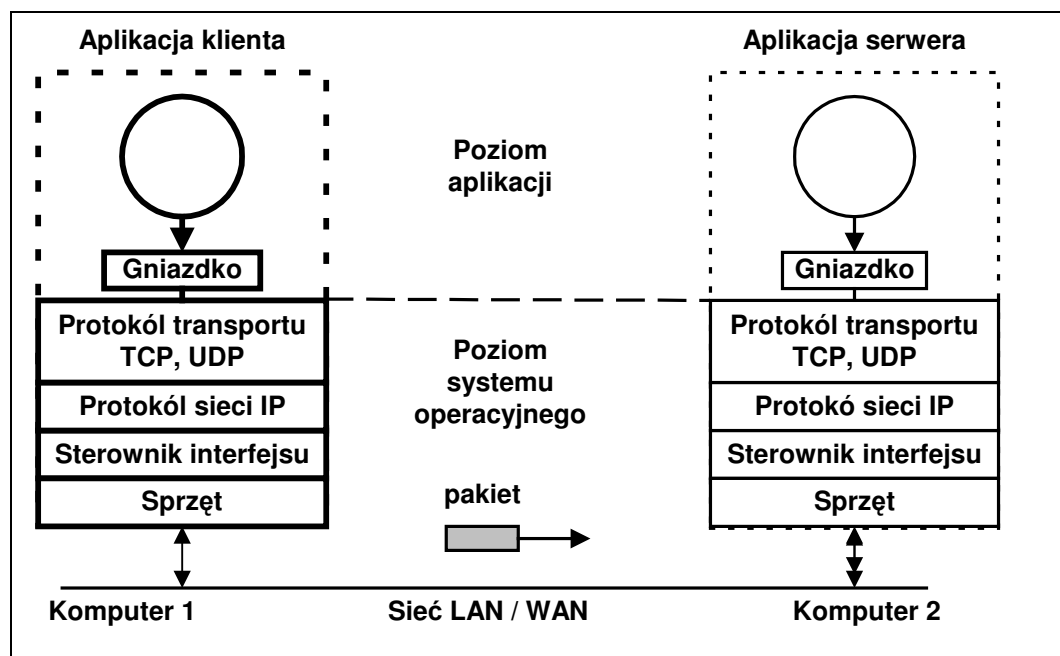


1 Interfejs gniazdek

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



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

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 gniazdek:

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

1.1 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 protocol)
```

domain Specyfikacja domeny komunikacyjnej. Podstawowe domeny:
AF_UNIX, AF_INET, AF_INET6

typ Semantyka komunikacji. Podstawowe style:
SOCK_STREAM, SOCK_DGRAM, SOCK_RAW

protocol Specyfikacja protokołu. Zwykle dla domeny i stylu jest implementowany tylko jeden protokół.

Funkcja zwraca:

- > 0 Uchwyt gniazdka
- 0 błąd

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

Przykład 1-1 Tworzenie gniazdka strumieniowego w domenie internetu

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.

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.

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	-
AF_INET	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP, ICMP
AF_INET6	SOCK_STREAM	TCP
	SOCK_DGRAM	UDP
	SOCK_RAW	IP6, ICMP6

Tab. 1-1 Zestawienie parametrów gniazd

1.2 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. 1-2 Sposoby wewnętrznej reprezentacji liczb

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

Tab. 1-2 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)`

1.3 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>`.

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

Wartości pola `sa_family` są: `AF_UNIX`, `AF_INET`, `AF_INET6`

Powyższy format jest formatem ogólnym – jest on słuszny dla różnych dziedzin (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

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 sockadr_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. 1-3 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. 1-4 Zakresy portów i ich przeznaczenie

```

struct sockaddr_in adr_moj, adr_cli;
int s, I;
. . .
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 1-2 Przepisanie adresu do gniazda dziedziny AF_INET

Dziedzina internetu AF_INET6

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

```
struct sockadr_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 bitowy (128 bajtów).

Dziedzina Unixa AF_UNIX

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

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

unix_socket = socket(AF_UNIX, type, 0);
error = socketpair(AF_UNIX, type, 0, int *sv);
```

Adres dziedziny AF_UNIX jest reprezentowany przez następującą strukturę:

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

Pole `sun_family` zawsze zawiera stałą `AF_UNIX`.

1.4 Funkcje dotyczące adresów

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

Pobieranie adresu komputera

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

Gdzie:

name Nazwa komputera w postaci łańcucha zakończony 0
len Maksymalna długość łańcucha

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_adres)
```

Gdzie:

ip_adres Zapis adresu IP z kropką w postaci łańcucha

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

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

Gdzie:

`ip_adres` 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 1-3 Ustalanie adresu klienta za pomocą funkcji `inet_aton`

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 1-4 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;        // Długość 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 1-5 Uzyskanie adresu komputera z linii poleceń

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

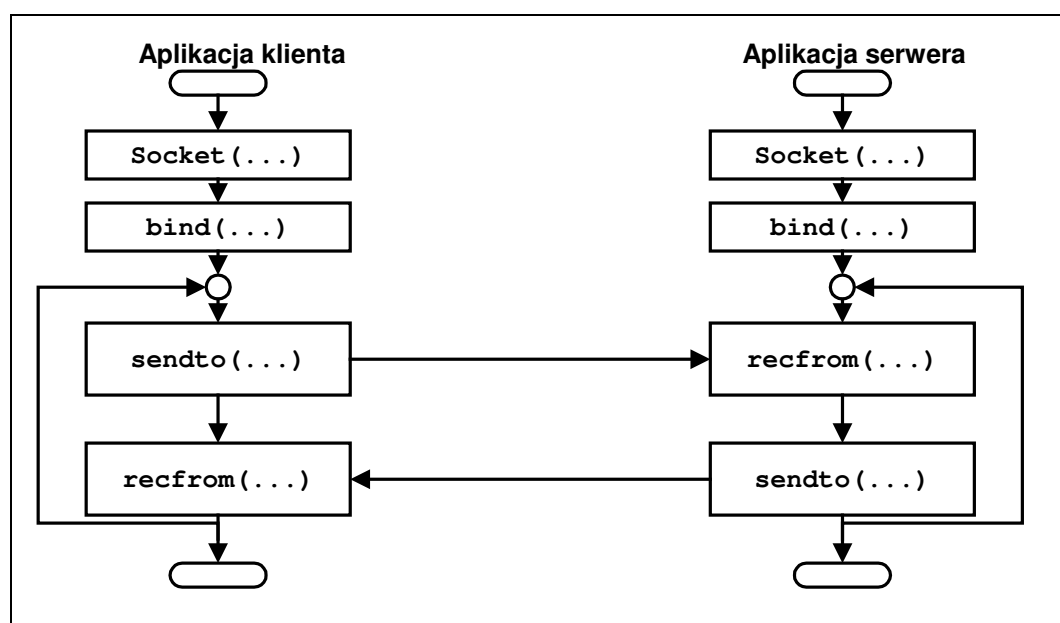
1.5 Komunikacja bez kontroli połączenia

Klient:

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

Serwer:

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



Rys. 1-3 Przebieg komunikacji bezpołączeniowej

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 gniazdka
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 przyszedły 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 gniazdka
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 Długość adresu

Funkcja zwraca:

>0 Liczbę wysłanych bajtów
-1 Gdy błąd

```
// Uruchomienie:  udp_serw adres
// Proces odbierający komunikaty - wysyła 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 1-6 Transmisja bezpołączeniowa 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 BUFLLEN 80
#define KROKI 10
#define PORT 9950

typedef struct {
    int typ;
    char buf[BUFLLEN];
} 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[BUFLLEN];
    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 1-7 Transmisja bezpołączeniowa - klient

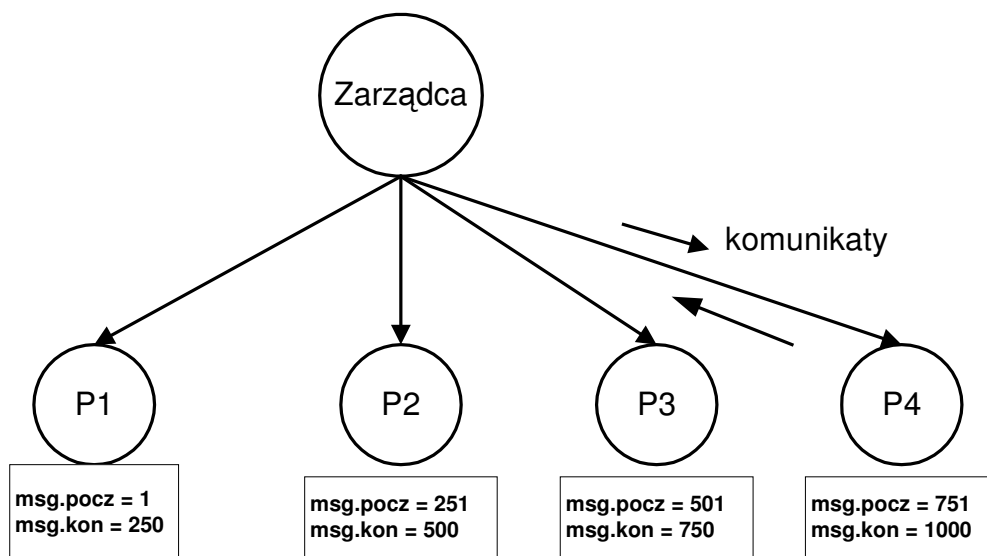
1.6 Przykład – znajdowanie liczb pierwszych

Znaleźć liczby pierwsze w zakresie od 2 do N na P procesorach

Gdy $N = 100$ a $P = 4$

`pocz[4] = { 2, 26, 51, 76 };`

`kon[4] = { 25, 50, 75, 100};`

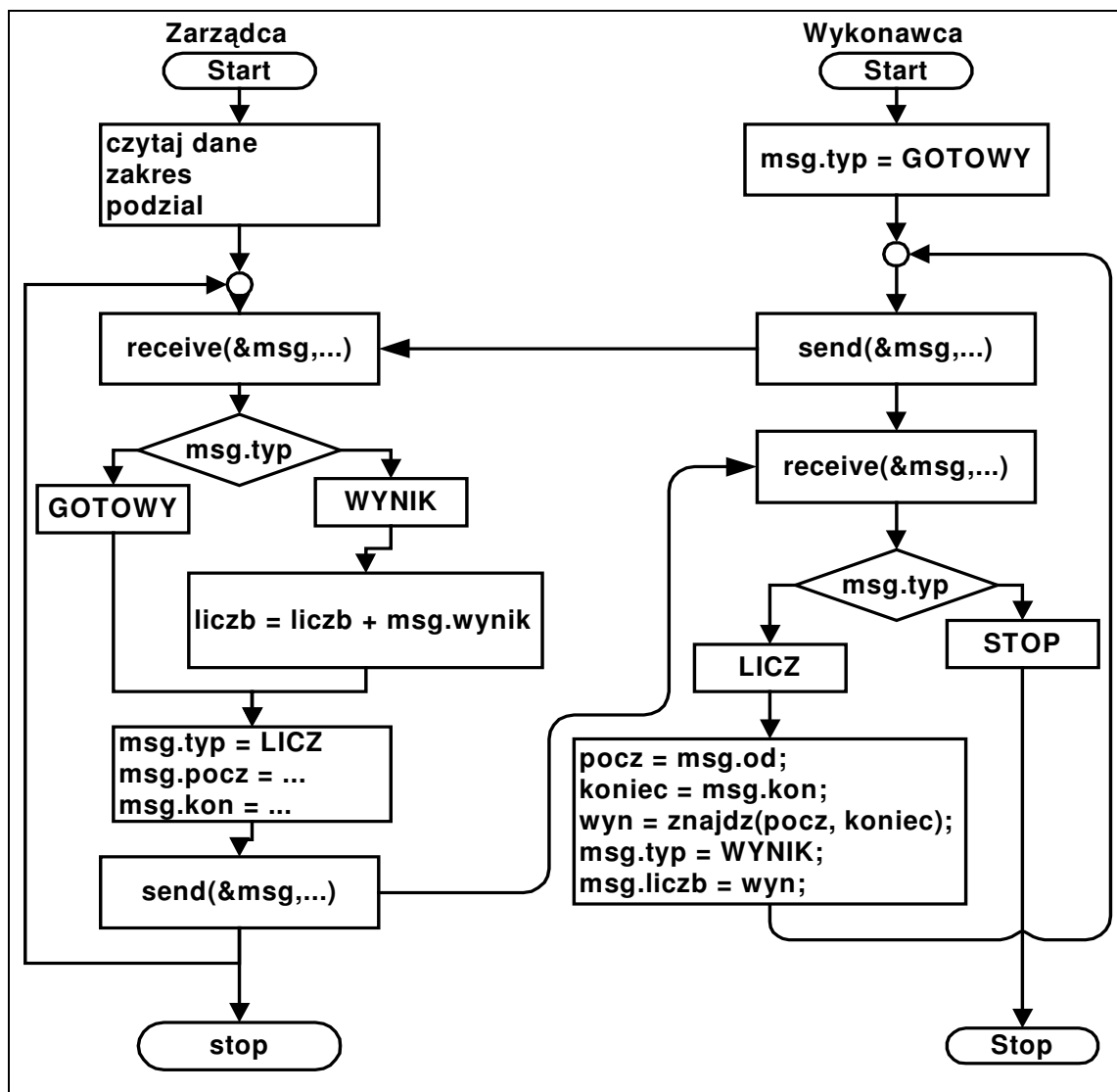


Polecenie	<code>msg.typ</code>	Opis
GOTOWY	1	Wykonawca zgłasza gotowość do pracy
WYNIK	2	Wykonawca zgłasza wynik – ile liczb w przedziale
LICZ	3	Zlecenie wykonania obliczeń w zakresie <code>msg.pocz</code> do <code>msg.kon</code>
STOP	4	Zlecenie zatrzymania

```

struct { // Komunikat
    int typ; // Typ polecenia
    int numer; // Numer procesu
    int pocz; // Początek zakresu obliczeń
    int kon; // Koniec zakresu obliczeń
    int liczb; // Ile liczb pierwszych w zakresie
} msg

```



```
// Plik wspolny
#define SIZE sizeof(struct sockaddr_in)
#define MY_PORT 2001

#define GOTOWY 1
#define WYNIK 2
#define LICZ 3
#define STOP 4

typedef struct { // Komunikat
    int typ;
    int numer;
    int pocz;
    int kon;
    int liczb;
} komunikat_t;
```

Plik zawierający definicje "wspolny.h"

```
// Znajdowanie liczb pierwszych - zarzadca
// Usuchomienie: lp-serw zakres liczba_wezlow
// Program wspolpracuje z progr. lp-cli
// Komunikacja gniazdka UDP
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include "wspolny.h"

main(int argc, char *argv[]) {
    int sock, length, i = 0;
    struct sockaddr_in name;
    struct sockaddr_in nadawca;
    int cnt, res, wynik, liczb=0;
    komunikat_t msg;
    int dljad = SIZE;
    int podzial = atoi(argv[2]); // liczba wezlow
    int delta=0, start=1;
    delta = atoi(argv[1])/podzial-1;

    // Utworzenie gniazdka typu datagram
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("Blad tworzenia gniazdka"); exit(1);
    }
}
```



```
// Ustalenie adresu punktu odbiorczego
name.sin_family = AF_INET;
name.sin_addr.s_addr = INADDR_ANY;
name.sin_port = htons(MY_PORT);
if(bind(sock, &name, SIZE)) {
    perror("Bład tworzenia adresu"); exit(1);
}

do {
    // Odbior zgłoszenia -----
    res = recvfrom(sock, &msg, sizeof(msg), 0, &nadawca, &dlnad);
    if(res < 0) perror("Odbior");
    printf("Odebrano: %d %d \n", res, msg.typ);
    if((msg.typ == GOTOWY) || (msg.typ == WYNIK)) {
        // Zlecamy prace
        msg.typ = LICZ;
        msg.pocz = delta*i;
        msg.kon = delta*(i+1) - 1 ;
        msg.numer = i;
        printf("Zlecenie od: %d do: %d\n", msg.pocz, msg.kon);
        if(i >= podzial) break;
        i=i+1;
    }

    if(msg.typ == WYNIK) { // Odbieramy wynik
        liczb = wynik + msg.liczb;
        printf("liczb: %d\n", liczb);
    }

    // Wyslamy polecenie -----
    res = sendto(sock, &msg, sizeof(msg), 0, &nadawca, dlnad);
    if(res < 0) {
        perror("send");
    }
    printf("wyslano %d\n", res);
} while(1);
printf("Liczb pierwszych: %d\n", liczb);
close(sock);
}
```

Znajdowanie liczb pierwszych – proces zarządzający

```
// Znajdowanie liczb pierwszych - program wykonawczy
// Usuchomienie: lp-cli adres_zarzadcy moj_numer
// Program wspolpracuje z progr. lp-serw
// Komunikacja gniazdka UDP
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#include "wspolny.h"

main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp, *gethostbyname();
    struct sockaddr nadawca;
    struct sockaddr_in klient;
    int cnt, res, moj_numer, i=0;
    int dlnad = SIZE;
    komunikat_t msg;
    moj_numer = atoi(argv[2]);

    // Utworzenie gniazdka typu internet, datagram
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock < 0) {
        perror("Blad tworzenia gniazdka"); exit(1);
    }
    printf("Gniazdko %d\n", sock);

    // Ustalenie adresu gniazdka tego procesu ---
    klient.sin_family = AF_INET;
    klient.sin_addr.s_addr = INADDR_ANY;
    klient.sin_port = INADDR_ANY;
    if(bind(sock, &klient, SIZE)) {
        perror("Blad tworzenia adresu"); exit(1);
    }
}
```

```
// Ustalenie adresu docelowego -----
hp = gethostbyname(argv[1]);
if (hp == 0) {
    printf("%s: unknown host\n", argv[1]);
    exit(2);
}
memcpy(&name.sin_addr, hp->h_addr, hp->h_length);
name.sin_family = AF_INET;
name.sin_port = htons(MY_PORT);
printf("Adres docelowy: %s\n", inet_ntoa(name.sin_addr));

msg.typ = GOTOWY;
msg.numer = moj_numer;
msg.liczb = 0;

for(cnt=0; cnt<10; cnt++) {
    // Wysylanie komunikatu / zgloszenia -----
    printf("Wyslano komunikat: %d\n",msg.typ);
    res = sendto(sock, &msg, sizeof(msg), 0, &name,
                sizeof(name));
    if(res < 0) {
        perror("Wysylanie"); break;
    }
    printf("Wyslano %d %d\n",res,msg.typ);

    // Odbior zlecenia -----
    res = recvfrom(sock, &msg, sizeof(msg), 0, &nadawca, &dlnad);
    if(res < 0) perror("Odbior");
    printf("Odebrano %d %d z %d\n",res,msg.typ,msg.numer);
    if(msg.typ == LICZ) { // Obliczenia
        printf("Obliczenia od %d do %d\n",msg.pocz,msg.kon);
        msg.liczb = znajdz(msg.pocz,msg.kon);
        printf("Znaleziono %d liczb\n",msg.liczb);
        msg.typ = WYNIK;
    }
    if(msg.typ == STOP) { // zakonczenie
        break;
    }
}
close(sock);
}
```

```
// Procedura stwierdza ile liczb pierwszych jest w przedziale
int znajdz(int od, int kon) {
    int i, j, liczb = 0, pierwsza;
    for(j = od; j < kon; j++) {
        pierwsza = 1;
        for(i=2; i*i <= j; i++) {
            if((j%i) == 0) {
                pierwsza = 0;
                break;
            }
        }
        if(pierwsza) liczb++;
    }
    return liczb;
}
```

Znajdowanie liczb pierwszych – proces wykonawczy

1.7 Transmisja z kontrolą połączenia

Klient:

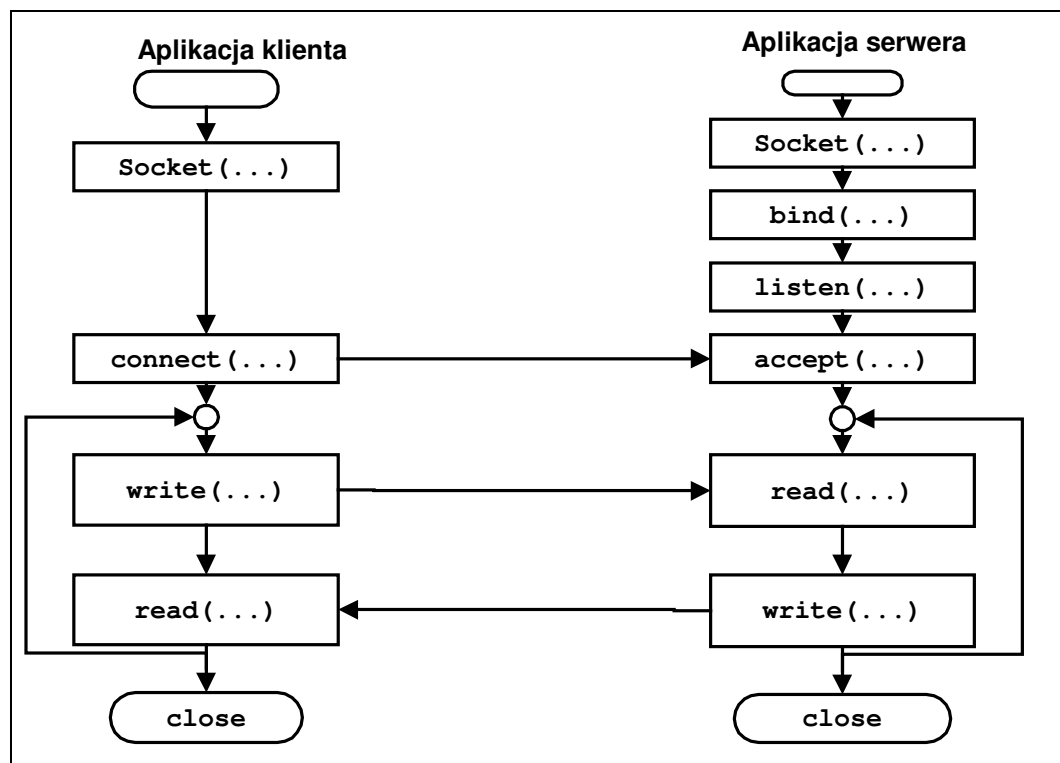
1. Tworzy gniazdko **socket**
2. Nadaje gniazdku adres **bind** (konieczne przy odbiorze)
3. Łączy się z serwerem **connect**
4. Nadaje lub odbiera dane **write, read, recv, send**

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. 1-4 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 gniazdka

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

```
// Gniazdko - przyklad trybu polaczeniowego
// Uzywany port 2000
// Uruchomienie: tcp-serw
#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 gniazdko
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) { perror("Blad gniazdko"); exit(1); }

    // Adres gniazdko
    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 gniazdko"); exit(1);
    }

    // Uzyskanie danych polaczenia
    length = sizeof(server);
    if (getsockname(sock, &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 { /* przesyłanie bajtów -----*/
            cnt++;
            // Odbior -----
            res = read(msgsock, &msg, sizeof(msg));
            if(res < 0) {
                perror("Bład odczytu"); break;
            }
            if(res == 0) {
                printf("Pol zamkn\n"); break;
            }
            printf("Odeb: Msg %d %s\n", cnt, msg.tekst);
            msg.typ = 1;
            sprintf(msg.tekst, "Odpowiedz %d", cnt);
            printf("Wysylam: %s\n", msg.tekst);
            res = write(msgsock, &msg, sizeof(msg));
            sleep(1);
        } while (res != 0);
        close(msgsock);
    }
} while (1);
printf("Koniec\n");
}
```

Przykład 1-8 Serwer tcp_serw.c działający w trybie z kontrolą połączenia

```
// Program odbiera dane od programu tcp-serw
// uruchomionego na wezle addr. Uzywany port 2000
// Uruchomienie: tcp-cli addr
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdlib.h>
#define MY_PORT 2000
#define TSIZE 32
```

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

main(int argc, char *argv[]){
    int sock, cnt, res;
    struct sockaddr_in server;
    struct hostent *hp, *gethostbyname();
    komunikat_t msg;

    // Tworzenie gniazdka
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) {
        perror("Bład 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, &server, sizeof(server)) < 0) {
        perror("Polaczenie"); exit(1);
    }
    printf("Polaczenie nawiazane\n");

    // Petla odczytu -----
    cnt = 0;
    do {
        cnt++;
        // memset(&msg, 0, sizeof(msg));
        // Wyslanie komunikatu -----
        msg.typ = 1;
        sprintf(msg.tekst, "Komunikat %d", cnt);
```

```
printf("Wysylam: %s\n",msg.tekst);
res = write(sock,&msg,sizeof(msg));

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

Przykład 1-9 Klient tcp_cli.c w trybie z kontrolą połączenia

1.8 Obsługa sygnałów

Pewne istotne zdarzenia powodują generowanie sygnałów.

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

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

Przykłady opcji:

SO_BROADCAST	Ustawienie trybu rozgłaszania
SO_RCVBUF	Ustawienie wielkości bufora odbiorczego
SO_SNDBUF	Ustawienie wielkości bufora nadawczego
SO_KEEPALIVE	Wysyłaj pakiety kontrolne
SO_RCVTIMEO	Timeout odbioru
SO_SNDTIMEO	Timeout nadawania

1.10 Serwer współbieżny

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

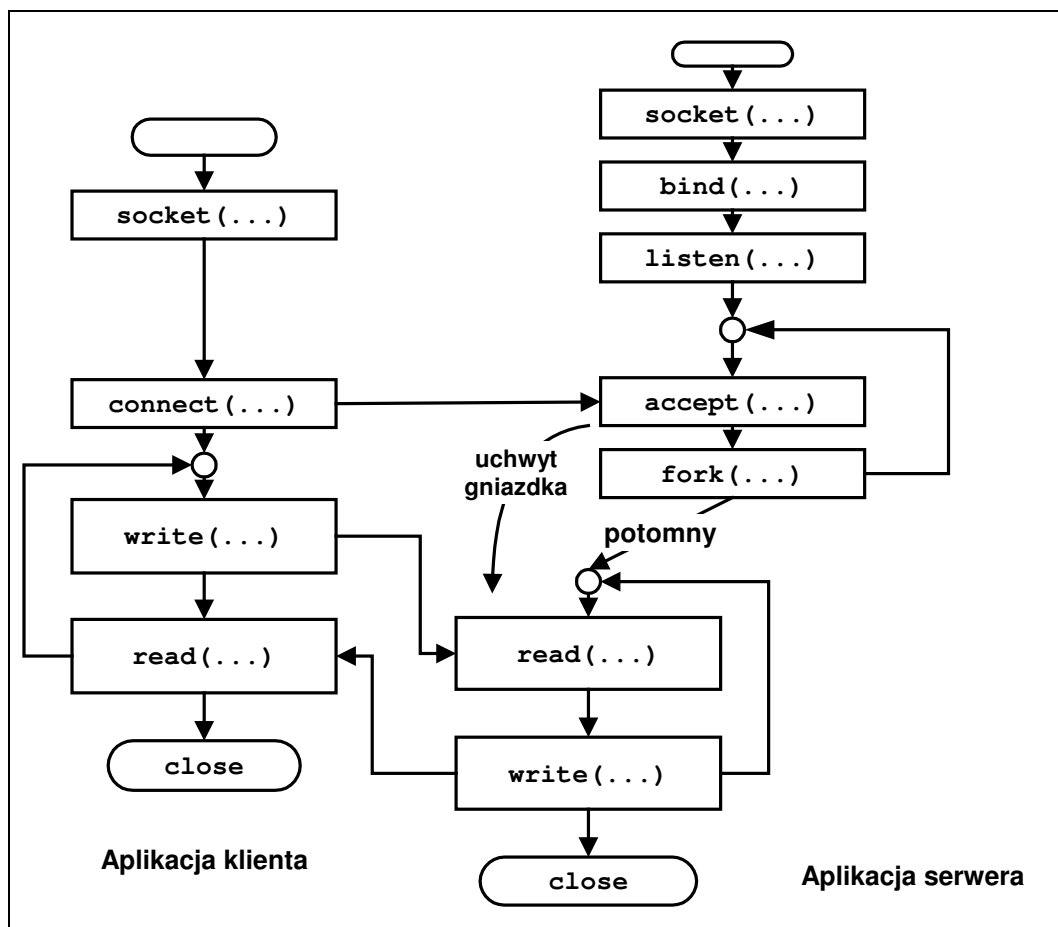
Schemat działania serwera współbieżnego

1. 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 gniazdko. To gniazdko będzie używane w połączeniu z klientem. Dla połączenia tworzy się nowy proces i przechodzi się do 4.

Proces obsługujący połączenie:

Korzysta z nowego gniazdko którego numer jest przekazany jako parametr

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

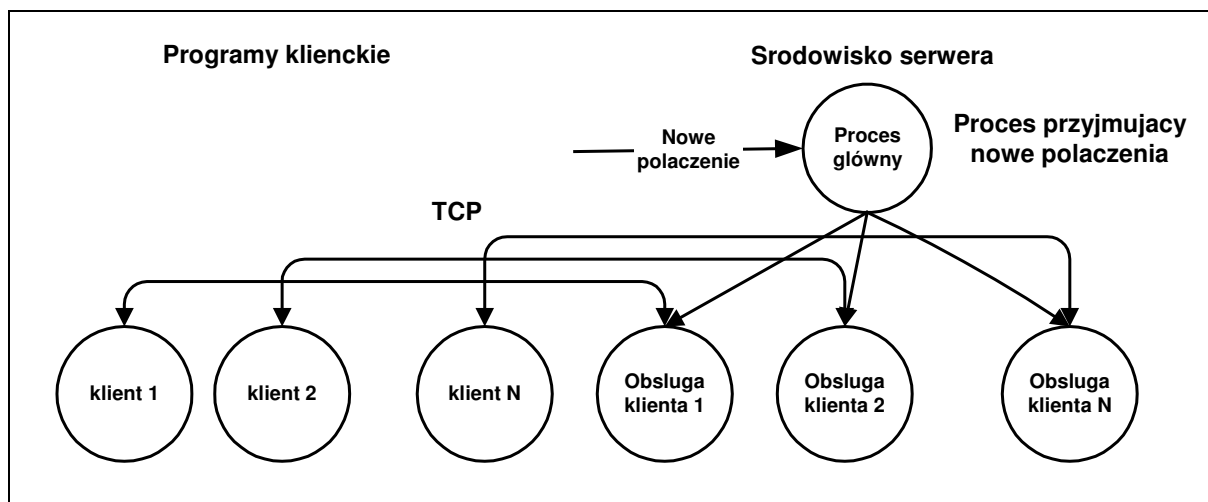


Rys. 1-5 Serwer współbieżny

Gdy kończą się 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**.



Rys. 1-6 Serwer współbieżny – każdy z klientów obsługiwany przez oddzielny proces

```
// Gniazdko - 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 gniazdko
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock < 0) { perror("Blad gniazdko"); exit(1); }

    // Adres gniazdko
    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 gniazdko"); 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 1-10 Serwer współbieżny w trybie połączeniowym. Dla każdego połączenia tworzony nowy proces.

1.11 Wykorzystanie interfejsu gniazdek do budowy aplikacji równoległych

Aby zbudować aplikację równoległą wykonywalną na klastrze należy dysponować dwoma narzędziami:

- Uruchamianie procesów na zdalnym węźle
- Zapewnienie komunikacji ze zdalnymi procesami

Uruchamianie procesów na zdalnym węźle można zrealizować za pomocą usługi rsh lub ssh.

Zapewnienie komunikacji ze zdalnymi procesami można zrealizować za pomocą interfejsu gniazdek BSD

1.12 Uruchamianie programu na zdalnym węźle

Usługa rshd

Dostępna na BSD, Solaris, Linux, nie jest to mechanizm POSIX.

Gdy na odległym komputerze uruchomiony jest demon **rshd** to z lokalnego komputera można wykonać na zdalnym polecenie **rsh** lub funkcję **rcmd**. Istnieją też implementacje tej usługi dla Windows

```
rsh    [host] [-l nazwa_użytkownika] [-n] [polecenie]
```

host	określa przy użyciu adresu IP lub nazwy host zdalny (komputer), na którym należy wykonać polecenie wskazane przez parametr polecenie.
-l nazwa_użytkownika	Nazwa użytkownika
-n	przekierowuje dane wejściowe polecenia rexec do urządzenia NUL. Zapobiega to wyświetlaniu wyników polecenia na komputerze lokalnym.
polecenie	Określa polecenie, które należy wykonać na komputerze zdalnym.

rcmd – wykonanie polecenia na zdalnym komputerze

```
rcmd(char **ahost, int inport, const char *locuser,  
const char *remuser, const char *cmd, int *fd2p)
```

ahost	Nazwa komputera zdalnego
inport	Numer portu na którym rezyduje serwer usługi rsh
locuser	Identyfikator użytkownika na maszynie lokalnej
remuser	Identyfikator użytkownika na maszynie zdalnej
cmd	Polecenie do wykonania
fd2p	0 – standardowy kanał kontroli ~0 – dodatkowy kanał kontroli

Usługa rexec

Gdy na odległym komputerze uruchomiony jest demon **rexcd** to z lokalnego komputera można wykonać na zdalnym polecenie rexec.

```
rexec [host] [-l nazwa_użytkownika] [-n] [polecenie]
```

host	określa przy użyciu adresu IP lub nazwy host zdalny (komputer), na którym należy wykonać polecenie wskazane przez parametr polecenie.
-l nazwa_użytkownika	Nazwa użytkownika
-n	przekierowuje dane wejściowe polecenia rexec do urządzenia NUL. Zapobiega to wyświetlaniu wyników polecenia na komputerze lokalnym.
polecenie	Określa polecenie, które należy wykonać na komputerze zdalnym.
/?	Wyświetla help

Podobnie istnieje funkcja rexec(...) wykonywana z poziomu języka C.

Usługa rsh ma niski stopień zabezpieczenia. Obecnie w jej miejsce stosuje się usługę ssh opartą na szyfrowaniu niesymetrycznym. Usługa ta ma wysoki stopień bezpieczeństwa jest jednak trudniejsza w konfiguracji.