



TIN

Techniki Internetowe

zima 2019

Grzegorz Blinowski
Instytut Informatyki
Politechniki Warszawskiej



Plan wykładów

- 2 Intersieć, ISO/OSI, protokoły sieciowe, IP
- 3 Protokół IP i prot. transportowe: UDP, TCP
- 4 Model klient-serwer, techniki progr. serwisów**
- 5 Protokoły aplikacyjne: telnet, ftp, smtp, nntp, inne
- 6 HTTP
- 7, 8 HTML, XML
- 9, 10, 11 Aplikacje WWW, CGI, sesje, serwery aplikacji
serwlety, integracja z backendem SQL
- 12 Aspekty zaawansowane: wydajność,
przenośność, skalowalność; klastering
- 13 XML, RDF, SOAP, WSDL, ontologie
- 14 Wstęp do zagadnień bezpieczeństwa
(IPSec, VPN, systemy firewall)
oraz aspekty kryptograficzne (DES, AES, RSA,
PGP, S/MIME), tokeny i akceleracja sprzętowa

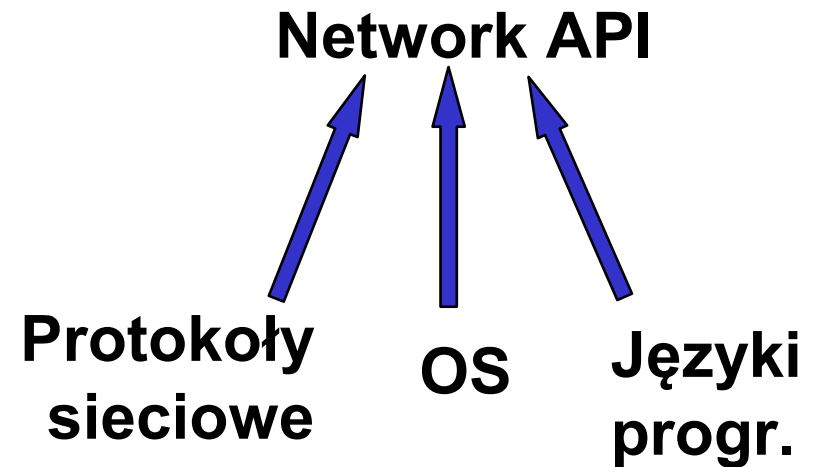


Gniazda BSD (sockets)



Gniazda BSD

- API powinien być maksymalnie niezależny od systemu operacyjnego, protokołów, języków programowania
- Ciekawostka: oryginalny API sieciowy w Unixie (BBN - 1981) bazował wyłącznie na std. funkcjach `open()`, `read()`, etc.
- Po raz pierwszy interfejs gniazd wprowadzono w BSD Unix 4.1c (1982)

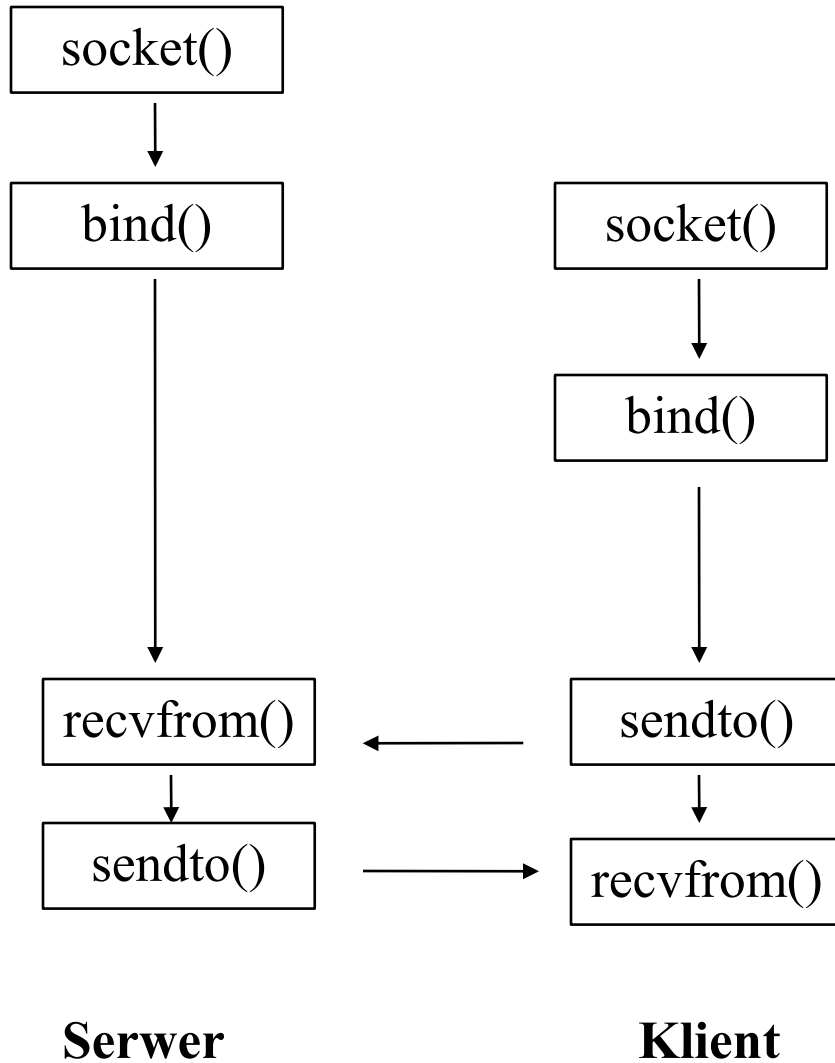


Standardowa wersja w BSD 4.3 - 1986

BSD sockets prócz komunikacji TCP/IP obsługuje inne prot. oraz komunikację lokalną ("Unix Domain Sockets")

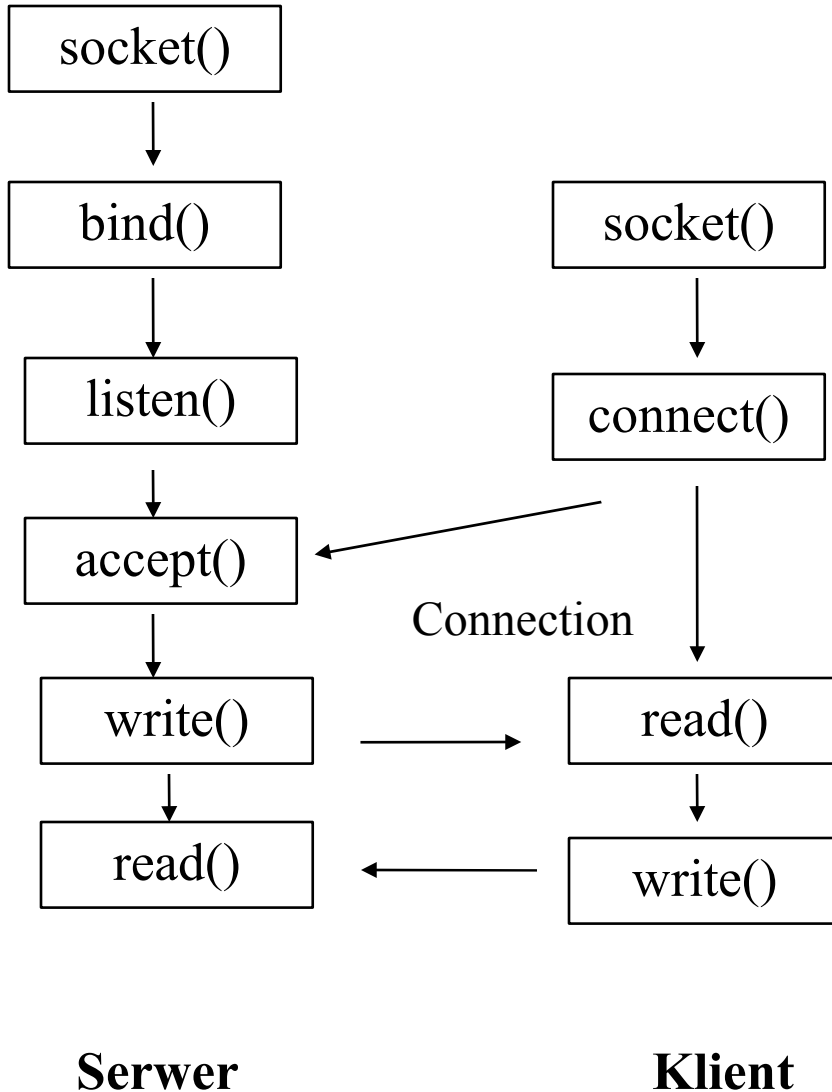


Komunikacja UDP





Komunikacja TCP





API - nagłówki

Unix (zazwyczaj)

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

Windows (winsock)

```
#include <windows.h>
#include <sys/types.h>
#include <winsock.h>
```



Adresowanie

Adres generyczny - **sockaddr** - stosowany we wszystkich wywołaniach systemowych

Adresy poszczególnych rodzin - "inet", "unix", ...

Stosuje się adres specyficzny dla rodziny rzutowany na generyczny

```
connect (sfd, (struct sockaddr*) &serv_addr,  
         sizeof(serv_addr))
```

```
struct sockaddr {  
    u_short sa_family;  
    u_short sa_data[14];  
}
```

```
struct sockaddr_in {  
    short sin_family;  
    u_short sin_port;  
    struct in_addr sin_addr;  
    char sin_zero[8];  
}
```

```
struct in_addr {  
    long int s_addr;  
}
```




Budowanie adresów

Przykład I (adres/port zadany):

```
sock_addr.sin_family = AF_INET;  
sock_addr.sin_addr.s_addr = inet_addr("192.168.1.1");  
sock_addr.sin_port = htons(8000);
```

Przykład II (adres/port zostanie wyznaczony):

```
sock_addr.sin_family = AF_INET;  
sock_addr.sin_addr.s_addr = htonl(INADDR_ANY);  
sock_addr.sin_port = htons(0);
```



socket()

```
int socket ( int family, int type, int protocol );
```

```
#include <sys/types.h>
```

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

family: rodzina protokołów {AF_INET - TCP/IP, AF_UNIX - Unix Dom., ...}

type: rodzaj komunikacji

SOCK_STREAM	stream socket	TCP	(IPPROTO_TCP)
SOCK_DGRAM	datagram socket	UDP	(IPPROTO_UDP)
SOCK_RAW	raw socket	IP, ICMP	(IPPROTO_RAW, IPPROTO_ICMP)

protocol: wyznaczony przez **family** oraz **type**, chyba, że SOCK_RAW

zwraca sukces: deskryptor gniazda (z rodziny deskryptorów plików)

błąd: -1

Np.:

```
If (( sockfd = socket (AF_INET, SOCK_STREAM, 0)) < 0)  
    err_sys ("socket call error");
```



bind()

```
int bind (int sockfd, const struct sockaddr *saddr, socklen_t addrlen);
```

bind przypisuje **lokalny** adres do gniazda

adres: 32 bit IPv4 + 16 bit port TCP lub UDP

sockfd: deskryptor gniazda zwrócony przez funkcję sys. **socket** ()

**saddr*: wskaźnik na strukturę z adresem

addrlen: wielkość adresu j.w.

zwraca sukces: 0

błąd: -1

Zastosowania: kilka różnych (głównie: "rejestracja" serwera)

Przykład:

```
If (bind (sd, (struct sockaddr *) &servaddr, sizeof (servaddr)) != 0)  
    errsyst ("bind call error");
```



connect()

```
int connect (int sockfd, const struct sockaddr *servaddr, socklen_t  
  addrlen);
```

connect zestawia (o ile to możliwe) połączenie TCP ze wskazanym serwerem, funkcja blokująca (do czasu nawiązania połączenia lub wystąpienia błędu)

adres: 32 bit IPv4 + a 16 bit port TCP lub UDP

sockfd: deskryptor gniazda zwrócony przez funkcję **socket()**

***saddr**: wskaźnik na strukturę z adresem

addrlen: wielkość adresu j.w.

zwraca **sukces:** 0

błąd: -1

Przykład:

```
if ( connect (sockfd, (struct sockaddr *) &servaddr, sizeof (servaddr)) != 0)  
  err_sys("connect call error");
```

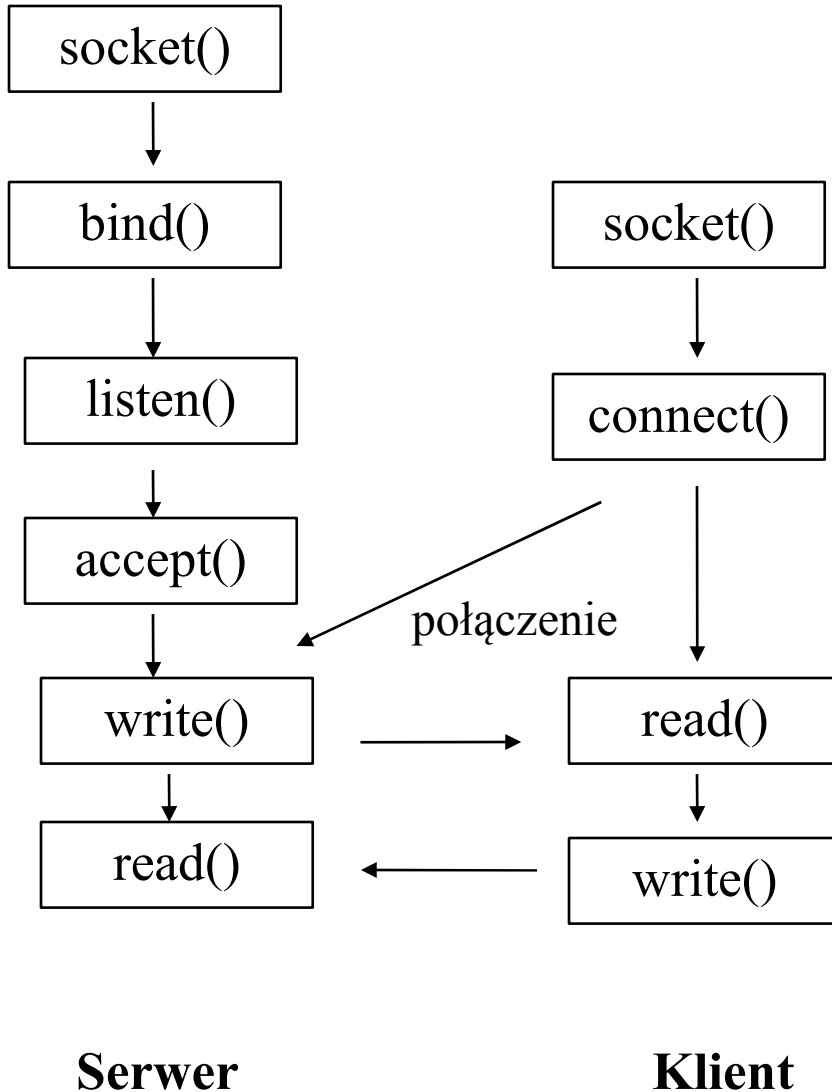


c.d. connect()

- Connect nie wymaga wywołania `bind()` - sam przydziela adres i port (efemeryczny)
- `connect()` ma też sens dla UDP:
 - zapamiętuje adres serwera - przydatne jeśli ma być wysłane wiele datagramów (`send()`, zamiast `sendto()`)
 - pozwala na uzyskanie komunikatów o błędach (asynchronicznie)
- Typowe wartości *errno* w wypadku wystąpienia błędu:
 - **ETIMEDOUT**
 - **ECONNREFUSED**
 - **ENETNOTREACH**
 - **EISCONN**



Komunikacja TCP





listen()

```
int listen (int sockfd, int backlog);
```

- Listen jest wywoływane **tylko przez serwer** TCP:
- Zamienia socket niepołączony w socket pasywny (nasłuchujący)
- **sockfd**: deskryptor gniazda zwrócony przez funkcję sys. **socket()**
- **backlog** - wyznacza maks. liczbę (jeszcze) nieodebranych połączeń, które mogą być skolejkowane w tym gnieździe
- W Linux **backlog** określa długość kolejki zestawionych połączeń, długość kolejki poł. niezestawionych określa parametr kernela:
`/proc/sys/net/ipv4/tcp_max_syn_backlog` (man 7 tcp)

zwraca **sukces:** 0

błąd: -1

Przykład:

```
If (listen (sd, 32) != 0)  
    errs ("listen call error");
```



accept()

```
int accept (int sockfd, struct sockaddr *cliaddr,  
            socklen_t *addrlen);
```

- **accept** - funkcja blokująca do czasu odebrania połączenia
- powoduje przyjęcie wchodzącego połączenia sieciowego, zwracane jest w pełni zasocjowane gniazdo. Oryginalne gniazdo pozostaje w stanie półasocjacji.
- W przypadku gdy wiele połączeń oczekuje na "zaakceptowanie" wybrane zostaje pierwsze z czoła kolejki oczekujących (FIFO).

sockfd: deskryptor gniazda zwrócony przez funkcję **socket**

cliaddr: zwracany jest adres klienta

addrlen: długość adresu *cliaddr*, musi być ustawiony na długość **bufora**
**cliaddr*

zwraca **sukces**: deskryptor socketu

błąd: -1



accept() - serwer iteracyjny

Po powrocie **accept** może ponownie nasłuchiwać na "oryginalnym" gnieździe, a gniazdo zwrócone powinno być wykorzystane do obsługi połączenia:

```
for (; ;) {  
    new_sfd=accept(sfd, ...);          /* wcześniej socket, listen */  
    if (newsfd<0) { ... /* błąd! */  
        exit(1); }  
    Do(new_sfd);          /* obsłuż połączenie */  
    close(new_sfd);      /* zamknij gniazdo zasocjowane */  
}
```



accept() - serwer współbieżny

Po powrocie **accept** może ponownie nasłuchiwać na "oryginalnym" gnieździe, a gniazdo zwrócone powinno być wykorzystane do obsługi połączenia:

```
for (; ;) {
    new_sfd=accept(sfd, ...);          /* wcześniej socket, listen */
    if (newsfd<0) { ... /* błąd! */
        exit(1); }
    if ( fork() == 0) { /* proces potomny */
        close (sfd);          /* zwolnij gniazdo nasłuchujące */
        Do(new_sfd);          /* obsłuż połączenie */
        exit(0);              /* koniec potomka */
    }
    close(new_sfd);           /* rodzic tego nie potrzebuje! */
}
```



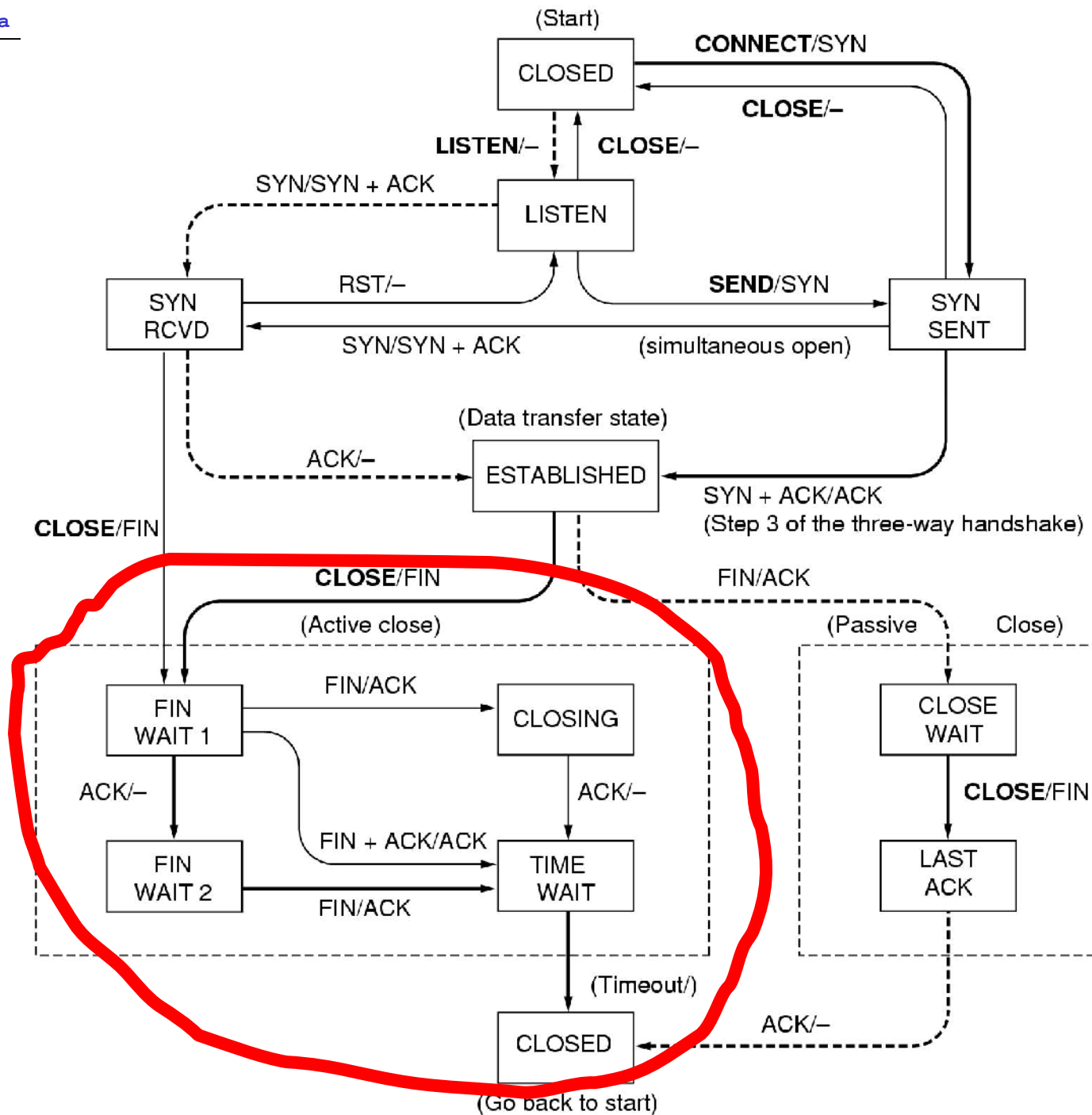
close()

```
int close (int sockfd);
```

- **close** gniazdo zostaje natychmiast zamknięte, funkcja nie blokuje
- *sockfd* - deskryptor zamykanego gniazda.

UWAGA – TCP spróbuje dostarczyć niewysłane dane, a następnie wynegocjować zamknięcie połączenia

zwraca sukces: 0
błąd: -1





Wysyłanie i odbieranie danych



Wysyłanie i odbieranie danych

- Funkcje "standardowe" - `read()`, `write()`
- Funkcje z API gniazd: `send()`, `recv()`, `sendto()`, `recvfrom()`
- **UWAGA** - Funkcje `we/wy` dla gniazd zachowują się inaczej niż ich odpowiedniki dla plików:
 - odbiór/wysłanie mniejszej liczby danych niż zamierzaliśmy jest częsty, najbardziej typowe przypadki:
 - TCP: `read()`, `recv()` - nigdy nie zwróci więcej bajtów niż żądamy (niezależnie od tego ile danych w buforze)
 - TCP: `read()`, `recv()` - zwróci maks. tyle bajtów ile aktualnie zostało zbuforowanych, np. `read(sfd, buf, 1024)` może zwrócić 500 B.
 - TCP: `read()`, `recv()`: jeśli nie ma żadnych danych zbuforowanych: blokada



Wysyłanie i odbieranie danych

- **UWAGA** - Funkcje we/wy dla gniazd zachowują się inaczej niż ich odpowiedniki dla plików:
 - odbiór/wysłanie mniejszej liczby danych niż zamierzaliśmy jest częsty, najbardziej typowe przypadki:
 - TCP: `write()`, `send()` - może wysłać mniej danych niż żądamy – zawsze sprawdzać co zwraca funkcja!
 - TCP: `write()`, `send()` - blokada jeśli odbiorca “nie nadaża”
 - UDP – `write()`, `send()` - wysyła datagram – atomowo!
 - UDP – `write(sfd, buf, 1); write(sfd, buf+1, 1);`
wysyła zawsze dwa datagramy, ale: `write(sfd, buf, 2);`
wysyła jeden (dla TCP może być różnie...)
 - czasy blokowania są do kilku rzędów wielkości dłuższe (do dziesiątków sekund w stosunku do maks. kilku ms dla urządzeń dyskowych)
 - `close()` - ma inną semantykę (patrz wcześniej)



send(), sendto()

```
int send(int sfd, char *buf, int nbytes, int flags)
```

```
int sendto(int sfd, char *buf, int nbytes, int flags,  
           struct sockaddr *to, int addrlen)
```

`sfd` - deskryptor gniazda

`buf` - adres bufora z danymi

`nbytes` - wielkość tego bufora

`flags` - dodatkowe flagi transmisji: **MSG_OOB**, MSG_EOR, MSG_EOF, MSG_DONTROUTE

- zwracają sukces: liczba-wysłanych-bajtów
błąd: -1
- mogą być używane w TCP i UDP (*connect()* wcześniej dla *send()*)
- dla UDP - wysłanie datagramu
- dla TCP: możliwość wysłania OOB, oraz flag EOR, EOF



recv(), recvfrom()

```
int recv(int sfd, char *buf, int nbytes, int flags)
```

```
int recvfrom(int sfd, char *buf, int nbytes, int flags,  
struct sockaddr *from, int *addrlen)
```

- **sfd** - deskryptor gniazda
- **buf** - adres bufora z danymi
- **nbytes** - wielkość tego bufora
- **flags** - dodatkowe flagi transmisji, MSG_PEEK, MSG_WAITALL
- zwracają sukces: **liczba-odebranych-bajtów**
błąd: **-1**
- mogą być używane w TCP i UDP
- są blokujące
- MSG_PEEK -możliwość "zajrzenia" do bufora bez usunięcia danych
- MSG_WAITALL - TCP: czeka na wszystkie dane, ale nie więcej niż:
SO_RCVBUF (zob. też `/proc/sys/net/core/rmem_default`)



Korzystanie z resolver-a (API DNS)

- API - Tłumaczenie nazw DNS na adresy IP (tak aby dało się wypełnić `sockaddr_in`)
- Komplikacja - odwzorowanie nazwa DNS - IP jest typu $n:m$

```
#include <netdb.h>

extern struct h_errno;

struct hostent
    *gethostbyname (char *name);

struct hostent
    *gethostbyaddr (char *addr,
        int len,
        int type);
```

```
struct hostent {
    char *h_name;
    char **h_aliases;
    int h_addr_type;
    int h_length;
    char **h_addr_list; }

h_errno:
    HOST_NOT_FOUND,
    TRY_AGAIN, NO_DATA
```



Resolver - "nowy interfejs"

```
#include <netdb.h>
```

```
int getaddrinfo(const char *node,  
               const char *service,  
               const struct addrinfo *hints,  
               struct addrinfo **res);
```

node - adres IPv4, IPv6 lub nazwa

service - można przekazać # portu do wstawienia do adr. wynikowego

hints - preferowany typ gniazda lub protokołu

Zwraca: 0 - OK, kod błędu inaczej

Zwraca: **res** - lista struktur zawierających adresy: ai_addr

```
struct addrinfo {  
    int    ai_flags;  
    int    ai_family;    // jak w socket  
    int    ai_socktype;  // jak w socket  
    int    ai_protocol;  // jak w socket  
    size_t ai_addrlen;  
    struct sockaddr *ai_addr;  
    char   *ai_canonname;  
    struct addrinfo *ai_next; // lista };
```

użyć: void **freeaddrinfo**(struct addrinfo ***res**); aby zwolnić wyniki



Więcej o adresach

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

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

konwersja adresu numerycznego na string, zwraca 0 jeśli błąd

```
int inet_aton(const char *cp, struct in_addr *inp);
```

konwersja adresu w postaci stringu na numeryczny, zwraca 0 jeśli błąd

```
in_addr_t inet_addr(const char *cp);
```

konwersja adresu w postaci stringu na numeryczny, zwraca -1 jeśli błąd
(niejednoznaczność z 255.255.255.255)

w.w. funkcje nie obsługują IPv6 i nie powinny być używane (ale b. często się je spotyka)



Jeszcze więcej o adresach

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

```
const char *inet_ntop(int af, const void *restrict src,
                      char *restrict dst, socklen_t size);
```

Konwersja adresu numerycznego z dziedziny `af` (`AF_INET`, `AF_INET6`) na napis; `src` – bufor z adresem numerycznym; `dst` – bufor na napis (rozmiar `INET_ADDRSTRLEN` lub `INET6_ADDRSTRLEN`); zwraca `NULL` jeśli błąd

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

Konwersja adresu zapisanego jako string z dziedziny `af` (`AF_INET`, `AF_INET6`) na numeryczny; `src` – bufor z adresem numerycznym; `dst` – bufor na napis; zwraca 0 lub -1 (niepoprawny `af`) jeśli błąd



Przykłady

Uwaga: przykładowy kod dostępny pod adresem:

`http://www.ii.pw.edu.pl/~gjb/tin.html`



Klient/serwer - gniazda TCP

Klient:

```
./client serv-name serv-port
```

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

```
#define DATA "Half a league, half a league . . ."
```

```
int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in server;
    struct hostent *hp;

    /* Create socket. */
    sock = socket( AF_INET, SOCK_STREAM, 0 );
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
}
```



```
/* uzyskajmy adres IP z nazwy ... */
server.sin_family = AF_INET;
hp = gethostbyname(argv[1] ); /* powinnismy sprawdzic argc! */

/* hostbyname zwraca strukture zawierajaca adres danego hosta */
if (hp == (struct hostent *) 0) {
    fprintf(stderr, "%s: unknown host\n", argv[1]);
    exit(2);
}
memcpy((char *) &server.sin_addr, (char *) hp->h_addr,
        hp->h_length);
server.sin_port = htons(atoi( argv[2]));
if (connect(sock, (struct sockaddr *) &server, sizeof server)
    == -1) {
    perror("connecting stream socket");
    exit(1);
}
if (write( sock, DATA, sizeof DATA ) == -1)

    perror("writing on stream socket");
close(sock);
exit(0);}
```




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

```
#define TRUE 1
```

```
void main(void)
{
```

```
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;
```

```
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
```

Serwer:
./serv



```
/* dowiaz adres do gniazda */
/* "prawdziwy" serwer pobrał by port poprzez getservbyname() */

server.sin_family = AF_INET;
server.sin_addr.s_addr = INADDR_ANY;
server.sin_port = 0;
if (bind(sock, (struct sockaddr *) &server, sizeof server)
    == -1)
    perror("binding stream socket");
    exit(1);
}
/* wydrukuj na konsoli przydzielony port */
length = sizeof server;
if (getsockname(sock, (struct sockaddr *) &server, &length)
    == -1) {
    perror("getting socket name");
    exit(2);
}
printf("Socket port %#d\n", ntohs(server.sin_port));
/* zacznij przyjmować połączenia... */
listen(sock, 5);
```



```
do {
    msgsock = accept(sock, (struct sockaddr *) 0, (int *)
    if (msgsock == -1 ) {
        perror("accept"); exit(3); }
    else do {
        memset(buf, 0, sizeof buf);
        if ((rval = read(msgsock,buf, 1024)) == -1) {
            perror("reading stream message");
            exit(4); }
        if (rval == 0)
            printf("Ending connection\n");
        else
            printf("-->%s\n", buf);
    } while (rval > 0);
    close(msgsock);
} while(TRUE);
/*
 * gniazdo sock nie zostanie nigdy zamkniete jawnie,
 * jednak wszystkie deskryptory zostana zamkniete gdy proces
 * zostanie zakonczony (np w wyniku wystapienia sygnalu)
 */

exit(0);
}
```



getservbyname()

```
struct servent* getservbyname(const char *name,  
                                const char *proto);
```

```
struct servent* getservbyport(int port, const char *proto);
```

- **name** – nazwa serwisu z bazy (/etc/services lub NIS/NIS+)
- **port** – numer portu z bazy
- **proto** – nazwa protokołu transportowego “tcp” | “udp” | NULL
- zwracają sukces: **wskaźnik na strukturę z opisem**
błąd: **NULL**

```
struct servent {  
    char    *s_name;           /* nazwa serwisu */  
    char    **s_aliases;      /* lista aliasów nazwy serwisu */  
    int     s_port;           /* numer portu */  
    char    *s_proto;         /* protokół transportowy */  
}
```



getservbyname() c.d.

`/etc/services:`

```
echo          7/tcp
echo          7/udp
daytime       13/tcp
daytime       13/udp
ftp-data      20/tcp      #File Transfer [Default Data]
ftp           21/tcp      #File Transfer [Control]
ssh           22/tcp      #Secure Shell Login
telnet        23/tcp
```

Przykład:

```
int main()
{
    struct servent* se = getservbyname("echo", "tcp");
    printf("Port # is: %d\n", ntohs(se->s_port) );
}
```



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

```
#define DATA "The sea is calm, the tide is full . . ."
```

```
int main(int argc, char *argv[])
{
    int sock;
    struct sockaddr_in name;
    struct hostent *hp;

    /* Create socket on which to send. */
    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
}
```

Klient/serwer - gniazda UDP

Klient:

```
./client serv-name serv-port
```



```
hp = gethostbyname(argv[1]);
if (hp == (struct hostent *) 0) {

    fprintf(stderr, "%s: unknown host\n", argv[1]);
    exit(2);
}
memcpy((char *) &name.sin_addr, (char *) hp->h_addr,
        hp->h_length);
name.sin_family = AF_INET;
name.sin_port = htons( atoi( argv[2] ));
/* Send message. */
if (sendto(sock, DATA, sizeof DATA ,0,
           (struct sockaddr *) &name, sizeof name) == -1)
    perror("sending datagram message");
close(sock);
exit(0);
}
```



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

void main(void)
{
    int sock, length;
    struct sockaddr_in name;
    char buf[1024];

    sock = socket(AF_INET, SOCK_DGRAM, 0);
    if (sock == -1) {
        perror("opening datagram socket");
        exit(1);
    }
    /* Create name with wildcards. */
    name.sin_family = AF_INET;
    name.sin_addr.s_addr = INADDR_ANY;
    name.sin_port = 0;
    if (bind(sock, (struct sockaddr *)&name, sizeof name) == -1) {
        perror("binding datagram socket");
        exit(1);
    }
}
```




```
/* Wydrukuj na konsoli numer portu */
```

```
length = sizeof(name);
```

```
if (getsockname(sock, (struct sockaddr *) &name, &length)
    == -1) {
    perror("getting socket name");
    exit(1);
}
```

```
printf("Socket port #%d\n", ntohs( name.sin_port));
```

```
/* Read from the socket. */
```

```
if ( read(sock, buf, 1024) == -1 ) {
    perror("receiving datagram packet");
    exit(2); }
```

```
printf("-->%s\n", buf);
```

```
close(sock);
```

```
exit(0);
```

```
}
```



Gniazda w domenie Unix (dygresja)

Gniazda "lokalne" (tj. w domenie adresowania "Unix") są dość często wykorzystywane w bardziej złożonych systemach serwerowych ...

... np. do komunikacji pomiędzy różnymi procesami tworzącymi serwer:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
...
struct sockaddr_un addr;

s = socket(AF_UNIX, SOCK_DGRAM, 0);
strcpy(addr.sun_path, "/tmp/foo");
addr.sun_family = AF_UNIX;
bind(s, (struct sockaddr *) &addr,
      strlen(addr.sun_path) + sizeof (addr.sun_family));
```



Gniazda z IPv6

- Adres IPv6 ma 128 bitów, przykład: 3ffe:8041:2:1:bc16:6ee6:26ab:7ba8
- Port używany jest tak samo (zmiany dotyczą warstwy 3)
- Używamy tych samych funkcji, ale innych struktur adresowych
- Używamy tych samych nagłówków .h

```
struct sockaddr_in6
{
    sa_family_t    sin6_family;
    in_port_t      sin6_port;
    uint32_t       sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t       sin6_scope_id;
};

struct in_addr6 {
    u_char        s6_addr[16];           /* IPv6 address */
};
```



Gniazda z IPv6 - podsumowanie

	IPv4	IPv6	
Data structures	AF_INET	AF_INET6	
	in_addr sockaddr_in	in6_addr sockaddr_in6	
Name-to-address functions	inet_aton() inet_addr()	inet_pton() *	IPv4 and IPv6 functions
	inet_ntoa()	inet_ntop() *	
Address conversion functions	gethostbyname() gethostbyaddr()	getipnodebyname() getipnodebyaddr getnameinfo() * getaddrinfo() *	



Gniazda z IPv6 c.d.

```
struct sockaddr_in6 server;
struct hostent *hp;
. . .
hp = gethostbyname2(argv[1], AF_INET6 );
if (hp == NULL) {
    . . .
}
memset( (char *)&server, 0, sizeof(server) );
server.sin6_len = sizeof(server);
memcpy((char *)&server.sin6_addr, hp->h_addr, hp->h_length);
server.sin6_family = AF_INET6 /* hp->h_addrtype; */
server.sin6_port = htons( 8000 );
. . .
s = socket( AF_INET6, SOCK_STREAM, 0);
if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0) {
    . . .
}
```



Multipleksowanie gniazd

- Kiedy obsługa wielu gniazd "na raz" może być wskazana, typowe przykłady:
 - oczekiwanie na więcej niż jednym accept (np. program P2P - czeka na zgłoszenie i wyszukania i na zgł. transferu)
 - obsługa wielu połączeń przez program, który (z jakichś przyczyn) nie jest wieloprocesowy / wielowątkowy
 - obsługa broadcastów i zwykłych połączeń
- Jak zrealizować oczekiwanie na więcej niż jednym gnieździe ? - accept jest funkcją blokującą
 - Użycie opcji nieblokowania - oczekiwanie aktywne (wada - marnowanie zasobów CPU)
 - timeout & przerwania - nieskuteczne czasowo - co się stanie gdy połączenie nadejdzie na początku okresu timeout-u?
 - Przy użyciu do tej pory poznanych mechanizmów nie da się efektywnie zrealizować!



Select()

```
int select (int nfds,  
            fd_set *rfds, fd_set *wfds , fd_set *efds,  
            struct timeval *timeout);
```

nfds - liczba deskryptorów, *zawsze deskryptory od 0 do nfd-1!*

*rfd*s, *wfd*s, *efd*s - zbiory deskryptorów: czytanie, pisanie, sygnał

timeout - zob. dalej

zwraca sukces: ≥ 0 (liczba gniazd spełn. warunek ze zbioru deskrypt.)

błąd: -1

Przykład:

```
fd_set readmask, writemask, exceptmask;
```

```
struct timeval timeout;
```

```
...
```

```
select(nfds, &readmask, &writemask, &exceptmask, &timeout);
```



Select() - fd_set

- Typ: `fd_set`
- makra:
 - `FD_SET(fd, &mask)` - dodaj deskryptor
 - `FD_CLR(fd, &mask)` - usuń deskryptor
 - `FD_ZERO(&mask)` - zeruj zbiór
 - `FD_ISSET(fd, &mask)` - sprawdź deskryptor
 - `FD_SETSIZE` - domyślnie 1024, można zwiększyć do 65536
- `Select()` modyfikuje zbiór deskryptorów po każdym wykonaniu (aby można było sprawdzić, które deskryptory były aktywne!)
- Timeout:
 - `timeout == NULL` - oczekiwanie nieskończone (lub sygnał)
 - `timeout.tv_sec == timeout.tv_usec == 0` - bez blokowania
 - `timeout` - wartości nie zerowe - czekaj zadany czas



```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
```

```
#define TRUE 1
```

```
int main(int argc, char *argv[])
{
    int sock, length;
    struct sockaddr_in server;
    int msgsock;
    char buf[1024];
    int rval;
    fd_set ready;
    struct timeval to;
```

```
/* ... utwórz gniazda i mu przypisz im adresy */
```



```
sock = socket( ... );
/* . . . */

listen(sock, 5);
do {
    FD_ZERO(&ready);
    FD_SET(sock, &ready);
    to.tv_sec = 5;
    to.tv_usec = 0;
    if (select(4, &ready, (fd_set *)0, (fd_set *)0, &to)
        == -1) {
        perror("select");
        continue;
    }
    if (FD_ISSET(sock, &ready)) {
        msgsock = accept(sock, (struct sockaddr *)0,
            (int *)0);
        if (msgsock == -1)
            perror("accept");
        else do {
```



```
    memset(buf, 0, sizeof buf);
    if ((rval = read(msgsock, buf, 1024)) == -1)
        perror("reading stream message");
    else if (rval == 0)
        printf("Ending connection\n");
    else
        printf("-->%s\n", buf);
} while (rval > 0);
close(msgsock);
} else
    printf("Do something else\n");
} while (TRUE);
exit(0);
}
```

Problem:

- Przykładowy program nie wykorzystuje równoczesnej obsługi kilku połączeń – `read()` wywoływany jest w osobnej pętli
- W następnym przykładzie równoległa obsługa `sock` i `msgsock`



Select() -przykład #2

```
#define READ_SIZE 20
int main(int argc, char **argv)
{
    int sock, length;
    struct sockaddr_in server;
    fd_set ready;
    struct timeval to;
    int msgsock=-1, nfds, nactive;
    char buf[1024];
    int rval=0,i;

    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        Exit(1); }
    nfd = sock+1;
/* dowiąż adres do gniazda */
...
/* wydrukuj na konsoli przydzielony port */
...
/* zacznij przyjmować połączenia... */
listen(sock, 5);
```



Select() - przykład #2

```
do {
    FD_ZERO(&ready);
    FD_SET(sock, &ready);
    if ( msgsock > 0 ) /* otwarte gniazdo danych */
        FD_SET(msgsock, &ready);
    to.tv_sec = 5;
    to.tv_usec = 0;
    if ( (nactive=select(nfds, &ready, (fd_set *)0,
        (fd_set *)0, &to) == -1) {
        perror("select");
        continue; }
    if ( FD_ISSET(sock, &ready)) {
        msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
        if (msgsock == -1)
            perror("accept");
        nfds=msgsock+1;
        printf("accepted...\n");
    } else
        if ( msgsock>0 && FD_ISSET(msgsock, &ready)) {
```



Select() - przykład #2

```
if ( msgsock>0 && FD_ISSET(msgsock, &ready)) {
    memset(buf, 0, sizeof buf);
    if ((rval = read(msgsock, buf, READ_SIZE)) == -1)
        perror("reading stream message");
    if (rval == 0) {
        printf("Ending connection\n");
        close( msgsock );
        msgsock=-1;
        nfds=sock+1; }
    else
        printf("-->%s\n", buf);
        sleep( 1 ); }
    if (nactive==0) printf("Timeout, restarting select...\n");
} while(TRUE);
    exit(0);
}
```

Problem:

- `sleep()` i mały bufor `READ_SIZE` wprowadzono w celach testowych (łatwo użyć kilku równoległych klientów)
- W programie jest błąd ...
- Co się stanie gdy `msgsock>0` i przyjdzie nowe połączenie ?



Select() - przykład #3 (poprawiony)

```
int main(int argc, char **argv)
{
    int sock, length;
    struct sockaddr_in server;
    fd_set ready;
    struct timeval to;
    int msgsock=-1, nfd, nactive;
    int socktab[MAX_FDS]; /* musimy mieć gniazdo na każde *
                           obsługiwane połączenie */
    char buf[1024];
    int rval=0, i;

    for (i=0; i<MAX_FDS; i++) socktab[i]=0;
    sock = socket(AF_INET, SOCK_STREAM, 0);
    if (sock == -1) {
        perror("opening stream socket");
        exit(1);
    }
    nfd = sock+1;

    /* dowiąż adres do gniazda */
    . . .
```



Select() - przykład #3

```
/* wydrukuj na konsoli przydzielony port */
. . .
/* zacznij przyjmować połączenia ... */

listen(sock, 5);

do {
    FD_ZERO(&ready);
    FD_SET(sock, &ready);
    for (i=0; i<MAX_FDS; i++) /* dodaj aktywne do zbioru */
        if ( socktab[i]>0 )
            FD_SET( socktab[i], &ready);
    to.tv_sec = 5;
    to.tv_usec = 0;
    if ( (nactive=select(nfds, &ready, (fd_set *)0,
        (fd_set *)0, &to)) == -1) {
        perror("select");
        continue;
    }
}
```




Select() - przykład #3

```
if ( FD_ISSET(sock, &ready) ) {
    msgsock = accept(sock, (struct sockaddr *)0, (int *)0);
    if (msgsock == -1)
        perror("accept");
    nfds=max(nfds, msgsock+1);
    /* brak sprawdzenia czy msgsock>MAX_FDS */
    socktab[msgsock]=msgsock;
    printf("accepted...\n");
}
for (i=0; i<MAX_FDS; i++)
    if ( (msgsock=socktab[i])>0 && /* msgsock zawsze == i */
        FD_ISSET(socktab[i], &ready) ) { /* ale kod ładniejszy ...*/
        memset(buf, 0, sizeof buf);
        if ((rval = read(msgsock, buf, READ_SIZE)) == -1)
            perror("reading stream message");
        if (rval == 0) {
            printf("Ending connection\n");
            close( msgsock );
            socktab[msgsock]=-1; } /* usuń ze zbioru */
        else {
            printf("- %2d ->%s\n", msgsock, buf);
            sleep( 1 ); }
    }
```



Rozgłaszanie (broadcast)

- Rozgłaszanie tylko dla UDP! (dlaczego?)
- Rozgłaszanie wymaga:
 - Interfejsu kanałowego, który je obsługuje (np. Ethernet, łączy punkt-punkt typowe dla WAN **nie**)
 - Kod powinien (choć nie musi) sprawdzać czy interfejs sieciowy obsługuje rozgłaszanie
 - Odpowiedniego skonstruowania adresu IP: w części identyfikującej host-a '11111'
 - Ustawienia opcji rozgłaszania dla gniazda
- Rozgłaszanie odbywa się na poziomie warstwy 2 – jest to ograniczone do danej sieci (na poziomie 2); nie mylić z rozsiewaniem (**multicast**), które obsługiwane jest na poziomie 3 i jest przekazywane poza granice sieci lokalnej



Rozgłaszanie (przykład)

- Wprowadzenie gniazda w tryb rozgłaszania:

```
int val = 1;
...
setsockopt(sockd, SOL_SOCKET, SO_BROADCAST, &val,
           sizeof(val));
```

- Wysyłanie odbywa się jak zwykle - `sendto(...)`, trzeba jednak wykorzystać adres “rozgłaszający”
- Można też wykorzystać adres: `INADDR_BROADCAST`
- Idealne rozwiązanie – enumeracja interfejsów w celu sprawdzenia, które obsługują rozgłaszanie (patrz dalej)



Rozgłaszanie (przykład)

- Struktura `ifreq` opisuje interfejs sieciowy:

```
struct ifreq {
    #define IFNAMSIZ 16
    char ifr_name[IFNAMSIZ]; /* nazwa, np., "en0" */
    union {
        struct sockaddr ifru_addr;
        struct sockaddr ifru_dstaddr;
        char ifru_otype[IFNAMSIZ];
        struct sockaddr ifru_broadaddr; /* adres rozgłaszania */
        short ifru_flags;
        ... } ifr_ifru;
    #define ifr_addr ifr_ifru.ifru_addr
    #define ifr_broadaddr ifr_ifru.ifru_broadaddr
    #define ifr_flags ifr_ifru.ifru_flags
};
```



Rozgłaszanie (przykład)

- Musimy pobrać strukturę opisującą interfejsy sieciowe:

```
if (ioctl(s, SIOCGIFNUM, (char *)&numifs) == -1) {
    numifs = MAXIFS; // ustalmy liczbę interfejsów
}
bufsize = numifs * sizeof(struct ifreq); // przydziel pamięć
reqbuf = (struct ifreq *)malloc(bufsize);
if (reqbuf == NULL) { ... } // brak pamięci ?!
}
ifc.ifc_buf = (caddr_t)&reqbuf[0]; // ifc będzie zawierać inf.
ifc.ifc_len = bufsize;           // konf. o interfejsach
if (ioctl(s, SIOCGIFCONF, (char *)&ifc) == -1) { // pobierzmy inf.
    perror("ioctl(SIOCGIFCONF)");           // konfiguracyjną
    exit(1);
}
```



Rozgłaszanie (przykład)

- Musimy pobrać kolejne struktury interfejsu i znaleźć/sprawdzić interfejs, który obsługuje rozgłaszanie:

```
struct ifreq *ifr;
ifr = ifc.ifc_req;
for (n = ifc.ifc_len/sizeof (struct ifreq); --n >= 0; ifr++) {
    if (ifr->ifr_addr.sa_family != AF_INET)
        continue;
    if (ioctl(s, SIOCGIFFLAGS, (char *) ifr) < 0) {
        ...
    }
    if ((ifr->ifr_flags & IFF_UP) == 0 ||
        (ifr->ifr_flags & IFF_LOOPBACK) ||
        (ifr->ifr_flags & (IFF_BROADCAST | IFF_POINTOPOINT)) == 0)
        continue;
}
```



Rozgłaszanie (przykład)

- Tak pobierzemy adres rozgłaszania dla właściwego interfejsu:

```
if (ioctl(s, SIOCGIFBRDADDR, (char *) ifr) < 0) {  
    ...  
}  
memcpy((char *) &dst, (char *) &ifr->ifr_broadaddr,  
sizeof ifr->ifr_broadaddr);
```



Wskazówki dot. kodu projektowego (gniazda)

- Pamiętajmy, że `read()` może zwrócić niepełne dane, nie przetwarzamy danych (np. nagłówków) jeśli nie mamy ich kompletu.
- Kod przetwarzający dane protokołu powinien być "wielopoziomowy" – niższa warstwa zapewnia odczyt (skompletowanie) odpowiedniej porcji danych; wyższa warstwa przetwarza nagłówki; odrębny "moduł" (funkcja, wątek, proces) reaguje i ewentualnie generuje odpowiedź
- Nie stosujemy aktywnego oczekiwania na operacjach `read()`, `write()`
- Nie zawieszamy się (`read()`, `recv()`) w oczekiwaniu na dane jeśli na innym gnieździe też mogą pojawić się dane - stosujemy `select()`
- Pilnujemy zamykania nieużywanych deskryptorów gniazd – w procesach potomnych, po zakończeniu obsługi.
- Nie zamknięcie niepotrzebnego gniazda (np. nasłuchującego w `accept` w procesie potomnym) powoduje, że dla tego deskryptora nigdy nie nastąpi sytuacja końca pliku.
- W modelu wielowątkowym (wieloprocessowym) panujemy na powołanymi wątkami (procesami) – wiemy kiedy się kończą, i co trzeba zrobić gdy to nastąpi:
 - W przypadku procesu: `wait()` w proc. macierzystym lub `signal(SIGCLD, SIG_IGN)`;
 - W modelu wielowątkowym: możemy stosować `pthread_join(...)`



Wskazówki dot. kodu projektowego (gniazda)

- Odzyskujemy/zwalniamy wykorzystane zasoby (calloc()/free(), etc.)
- W modelu wielowątkowym pamiętamy, że wszystkie zmienne poza automatycznymi są globalne, dotyczy to:
 - zmiennych globalnych procesu,
 - dynamicznie przydzielonych struktur danych (nie zostaną usunięte po zakończeniu wątku, nawet jeśli wskaźnik automatyczny),
 - zmiennych niejawnie wykorzystywanych przez niektóre funkcje (np. strstr() - zob. MT-safe w man),
 - stosujemy sekcje krytyczne przy dostępie do zasobów współdzielonych!
- Wątki: na stronie <http://www.ii.pw.edu.pl/~gjb/tin.html> znajduje się link do materiałów dot. wątków