# Sieci TCP/IP - cz. 2

## Interfejs gniazd
## (BSD UNIX *sockets*)

Ostatnia modyfikacja: 08.04.2020

# Standardy

- Interfejs gniazd został opracowany w Univ. of California, Berkeley Univ. i włączony do 4.2BSD UNIX w 1983 r.
- Rodzina standardów IEEE: Portable Operating System Interface (POSIX),
  - Pierwszy: IEEE Std. 1003.1-1988, uakt. w 1990 (również jako ISO/IEC 9945-1:1990)
  - Ostatni: IEEE Std. 1003.1-2008 (z uzupełniem w r. 2013)
    - Część Std. 1003.1B w rozdz. 2.10 zawiera opis interfejsu sockets, a w rozdz. 3 – opisy funkcji interfejsu gniazd
- Open Group Technical Standard: Base Specifications, Issue 6, December 2001.
- Wspólny standard IEEE 1003.1™ POSIX ® oraz The Open Group Base Specifications Issue 6 (2008).
  - Jest zgoda IEEE i The Open Group na włączenie materiału standardu do projektów: Linux Man Pages oraz FreeBSD
- Windows Sockets API (WinSock) – interfejs gniazd firmy Microsoft, częściowo zgodny z interfejsami Unix i POSIX (patrz: strona domowa, dokumentacja techniczna)

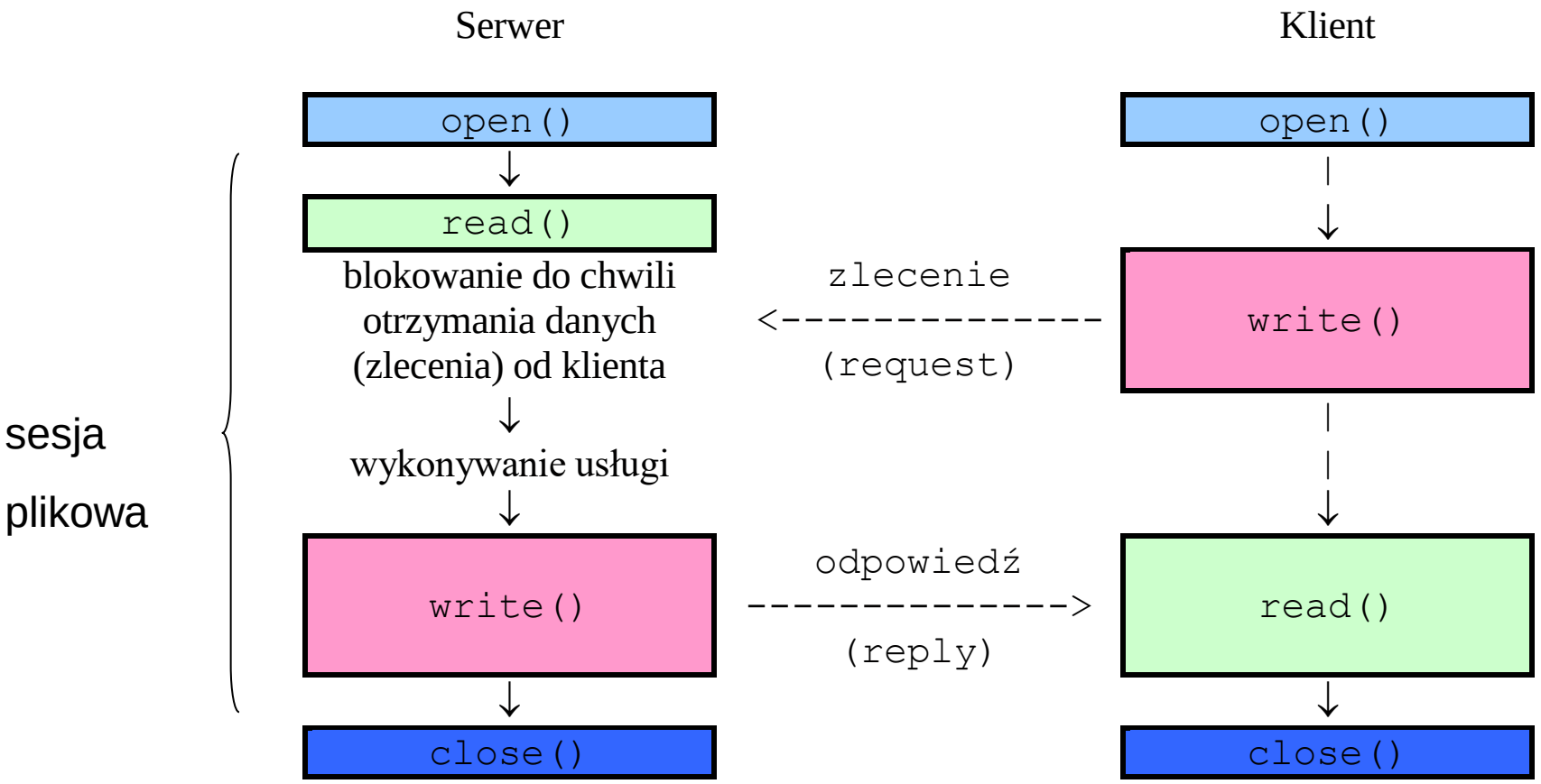
# Specyfika gniazd typu BSD UNIX

- Gniazda reprezentują punkt końcowy komunikacji. Każde gniazdo ma typ i określony protokół. Gniazdo jest dostępny przez deskryptor przydzielany przez system przy jego tworzeniu.
- Gniazda są zintegrowane z Unixowym wejściem-wyjściem
  - Deskryptory gniazd należą do tej samej przestrzeni co deskryptory plików, czy łącz
  - Możliwe jest wykorzystanie tych samych funkcji read/write do komunikacji (po utworzeniu gniazda i wskazaniu nadawcy/odbiorcy)
  - Biblioteka gniazd jest wbudowana w jądro systemu operacyjnego (dostęp przez mechanizm funkcji systemowych)
  - Wspólna obsługa błędów funkcji interfejsu (zmienna **errno** i komunikaty błędów)
- Gniazda w domenie lokalnej (UNIX) są widoczne w systemie plików systemu komputerowego jako pliki specjalne typu **socket**; są one używane do komunikacji międzyprocesowej w obrębie tego systemu.

**Przykład**: /tmp/.X11-unix/X0 w systemie Linux z działającym serwerem X Window System.

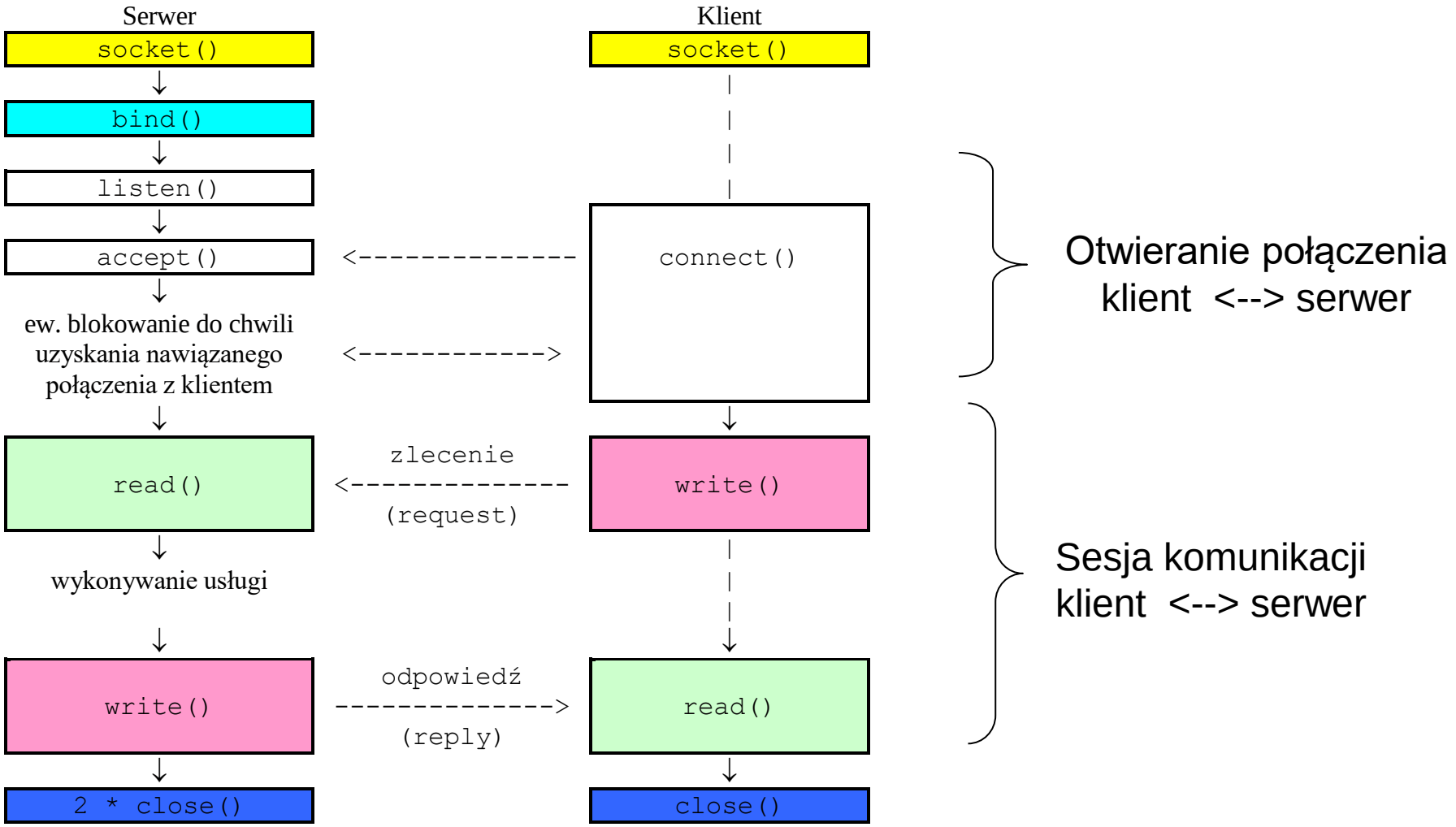
# Prosta komunikacja (zlecenie-odpowiedź)

## Komunikacja za pomocą łączy (nazwanych) lub plików



# Prosta komunikacja (zlecenie-odpowiedź) – interfejs gniazd

## Prosta komunikacja połączeniowa (TCP) typu zlecenie-odpowiedź



# Tworzenie gniazd

```
#include <sys/socket.h>
int socket(
  int family, /* rodzina protokołów
                 POSIX: dostępne protokoły rodziny zależą od implementacji,
                 typowo: PF_INET, PF_INET6, PF_LOCAL (dawniej PF_UNIX),
                         PF_IPX, PF_PACKET */
  int type, /* typ komunikacji
         POSIX: SOCK_STREAM, SOCK_DGRAM,  SOCK_SEQPACKET i określone
                 przez implementację; spotykane: SOCK_RAW, SOCK_PACKET */
  int protocol /* typowo = 0 (domyślny protokół dla wybranej rodziny */
);
```

- Funkcja **socket** tworzy struktury systemowe gniazda, zwracając nieujemny deskryptor gniazda, albo -1 ; wówczas zmienna globalna **errno** (patrz **man errno**) zawiera kod błędu (patrz **man socket**).
- Stan początkowy gniazda: CLOSED.
- Tworzenie gniazd dla niektórych protokołów może wymagać od procesu stosownych uprawnień
- W praktyce każda rodzina protokołów jest związana z jedną rodziną adresowania, a symboliczne oznaczenia rodzin spełniają warunek: **AF_XXX == PF_XXX**

# Opcje gniazd

- Opcje gniazd umożliwiają zmianę zachowania gniazd. Opcje można ustawiać dla różnych warstw stosu protokołów. Najwyższa warstwa, to warstwa gniazd (`SOL_SOCKET`).
- Funkcje pobierania (`getsockopt`) i ustanawiania (`setsockopt`) opcji **optval** (o długości **optlen**) dla gniazda **sockfd**:

```
#include <sys/socket.h>
 int getsockopt(int sockfd, int level, int optname,
     void *optval, socklen_t *optlen);

 int setsockopt(int sockfd, int level, int optname,
     const void *optval, socklen_t optlen);
```

**level** : określa warstwę API, która ma interpretować opcję (np. SOL_SOCKET)

Uwaga: funkcje `fcntl`() i `ioctl`() również umożliwiają zmianę zachowania gniazd (np. obsługa bez blokowania).

# Opcje gniazd (SOL_SOCKET)

| Option | Parameter Type | Parameter Meaning |
|---|---|---|
| SO_ACCEPTCONN | int | Non-zero indicates that socket listening is enabled (*getsockopt*( ) only). |
| SO_BROADCAST | int | Non-zero requests permission to transmit broadcast datagrams (SOCK_DGRAM sockets only). |
| SO_DEBUG | int | Non-zero requests debugging in underlying protocol modules. |
| SO_DONTROUTE | int | Non-zero requests bypass of normal routing; route based on destination address only. |
| SO_ERROR | int | Requests and clears pending error information on the socket (*getsockopt*( ) only). |
| SO_KEEPALIVE | int | Non-zero requests periodic transmission of keepalive messages (protocol-specific). |
| SO_LINGER | struct linger | Specify actions to be taken for queued, unsent data on *close*( ): linger on/off and linger time in seconds. |
| SO_OOBINLINE | int | Non-zero requests that out-of-band data be placed into normal data input queue as received. |
| SO_RCVBUF | int | Size of receive buffer (in bytes). |
| SO_RCVLOWAT | int | Minimum amount of data to return to application for input operations (in bytes). |
| SO_RCVTIMEO | struct timeval | Timeout value for a socket receive operation. |
| SO_REUSEADDR | int | Non-zero requests reuse of local addresses in *bind*( ) (protocol-specific). |
| SO_SNDBUF | int | Size of send buffer (in bytes). |
| SO_SNDLOWAT | int | Minimum amount of data to send for output operations (in bytes). |
| SO_SNDTIMEO | struct timeval | Timeout value for a socket send operation. |
| SO_TYPE | int | Identify socket type (*getsockopt*( ) only). |

# Opcje gniazd (IPPROTO_TCP)

| Option | Parameter type | Parameter meaning |
|---|---|---|
| TCP_KEEPALIVE | int | # of secs between keep alive messages |
| TCP_MAXRT | int | Max. retransmission time |
| TCP_MAXSEG | int | Max. size of TCP segments |
| TCP_NODELAY | int | avoid coalescing of small segments (turn-off Nagle algorithm) |

Linux:

| | | |
|---|---|---|
| TCP_QUICKACK | int | Enable/disable quick ACK mode. When enabled acknowledgments are sent immediately, and not delayed (as usual). Since Linux 2.4.4 |

# Struktury adresowe gniazd (POSIX)

- Ogólna struktura adresowa (sys/socket.h)

```
struct sockaddr /* Ogólna struktura adresowa; musi zawierać pola: */
  sa_family_t   sa_family; /* rodzina adresowa: AF_xxx */
  char          sa_data[]; /* adres właściwy dla protokołu (zm. dł.)*/
```

- Struktura adresowa (**sockaddr_in**) dla protokołu IPv4 (netinet/in.h)

```
typedef uint32_t        in_addr_t;
typedef uint16_t        in_port_t;
struct in_addr { in_addr_t s_addr; }; /* 32-bitowy adres IPv4 */
struct sockaddr_in {/* Struktura adresowa dla IPv4; musi zawierać
                       takie pola: */
  sa_family_t     sin_family;   /* ==AF_INET */
  in_port_t       sin_port;     /* 16b. numer portu (uint16_t) */
  struct in_addr  sin_addr;     /* 32b. adres IPv4 (uint32_t) */
};
```

/* **Uwaga: sin_port** i **sin_addr** są w **porządku sieciowym** */

# Struktury adresowe gniazd - c.d.

- Struktura adresowa (**sockaddr_in6**) dla protokołu IPv6 (netinet/in.h)

```
struct in6_addr  {uint8_t s6_addr[16];}; /* 128-bitowy adres IPv6 */

struct sockaddr_in6 {/* Struktura ogólna dla IPv6; musi zawierać pola: */
  sa_family_t     sin6_family;    /* AF_INET6 */
  in_port_t       sin6_port[14];  /* 16b. numer portu (uint16_t) */
  uint32_t        sin6_flowinfo;  /* et. przepływu i priorytet */
  struct in6_addr sin6_addr;      /* adres IPv6 */
};
/* Uwaga:sin6_port, sin6_flowinfo i sin6_addr są w porządku sieciowym */
```

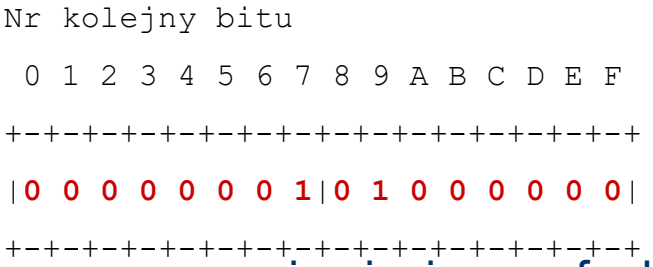
- Struktura dla rodziny adresowej UNIX (AF_UNIX) (sys/un.h)

```
typedef uint32_t in_addr_t;

struct in_addr   in_addr_t s_addr; /* 32-bitowy adres IPv4 */
struct sockaddr_un {/* Struktura adresowa dla domeny UNIX; musi
                       zawierać pola: */
  sa_family_t    sun_family; /* AF_UNIX/AF_LOCAL */
  char           sun_path[]; /* nazwa ścieżkowa, POSIX: długość nieokreślona,
                                typowa długość: 92 do 108 */
};
Uwaga: ścieżka powinna być bezwzględna!!
```

# Funkcje do zamiany kolejności bajtów

- **Porządek sieciowy** bajtów w liczbach wielobajtowych: pierwszy bajt jest najbardziej znacząc (*big-endian*). Np. liczba dwubajtowa **320** jest reprezentowana następująco --->




- Do zmiany porządku bajtów służą w oprogramowaniu sieciowym funkcje **htons**, **htonl**, **ntohs**, **ntohl**

| Typ danej w porządku stacji (host ordered) | Funkcja konwersji | Dana w porządku sieciowym |
|---|---|---|
| **uint16_t** | -------> **htons** ------> | **uint_16_t** |
| **uint16_t** | <------- **ntohs** <------ | **uint_16_t** |
| **uint32_t** | -------> **htonl** ------> | **uint32_t** |
| **uint32_t** | <------- **ntohl** <------ | **uint32_t** |

# Konwersja adresów IP z/do postaci kropkowej

- Funkcje konwersji adresów IPV4 bez obsługi błędów (użycie niezalecane, chociaż w POSIX)

```
#include <arpa/inet.h>
in_addr_t inet_addr(const char *p);/* konwersja adresu kropkowego z p na
                adres IP w postaci binarnej w sieciowej kolejności
                bajtów; przy błędzie zwraca (in_addr_t)(-1) */

char *inet_ntoa(struct in_addr inaddr); /* przekształca adres z inaddr (w
  sieciowej kolejności bajtów) w napis w notacji kropkowo dziesiętnej;
  funkcja zwraca adres napisu przechowywanego w buforze statycznym
  funkcji. */
```

- Funkcja konwersji z obsługą błędów (nie POSIX)

```
#include <arpa/inet.h>

int_t inet_aton(const char *p, struct in_addr *addrptr);/* konwersja
  adresu kropkowego z *p na adres IP w postaci binarnej w sieciowej
  kolejności bajtów - zapamiętany pod adresem addrptr; funkcja zwraca 1
  dla poprawnej konwersji i 0 dla błędu (errno nie  jest zmieniane) */
```

# Konwersja adresów - POSIX

- Funkcje POSIX do konwersji adresów IPv4 i IPv6, z obsługą błędów

```
#include <arpa/inet.h>
const char *inet_ntop(int af,  /* rodzina adresowania */
        const void *restrict src,  /* wskaźnik na bufor z adresem (IPv4 jeśli af==AF_INET */
        char *restrict dst,  /* wskaźnik na bufor w którym funkcja zapisze
                postać tekstową adresu */
        socklen_t size /* rozmiar bufora wskazywanego przez dst */
);
```

/* Funkcja przeprowadza konwersję adresu IPv4/v6 z postaci binarnej (porządek sieciowy) na postać prezentacyjną (ASCII); w przypadku sukcesu zwraca wskaźnik bufora, przy błędzie zwraca NULL (kod błędu w **errno**) . Patrz **man inet_ntop**.*/

```
#include <arpa/inet.h>
int inet_pton(int af, /* rodzina adresowania */
    const char *restrict src, /* wskaźnik na adres w postaci prezentacyjnej (tekstowej) */
    void *restrict dst      /* wskaźnik na bufor w którym funkcja zapisze postać binarną adresu
                            (w porządku sieciowym),  Długość bufora:
                                    32b dla af==AF_INET, 128b dla af==AF_INET6 */
);
```

/* Dla adresu IPv4 funkcja przeprowadza konwersję adresu z postaci kropkowej (**ddd.ddd.ddd.ddd**) na binarną; w przypadku sukcesu zwraca **1**, dla błędnej postaci adresu: **0;** dla nieznanego **af** zwraca -1 ustawiając **errno** na **EAFNOSUPPORT** Funkcja przyjmuje też adresy IPv6 o postaci x:x:x:x:x:x:x:x (patrz **man inet_pton**).

# Korzystanie z nazw stacji

- Struktura adresów stacji **(<netdb.h>)**

```
struct hostent {
char *   h_name;/* oficjalna (kanoniczna) nazwa stacji */
char **  h_aliases; /* tablica pseudonimów, zakończona NULL */
int      h_addrtupe; /* typ adresu (AF_INET lub AF_INET6) */
int      h_length; /* długość adresu (4 lub 16) */
char **  h_addr_list; /* wskaźnik do tablicy wskaźników
                         z adresami IPv4 i IPv6, zakończona NULL */
};
#define h_addr   h_addr_list[0] /* pierwszy adres w tablicy adresów */
```

- Funkcje do obchodu bazy danych adresów: `gethostent(),endhostent()`
- Funkcje do transformacji adresów stacji (wycofane ze standardu POSIX.1-2008)

```
struct hostent* gethostbyname(const char *nazwa/* nazwa symboliczna stacji */
);/* Funkcja zwraca wskaźnik na strukturę adresów stacji o danej nazwie, albo
  NULL (ustawiając zmienną h_errno, zdefiniowaną w <netdb.h>) */
struct host * gethostbyaddr(const char *adres, /* wskaźnik na adres IP w postaci
  binarnej(struct in_addr, albo struct in6_addr) */
  int len, /* długość adresu */
  int typ /* typ adresu (np. AF_INET) */
);/* wartoßci zwracane jak dla gethostbyname() */
```

# Przykład wykorzystania gethostbyname()

```
int main(int argc, char *argv[]){
  char *host;
  int   sockfd, port=….;
  struct sockaddr_in   serv_addr;
  if ( argc>=2){
        host=argv[1];
        if ( argc==3 ) port=atoi(argv[2]);
  } else   host=”127.0.0.1”;
  fprintf(stderr,"%s: server %s, TCP port %d\n", argv[0],host,port);
  memset(&serv_addr,0, sizeof(serv_addr));
  serv_addr.sin_family  = AF_INET;
  if (!isdigit(host[0]) || inet_aton(host,&serv_addr.sin_addr)==0){
  struct hostent *hp=gethostbyname(host);
        if(hp==NULL){
           fprintf(stderr,"%s: host %s not found\n",argv[0],host);
               exit(1);
        }
        serv_addr.sin_addr.s_addr=
               ((struct in_addr *)(hp->h_addr))->s_addr;
  }
  serv_addr.sin_port= htons(port);
..................
}
```

# POSIX : tłumaczenie adresów

**Network address and service translation**

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
/* Given node and service, which identify an Internet host and a  service,
   getaddrinfo(2)  returns  one  or more addrinfo structures, each of which
   contains an Internet address that can be specified in a call to bind(2)
   or  connect(2). */
int getaddrinfo (const char *node, const char *service,/* one can be NULL */
                 const struct addrinfo *hints,/* selection criteria */
                       struct addrinfo **res);/* ptr to a linked list of
                                                 addrinfo structures */
void freeaddrinfo(struct addrinfo *res);/* memory dealloc. of struct addrinfo*/
const char *gai_strerror(int errcode);/*  get*info() error code -> string */
```

**Address to name translation in protocol independent manner**

```
int getnameinfo (
  const struct sockaddr *sa, socklen_t salen,/* socket addr */
  char *host, socklen_t hostlen, /* caller-alloc. buffer to receive host name */
  char *serv, socklen_t servlen, /* caller-alloc. buffer to receive service name
        (names are null terminated) */
  int flags /* modifies function behavior, see getnameinfo(3) */
); /* performs protocol-independent conversion: socket address to a corresponding
  host and/or service. Error codes translated to strings by gai_strerror() */
```

# Przykład użycia getaddrinfo (Tut.7)

```
struct sockaddr_in make_address(char *address, char *port){
        int ret;
        struct sockaddr_in addr;
        struct addrinfo *result;
        struct addrinfo hints = {};
        hints.ai_family = AF_INET;
        if((ret=getaddrinfo(address,port, &hints, &result))){
                fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(ret));
                exit(EXIT_FAILURE);
        }
        addr = *(struct sockaddr_in *)(result->ai_addr);
        freeaddrinfo(result);
        return addr;
}
```

```
Note (see getaddrinfo(3)):
struct addrinfo {
  int              ai_flags;         /* see getaddrinfo(3) */
  int              ai_family;        /* AF_INET, AF_INET6, AF_UNSPEC */
  int              ai_socktype;      /* SOCK_STREAM, SOCK_DGRAM */
  int              ai_protocol;      /* 0 - def. */
  socklen_t        ai_addrlen;        /* socket address in bytes */
  struct sockaddr *ai_addr;          /* socket address */
  char            *ai_canonname;     /* official host name */
  struct addrinfo *ai_next;          /* used for making linked list */
}
```

# Adresowanie gniazda

```
#include <sys/socket.h>
int bind{
  int    sockfd; /* deskryptor gniazda */
  const struct sockaddr *addr; /* wskaźnik na strukturę adresową */
  socklen_t addrlen; /* długość struktury adresowej (w bajtach) */
}
```

- Funkcja **bind** przypisuje gniazdu *adres protokołowy* zwracając normalnie wartość 0; w przypadku błędu zwraca -1 po ustawieniu *errno* (EADDRINUSE, EINVAL,... patrz **man bind**). W domenie UNIX nazwa ścieżkowa gniazda powinna być bezwzględna. Domyślne prawa własności do pliku gniazda (777) zmienia aktualna wartość umask procesu.
- Jeśli **port=0**, to system wybiera automatycznie **port efemeryczny**.
- Użycie adresu uogólnionego (numer IPv4 ma wartość INADDR_ANY, albo numer IPv6 ma wartość IN6ADDR_ANY), powoduje, że dla stacji wielosieciowej (*multihomed host*) oprogramowanie zwleka z wyborem lokalnego adresu aż:
  - gniazdo będzie połączone (TCP), albo
  - będzie wysłany datagram (UDP)
- **Modele systemu końcowego**: słaby i silny (różnica dla stacji wielosieciowych). Model silny akceptuje tylko datagramy przybywające do tego interfejsu, który jest związany z tym samym adresem co podany w przysłanym datagramie (RFC1122).

# Otwieranie połączenia TCP - klient

**Strona klienta** (aktywna):

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


int connect(int sockfd, /* numer gniazda strony klienta */
  const struct sockaddr *servaddr, /* struktura adresowa serwera */
  socklen_t servaddrlen/* długość struktury adresowa serwera */
);
```

`/*` funkcja inicjuje uzgadnianie 3-fazowe połączenia; zwraca normalnie 0, a -1 w przypadku błędu (po ustawieniu `errno`) `*/`

Uwagi:

- Jeżeli gniazdo sockfd (w domenie Internet) nie jest wcześniej zaadresowane, to connect() automatycznie adresuje go, używając portu efemerycznego. W domenie Unix tak nie jest.
- W domenie Unix przepełnienie kolejki oczekujących serwera wywołuje powrót z -1, `errno=ECONNREFUSED`
- Jeżeli `connect`() powróci w rezultacie asynchronicznej obsługi sygnału, nawiązywanie połączenia sieciowego jest kontynuowane (w tle), ale nie można ponownie wywołać `connect`(), aby kontynuować oczekiwanie na połączenie i dowiedzieć się o tym czy połączenie zostało zrealizowane czy też nie. Tutorial 7 przedstawia jedno z rozwiązań tego problemu.

# Otwieranie połączenia TCP - serwer

**Strona serwera** (pasywna):

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


int listen(int sockfd, /* numer gniazda, które ma rozpocząć nasłuch */
  int backlog/* określa długość kolejek połączeń (nawiązywanych
                (SYN_RCVD) i nawiązanych (ESTABLISHED)) */
);/* funkcja ustanawia długość kolejek połączeń; zwraca normalnie 0, a
  -1 w przypadku błędu (po ustawieniu errno) */

int accept(int sockfd, /* numer gniazda nasłuchującego */
  struct sockaddr *restrict cliaddr,/* wsk. struktury adresowej
                                       połączonego klienta */
  socklen_t *restrict addren /* długość struktury wskazywanej
                        przez cliaddr */
);/* funkcja zamienia najstarsze połączenie uzgodnione na nowe połączenie nawiązane (ew.
  czekając), zwraca nr nowego gniazda (połączonego) lub –1 (po ustawieniu errno)
```

# Otwieranie połączenia TCP - uwagi

- Gniazdo nasłuchujące i gniazdo połączone (wymiany danych, utworzone przez **accept**()) są związane z **tym samym portem TCP**
- Nowe gniazdo dziedziczy opcje: `SO_DEBUG, SO_DONTROUTE, SO_SNDBUF, SO_KEEPALIVE, SO_LINGER, SO_OOBINLINE, SO_RCVBUF`, ale żadnych własności, które można ustawić za pomocą **`fcntl`**`()` z poleceniem `F_SETFL` (np. stan *nieblokujący*)
- Odebranie (i obsługa) sygnału w czasie wywołania funkcji długich (normalnie blokujących), w tym **`accept`**`()` i **`connect`**`()` powoduje normalnie powrót tych funkcji z wartością –1 po ustawieniu w **`errno`** wartości **`EINTR`**.
- Domyślne zachowanie funkcji **`connect`**`()` i **`accept`**`()` (blokowanie do momentu, gdy funkcja może zostać zrealizowana, bądź wykryty zostanie błąd wykonania) można zmienić - wykonując zmianę **trybu obsługi** odpowiedniego gniazda (na nieblokujący).

```
#include <fcntl.h>

int sockfd, val;

val=fcntl(sockfd, F_GETL, 0);

if(fcntl(sockfd, F_SETFL, val|O_NOBLOCK)) {/* error */};
```

Ostrzeżenie: do czasu powrotu do zachowania domyślnego – również inne funkcje długie (np. **`read`**`()` będą wykonywane jako nieblokujące.

# Gniazda TCP – przesyłanie danych

- Podstawowe i ”ulepszone” funkcje wejścia/wyjścia:

```
#include <sys/socket.h>
ssize_t read(int sockfd, void *buff, size_t nbytes);
ssize_t write(int sockfd, const void *buff, size_t nbytes);
ssize_t recv(int sockfd, void *buff, size_t nbytes, int flags);
ssize_t send(int sockfd, const void *buff, size_t nbytes, int flags);
```

`flags==MSG_PEEK` — podgląd nadchodzącego komunikatu bez usuwania z bufora,
`flags==MSG_OOB` — wysłanie/odbiór danych pozapasmowych.
`flags==MSG_DONTWAIT` — realizacja tego wywołania funkcji bez blokowania
`flags==MSG_WAITALL` — dla gniazd strumieniowych recv() blockuje aż pobierze wszystkie zamówione dane

- Funkcje odczytu/zapisu wielu buforów

```
struct iovec{    /* bufor rozproszonego odczytu/zapisu */
  void *iov_base; /* adres początkowy bufora */
  size_t iov_len; /* rozmiar bufora */
};
ssize_t readv(int sockfd, const struct iovec *iov, int iovcnt);
ssize_t writev(int sockfd, const struct iovec *iov, int iovcnt);
```

# Przesyłanie danych - uwagi

- Najbardziej ogólną postać maja¸ funkcje wejścia/wyjścia **`recvmsg`** i **`sendmsg`** (patrz **`man recv`**). W domenie UNIX umożliwiają przesyłanie aktywnych deskryptorów plików pomiędzy procesami.
- Próba zapisu do gniazda strumieniowego, którego przeciwny koniec został rozłączony, kończy się błędem **`EPIPE`**, a do procesu wysyłany jest sygnał **`SIGPIPE`** (aby tak się stało strona wysyłająca musi otrzymać powiadomienie o rozłączeniu albo zostanie osiągnięty maksymalny czas oczekiwania na potwierdzenie odbioru wysłanej wiadomości).
- Jeśli funkcja odczytu lub zapisu została przed jej zakończeniem przerwana obsługą sygnału - ustawiany jest kod błędu **`EINTR`**
- W trybie z blokowaniem próba odczytu z gniazda, którego drugi koniec został zamknięty do zapisu, zwraca nieodebrane jeszcze bajty; następna próba odczytu zwraca 0 bajtów.
- Funkcja wysyłająca dane typowo blokuje do czasu wysłania wszystkich bajtów. Jeśli `n=write()` oraz `n>0` , to wiemy jedynie, że `n` bajtów zostało skopiowanych do bufora wyjściowego (wysyłkowego), a nie znaczy to że te bajty zostały już doręczone do stacji odbiorczej. Dane są usuwane z bufora wysyłkowego wówczas, gdy odbiorca przyśle potwierdzenie ich odbioru (*acknowledgement*)

# Zamykanie połączenia TCP

```
#include <sys/socket.h>
int shutdown{
  int sockfd;     /* deskryptor zamykanego gniazda */
  int how;        /* SHUT_RD (0) - czytanie */
                  /* SHUT_WR (1) - pisanie */
                  /* SHUT_RDWR (2) - obydwa kierunki */
}/* zwraca 0, lub -1 dla błędu);

#include <sys/socket.h>
int close{ int sockfd     /* deskryptor zamykanego gniazda */
};/* Funkcja zamyka połączenie w obydwóch kierunkach;
  normalnie zwraca 0, lub -1 dla błędu */
```

**close()** wstrzymuje zarówno wysyłkę jak i odbiór danych za pomocą deskryptora gniazda Dalsze losy połączenia zależą od liczby dowiązań do gniazda oraz od opcji gniazda **SO_LINGER**.

# Zamykanie połączenia TCP – c.d.

**Zwlekanie** (*linger*) – okres czasu do faktycznego zamknięcia połączenia, który umożliwia wysłanie danych zalegających w buforze wysyłkowym. Zwlekanie dla gniazda TCP ustanawiane jest przez nadanie wartości>0 opcji `SO_LINGER` za pomocą funkcją `setsockopt` przy użyciu struktury:

```
struct linger
  int l_onoff; /* 0 - wyłączone, nie 0 - włączone */
  int l_linger; /* czas zwlekania w sekundach */
};
```

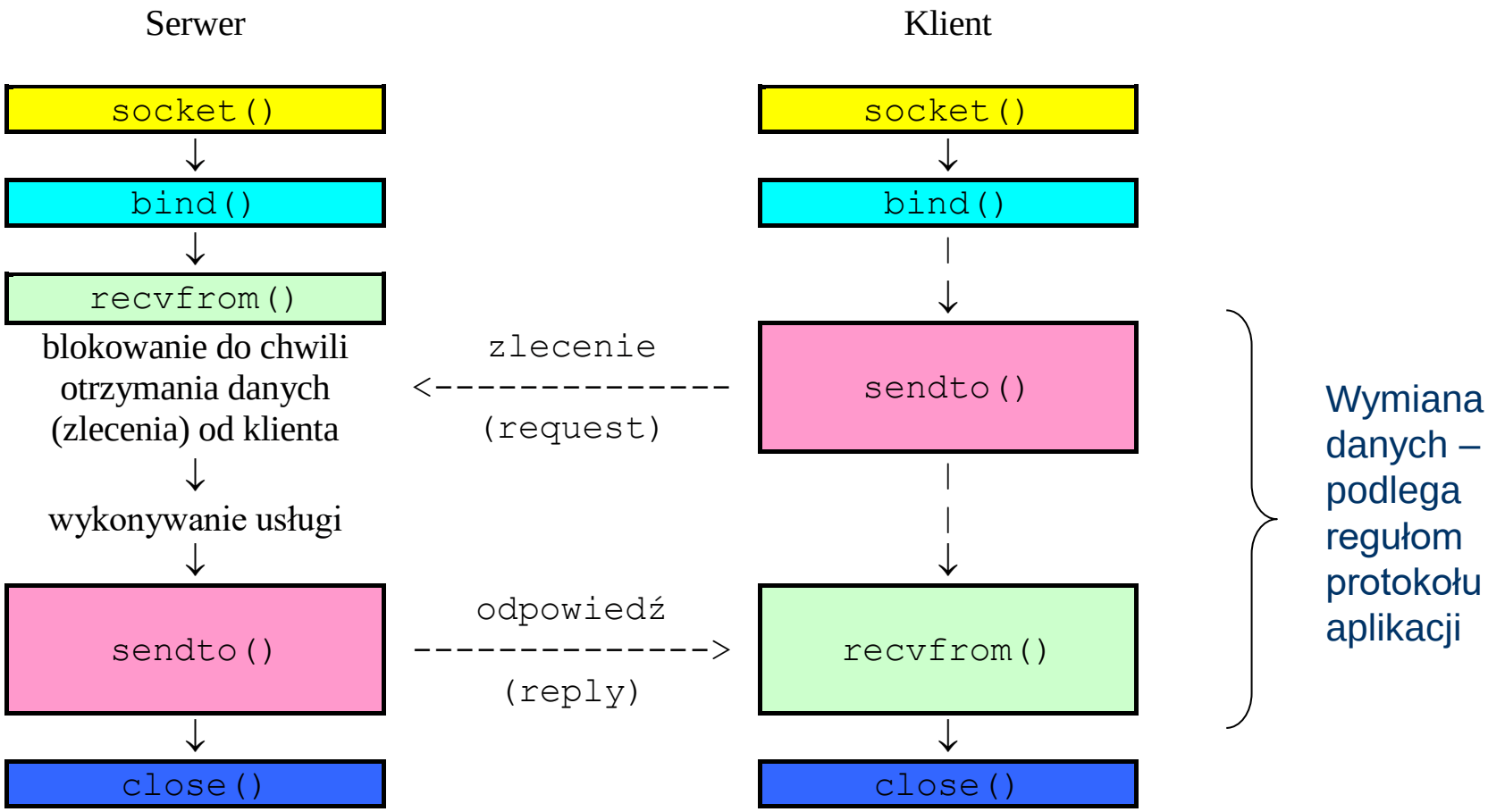
- Dla **`l_onoff=0`** funkcja **`close()`** natychmiast wraca, ale system próbuje przesłać zawartość bufora wysyłkowego do partnera; ginie zawartość bufora odbiorczego.
- Dla **`l_onoff!=0`** funkcje **close()** czy **shutdown()** wracają, jeżeli wszystkie dane z bufora wysyłkowego zostaną wysłane albo czas zwlekania (**`l_linger`**) się wyczerpał. Ginie zawartość bufora wysyłkowego i odbiorczego.

# Gniazda typu SOCK_STREAM - uwagi ogólne

- Gniazda SOCK_STREAM (połączeniowe, strumieniowe) tworzą niezawodne dwukierunkowe strumieniowe połączenie pomiędzy dwoma połączonymi gniazdami.
- Gniazdo typu SOCK_STREAM musi być połączone zanim może odebrać lub wysłać dane.
- Dane wysyłane przez gniazdo strumieniowe mogą być odbierane przez gniazdo połączone w porcjach o innych rozmiarach niż te użyte przy wysyłaniu. Strumień nie posiada rekordów logicznych.
- Dane mogą być (są) buforowane; zakończenie funkcji wysyłkowej nie oznacza że dane dotarły do odbiorcy, nawet że opuściły system nadawcy.
- Jeżeli dane nie mogą być doręczone przez określony czas – system nadawcy uznaje połączenie za przerwane i następne operacje wysyłki nie mogą być przeprowadzone pomyślnie. Sygnał SIGPIPE jest doręczany do procesu nadawcy, który próbuje wysłać dane do przerwanego strumienia. Sygnał nie jest generowany jeśli użyto flagi MSG_NOSIGNAL wywołując **send**( ), **sendto**( ), czy **sendmsg**( ).
- Wsparcie przesyłania danych pozapasmowych (OOB, pilnych) zależy od protokołu. Protokół TCP pozwala na przesłanie 1 oktetu (bajtu) OOB.

# Komunikacja za pomocą gniazd UDP

## Prosta komunikacja bezpołączeniowa (datagramowa) zlecenie-odpowiedź



# Gniazda UDP – przesyłanie danych

## Podstawowe funkcje wejścia/wyjścia

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

ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags,
  struct sockaddr *from, socklen_t *paddrlen);/* funkcja odbiorcza */

ssize_t sendto(int sockfd, const void *buff, size_t nbytes, int flags, const
  struct sockaddr *to, socklen_t addrlen); /* funkcja nadawcza */
```

Parametry:

**buff** – deskryptor gniazda

**buff** - wskaźnik na bufor o długości **nbytes**

**from** – jeśli nie NULL - wskaźnik na strukturę adresową (o długości *paddrlen) nadawcy datagramu otrzymanego przez **recvfrom()** . Aktualna długość adresu zostanie wpisana do *paddrlen.

**to** – wskaźnik struktury adresowej (o długości **addrlen**) odbiorcu datagramu wysyłanego przez **sendto()**

**flags** – sygnalizatory (jak dla **send()** i **recv()**)

Uwagi:

- Wysyłanie datagramów o długości 0 jest poprawne
- Dla bufora gniazda UDP wartość opcji SO_SNDBUF określa maksymalny rozmiar datagramu.
- Próba wysłania datagramu dłuższego od SO_SNDBUF powoduje błąd (errno==EMSGSIZE).
- Każdy datagram musi być wysłany (odebrany) w trakcie jednej operacji wejścia/wyjścia.

# Połączone gniazda UDP

Obserwacja: Funkcja **`write`**`()` nie ma argumentów, który może określać odbiorcę i nie może być prosto wykorzystana do komunikacji UDP.

Połączone gniazda UDP:

- Strony wywołują funkcję **`connect()`**
- Do wymiany danych używa się **`write()`** (lub **`send()`**) zamiast **`sendto()`** i **`read`** (lub `recv`) zamiast **`recvfrom`**
- Rozłączanie połączenia UDP – za pomocą **`connect()`** z deklaracją rodziny protokołów **`PF_UNSPEC`** (może wystąpić błąd nieimplementowanego protokołu, który należy pominąć).
- Czasowe połączenie gniazd zwiększa wydajność komunikacji – w przypadku wielu transmisji pomiędzy tymi samymi gniazdami.
- Połączenie gniazd umożliwia powiadamianie wysyłającego o tzw. błędzie asynchronicznym. Błąd ten polega na tym, że po wysłaniu datagramu za pomocą **sendto**() funkcja ta powraca po wstawieniu komunikatu do bufora wyjściowego. Jeżeli komunikatu nie można doręczyć (*port unreachable*) stacja docelowa odsyła komunikat ICMP. Komunikat ten jest niedostępny dla wysyłającego – jeżeli używał gniazda niepołączonego; może więc prowadzić do błędnego działania aplikacji.

# Pozyskiwanie adresów gniazd

- Pobieranie adresu lokalnego (zaadresowanego) lokalnego gniazda `sockfd`:

```
#include <sys/socket.h>
int_t getsockname(int sockfd, struct sockaddr *name,
                  socklen_t *namelen);
```

- Pobieranie adresu drugiej strony równorzędnego połączenia odbywającego się poprzez gniazdo `sockfd` (gniazdo TCP lub połączone UDP)

```
#include <sys/socket.h>
int getpeername(int sockfd , struct sockaddr *name,
                socklen_t *namelen);
```

**Uwagi**:

- Przed wywołaniem każdej z w/w funkcji zmienna `*namelen` musi być zainicjowana długością adresu struktury wskazywanej przez **name**. Wartość `*namelen` może być modyfikowana przez wywoływaną funkcję.

# Gniazda typu SOCK_DGRAM - uwagi ogólne

- Gniazda SOCK_DGRAM (datagramowe, bezpołączeniowe) umożliwiają zawodne przesyłanie pakietów danych (datagramów).
- Dane mogą być wysyłane do odbiorcy (lub odbiorców – dla trybu multicast czy broadcast) określonego w każdym wywołaniu funkcji wyjścia. Dane mogą być odbierane od więcej niż jednego nadawcy; adres nadawcy jest przy tym dostępny przy odbiorze jego datagramu.
- Aplikacja może również określić adres „korespondenta” (gniazdo połączone); wówczas wszystkie wywołania funkcji wyjścia, które nie określają odbiorcy są kierowane do „korespondenta”. Aplikacja może odbierać jedynie datagramy od „korespondenta”.
- Datagram musi być wysłane w trakcie jednego wywołania funkcji wyjścia (wysyłania) i musi być odebrany w trakcie wykonania jednej funkcji wejścia (odbioru) . Maksymalna wielkość datagramu zależy od protokołu.
- Dane mogą być (są) buforowane; zakończenie funkcji wysyłkowej nie oznacza że dane dotarły do odbiorcy, nawet że opuściły system nadawcy. Implementacja funkcji wysyłkowych powinna zapewniać wykrywanie możliwie dużej liczby możliwych błędów i ich sygnalizację przez wartość wyjścia funkcji.