: ./server <port>\n");
          exit(1);
     }
     Server *s = malloc(sizeof(Server));

     // Inicjalizacja struktury serwera
     s->port = atoi(argv[1]);
     s->num_workers = 0;
     s->shutdown_requested = 0;
     pthread_mutex_init(&s->lock, NULL);
     pthread_cond_init(&s->cond, NULL);

     // Start watku serwera
     pthread_t thr;
     pthread_create(&thr, NULL, &server, s);

     // Oczekiwanie na zakonczenie watkow wykonawczych
     pthread_mutex_lock(&s->lock);
     while (!s->shutdown_requested || s->num_workers > 0) {
          pthread_cond_wait(&s->cond, &s->lock);
     }
     pthread_mutex_unlock(&s->lock);

     // Czekaj na zakonczenie watku serwera
     pthread_join(thr, NULL);

     // Zwolnij strukture Serwera
     free(s);

     return 0;
}
```

```
// Kod watku serwera
void *server(void *arg) {
     Server *s = arg;
     int server_sock_fd;
     server_sock_fd = socket(PF_INET, SOCK_STREAM, 0);
     if (server_sock_fd < 0) {
          perror("socket");
          exit(1);
     }

     struct sockaddr_in server_addr;
     server_addr.sin_family = AF_INET;
     server_addr.sin_port = htons(s->port);
     inet_pton(AF_INET, "0.0.0.0", &server_addr.sin_addr);

     int rc = bind(server_sock_fd,(struct sockaddr *)
                   &server_addr,sizeof(server_addr));
     if (rc < 0) {
          perror("bind");
          exit(1);
     }

     // Ustanowienie gniazdka jako nieblokujące
     fcntl(server_sock_fd, F_SETFL, O_NONBLOCK);

     // Create a fd_set with the server socket file descriptor
     // (so we can use select to to a timed wait for incoming
     // connections.)
     // Utworzenie zbioru deskryptorow gniazdek oczekujacych
     // na dane. Wlaczenie gniazdka serwera
     fd_set fds;
     FD_ZERO(&fds);
     FD_SET(server_sock_fd, &fds);

     rc = listen(server_sock_fd, 5);
     if (rc < 0) {
          perror("listen");
          exit(1);
     }

     int done = 0;
     while (!done) {
          // Ustaw timeout 1 sek
          struct timeval timeout;
          timeout.tv_sec = 1;
          timeout.tv_usec = 0;

          // We want to wait for just the server socket
          fd_set readyfds = fds;
```

```
        // Zwieksz liczbe watkow wykonawczych
        pthread_mutex_lock(&s->lock);
        s->num_workers++;
        pthread_mutex_unlock(&s->lock);

        // Czekaj na polaczenie lub timeout
        int rc=select(server_sock_fd+1, &readyfds, NULL,
                      NULL, &timeout);

        if (rc == 1) {
          // Deskryptor gniazdka serwera staje się gotowy
          // Akceptujemy polaczenie bez blokowania
          int client_socket_fd;
          struct sockaddr_in client_addr;
          socklen_t client_addr_size = sizeof(client_addr);
          client_socket_fd = accept(server_sock_fd,
          (struct sockaddr*) &client_addr,&client_addr_size);

          if (client_socket_fd < 0) {
                perror("accept");
                exit(1);
          }

          // Utworzenie struktury opisu klienta, daje dostep
          // do jego gniazdka fd i struktury opisu serwera
          Worker *w = malloc(sizeof(Worker));
          w->s = s;
          w->client_fd = client_socket_fd;

          // Starujemy watek wykonawczy, nie dolaczalny
          pthread_attr_t attr;
          pthread_attr_init(&attr);
          pthread_attr_setdetachstate(&attr,
                         PTHREAD_CREATE_DETACHED);

          pthread_t thr;
          pthread_create(&thr, &attr, &worker, w);
        } else {
          // Timeout operacji select
          // Zmiejsz licznik num_workers gdyz nie utworzono
          // nowego watku
          pthread_mutex_lock(&s->lock);
          s->num_workers--;
          pthread_cond_broadcast(&s->cond);
          pthread_mutex_unlock(&s->lock);

          // Sprawdz czy ustawiony warunek zatrzymania
          // Gdy tak zatrzymaj czekanie na polaczenia
          pthread_mutex_lock(&s->lock);
          if (s->shutdown_requested) {
```

```
                    done = 1;
               }
               pthread_mutex_unlock(&s->lock);
          }
     }

     close(server_sock_fd);
     return NULL;
}

// Kod watku wykonawczego
void *worker(void *arg) {
     Worker *w = arg;
     Server *s = w->s;

     char buf[BUFSIZE];
     int read_fd = w->client_fd;
     // Duplikuj deskryptor gniazdka
     int write_fd = dup(w->client_fd);

     // Utworz dwa strumienie:
     // read_fh - do odczytu, write_fh - do zapisu
     FILE *read_fh  = fdopen(read_fd, "r");
     FILE *write_fh = fdopen(write_fd, "w");

     int done = 0;

     while (!done) {
          // Czytaj linie od klienta
          if (!fgets(buf, BUFSIZE, read_fh)) {
               break; // connection was interrupted?
          }

          // Gdy znaleziony znak \r bedzie on zastapiony
          // przez \n. Jest to potrzebne gdyz telnet po
          //klawiszu Enter wstawia dwa znaki \r\n
          char *cr = strchr(buf, '\r');
          if (cr) {
               *cr++ = '\n';
               *cr = '\0';
          }

          // Odeslanie komunikatu do klienta
          fprintf(write_fh, "%s", buf);
          fflush(write_fh);

          if (strcmp(buf, "quit\n") == 0) {
               done = 1;
          } else if (strcmp(buf, "shutdown\n") == 0) {
               done = 1;
```

```
                // Aplikacja ma byc zatrzymana
                pthread_mutex_lock(&s->lock);
                s->shutdown_requested = 1;
                pthread_cond_broadcast(&s->cond);
                pthread_mutex_unlock(&s->lock);
            }
        }

        // Zamknij strumienie klienta
        fclose(read_fh);
        fclose(write_fh);

        // Zwolnij strukture klienta
        free(w);

        // Zmiejsz licznik egzemplarzy watkow
        // wykonawczych
        pthread_mutex_lock(&s->lock);
        s->num_workers--;
        pthread_cond_broadcast(&s->cond);
        pthread_mutex_unlock(&s->lock);

        return NULL;
}
```

Przykład 5-2 Serwer wielowątkowy (na podstawie [4])

6. Testowanie stanu gniazd

Do testowania aktywności sieciowej używane jest polecenie **`netstat`** .
Polecenie **`netstat`** wyświetla:

- aktywne połączenia sieciowe TCP
- porty na których komputer nasłuchuje,
- tabelę trasowania protokołu IP
- statystyki sieci Ethernet
- statystyki protokołu IPv4 (dla protokołów IP, ICMP, TCP i UDP),
- statystyki protokołu IPv6 (dla protokołów IPv6, ICMPv6, TCP przez IPv6 i UDP przez IPv6)
- inne informacje

Niektóre opcje polecenia podano poniżej.

```
netstat [address_family_options] [--tcp|-t] [--udp|-u]
        [--udplite|-U]
        [--raw|-w] [--listening|-l] [--all|-a]
       [--numeric|-n]
      [--numeric-hosts] [--numeric-ports]
       [--numeric-users] [--symbolic|-N]
      [--extend|-e[--extend|-e]] [--timers|-o]
       [--program|-p]
      [--verbose|-v] [--continuous|-c] [--wide|-W]
```

Parametry:
-r – wyświetla tablice trasowania
-i – wyświetla interfejsy
-a – wyświetlanie wszystkich aktywnych połączeń protokołu TCP i portów protokołu TCP i UDP, na których komputer nasłuchuje
-t - wyświetla połączenia TCP i porty na których komputer nasłuchuje
-r – wyświetla tablicę trasowania jądra
-i – wyświetla interfejsy sieciowe
-u - wyswietla aktywne porty UDP
-p - wyświetla nazwy programów

Przykłady:

netstat -t	Wyswietlenie wszystkich połączeń TCP
netstat -at	Wyswietlenie wszystkich portów TCP
netstat -au	Wyswietlenie wszystkich portów UDP
netstat -lu	Wyswietlenie nasłuchujących portów UDP
netstat -lt	Wyswietlenie nasłuchujących portów TCP
netstat -pt	Wyswietlenie portów TCP z nazwami programów
netstat -s	Wyswietlenie statystyk

Tabela 3 Niektóre warianty polecenia **netstat**

```
$netstat -au
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign
Address         State
udp        0      0 *:bootpc                *:*
udp        0      0 *:36560                 *:*
udp        0      0 *:854                   *:*
```

Przykład 6-1 Działanie polecenia netstat –au **wyświetlenie otwartych portów UDP**

Literatura:

[1] Brian Ward, Jak działa Linux, Helion 2015.

[2] Robert Love, Linux Programowanie systemowe, Helion 2013.

[3] Warren Gay Linux Socket Programming by Example http://alas.matf.bg.ac.rs/manuals/lspe/0789722410

[4] Socket programming in C, http://ycpcs.github.io/cs365-spring2015/lectures/lecture15.html

[5] http://www.tack.ch/