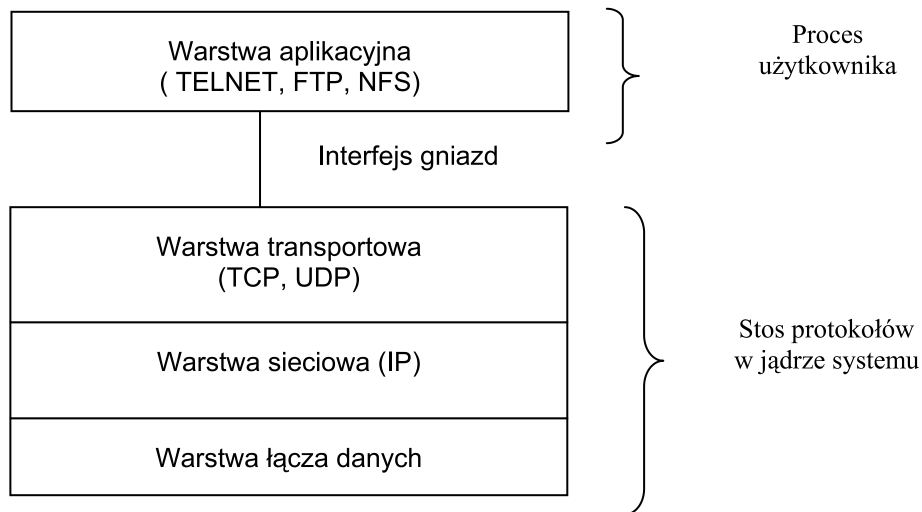
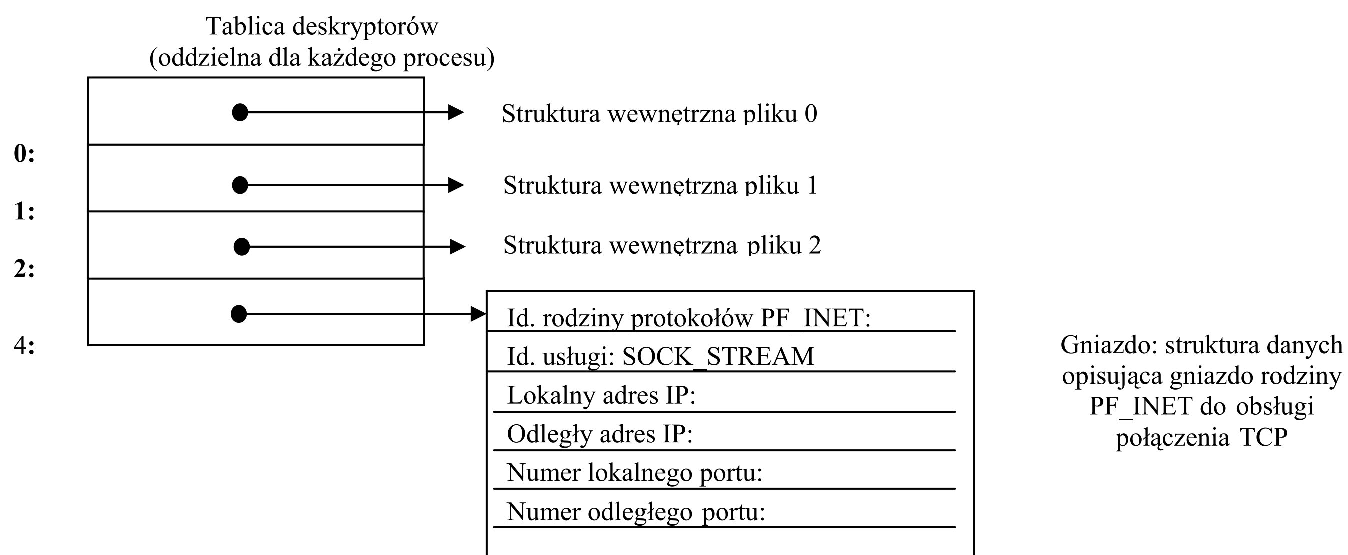
2. Interfejs gniazd

2.1. Gniazdo

- **Gniazdo** (ang. *socket*): pewna abstrakcja wykorzystywana do wysyłania lub otrzymywania danych z innych procesów. Pełni rolę punktu końcowego w linii komunikacyjnej.
- Interfejs gniazd to interfejs między programem użytkowym a protokołami komunikacyjnymi w systemie operacyjnym.

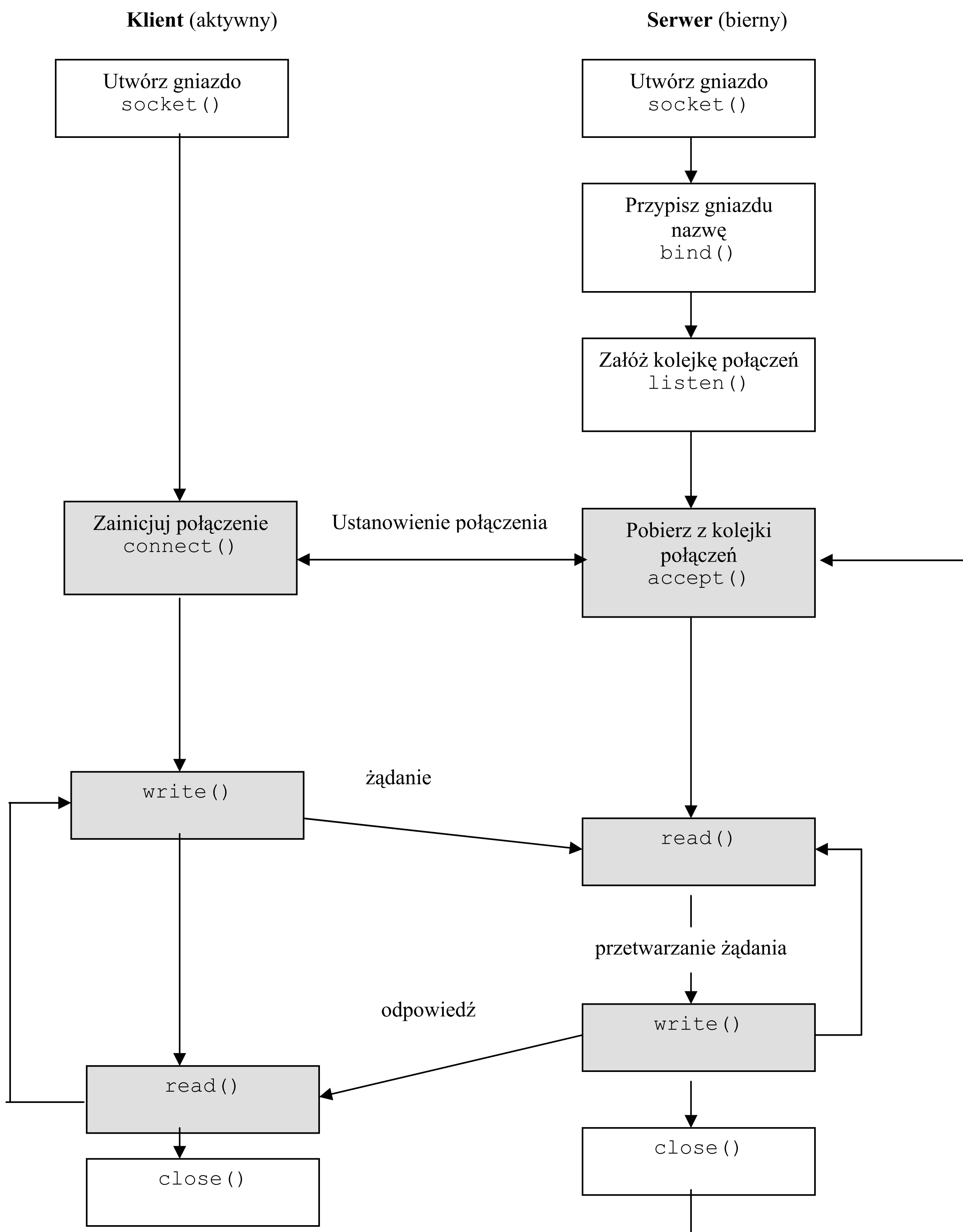


2.2. Gniazdo jako obiekt systemowy

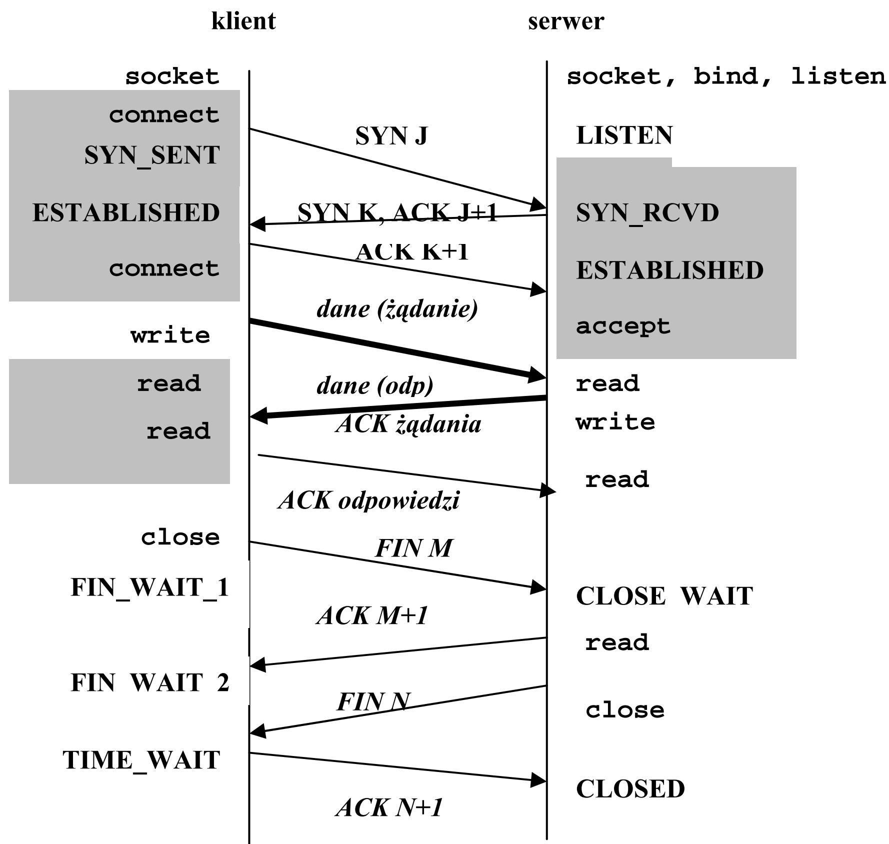


2.3. Przykład wykorzystania interfejsu gniazd: komunikacja serwer-klient oparta o TCP/IP

Serwer połączeniowy



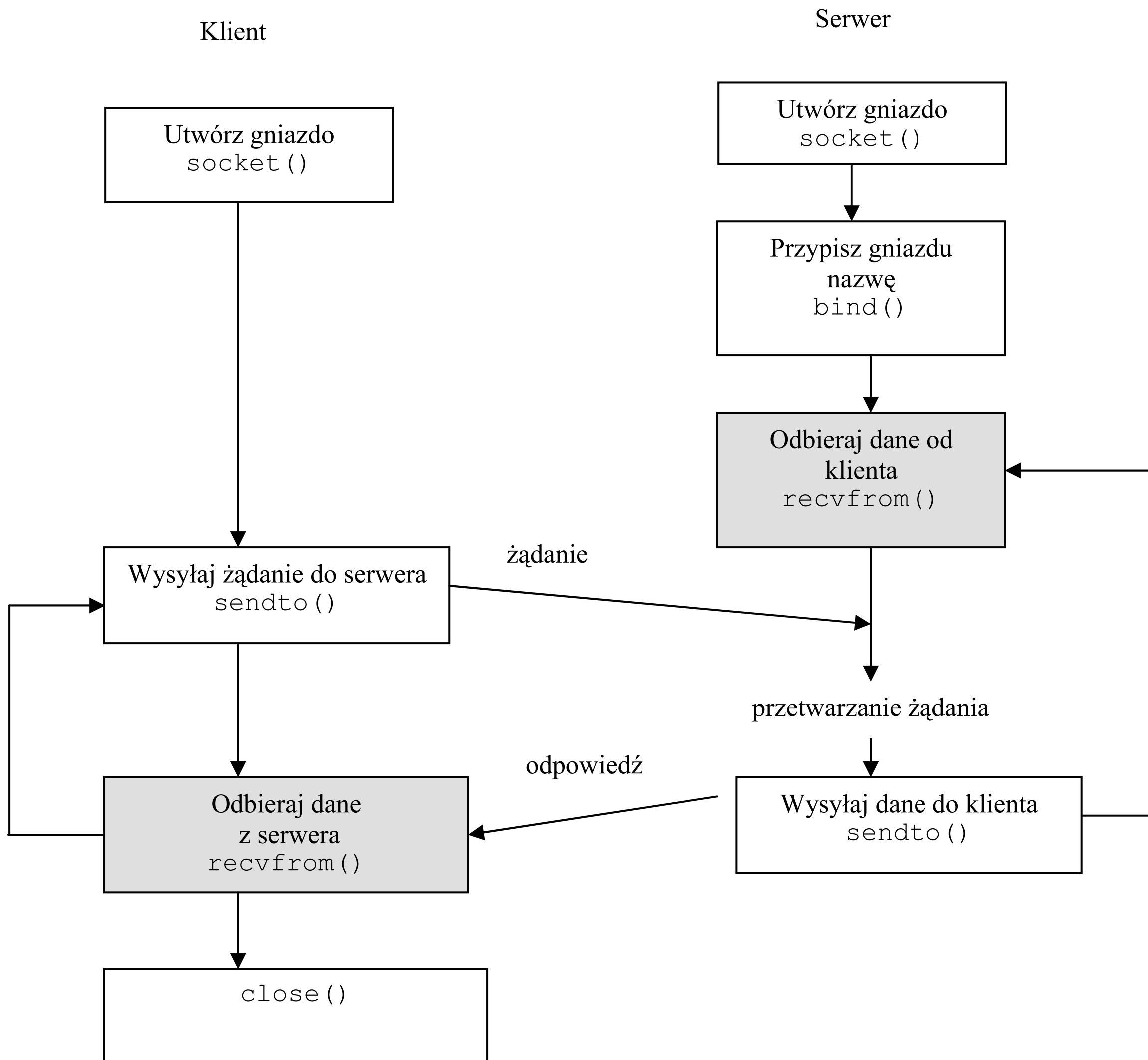
Wymiana pakietów przez połączenie TCP



Kody segmentów:

SYN	Zsynchronizuj numery porządkowe
ACK	Zawiera potwierdzenie
FIN	Koniec strumienia bajtów u nadawcy
RST	Skasuj połączenie
URG	Dane poza głównym strumieniem transmisyjnym (pozapasmowe)

Serwer bezpołączeniowy



2.4. Główne funkcje interfejsu gniazd

Funkcja	Opis
socket	Utworzenie gniazda (klient, serwer)
bind	powiązanie adresu lokalnego z gniazdem (serwer)
listen	przekształcenie gniazda niepołączonego w gniazdo bierne i założenie kolejki połączeń (serwer)
accept	przyjęcie nowego połączenia (serwer)
connect	nawiązanie połączenia klienta z serwerem
read	odbieranie danych z gniazda (klient, serwer)
write	przesyłanie danych do odległego komputera (klient, serwer)
recv	odbieranie danych z gniazda (klient, serwer)
send	przesyłanie danych do odległego komputera (klient, serwer)
recvfrom	odbieranie datagramu
sendto	wysyłanie datagramu
close	Zamknięcie gniazda (klient, serwer)
shutdown	zakończenie połączenia w wybranym kierunku (klient, serwer)
htons	zamiana liczby 16 bitowej na sieciową kolejność bajtów
ntohs	zamiana liczby 16 bitowej na kolejność bajtów hosta
htonl	zamiana liczby 32 bitowej na sieciową kolejność bajtów
ntohl	zamiana liczby 32 bitowej na kolejność bajtów hosta
inet_addr	konwersja adresu zapisanego w kropkowej notacji dziesiętnej na równoważną mu postać adresu binarnego 32 bitowego
inet_ntoa	konwersja adresu 32 bitowego zapisanego binarnie na adres w notacji kropkowej
inte_aton	konwersja adresu zapisanego w kropkowej notacji dziesiętnej na równoważną mu postać adresu binarnego 32 bitowego

- **`socket()`** - utworzenie gniazda (klient, serwer)

SKŁADNIA

```
#include <sys.types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

OPIS

- Funkcja `socket` tworzy nowe gniazdo komunikacyjne (czyli przydziela nową strukturę danych przeznaczoną do przechowywania informacji związanej z obsługą komunikacji; struktura ta będzie wypełniana przez kolejne funkcje). Funkcja zwraca deskryptor gniazda (liczba >0) lub –1 jeśli wystąpił błąd. Zmienna `errno` zawiera kod błędu.

- Parametry:
  - *`domain`* –rodzina protokołów komunikacyjnych używanych przez gniazdo (protokoły komunikacji lokalnej, IPv4, IPv6), opisana jest za pomocą odpowiedniej stałej: Przykłady:

Rodzina	Przykłady protokołów wchodzących do rodziny
`PF_INET` `PF_INET6`	protokół IPv4 protokół IPv6
`PF_UNIX`, `PF_LOCAL`	protokół UNIXa dla lokalnej komunikacji
inne	patrz opis funkcji `socket()`

  - *`type`* – typ żądanej usługi komunikacyjnej w ramach danej rodziny protokołów. Przykłady:

Typ	Znaczenie
`SOCK_STREAM`	połączenie strumieniowe (TCP)
`SOCK_DGRAM`	gniazdo bezpołączeniowe (datagramowe - UDP)
`SOCK_RAW`	gniazdo surowe
inne	patrz opis funkcji `socket()`

  - *`protocol`* – protokół transportowy w ramach rodziny; 0 oznacza protokół domyślny dla danej rodziny i danego typu usługi

Typ	Znaczenie
`IPPROTO_TCP`	połączenie strumieniowe (TCP)
`IPPROTO_UDP`	gniazdo bezpołączeniowe (datagramowe - UDP) gniazdo surowe

- Przykład:

```
/*  Utwórz gniazdo klienta: IPv4, TCP  */
int gniazdo_klienta;
if ( (gniazdo_klienta = socket(PF_INET, SOCK_STREAM, 0)) == -1)
  perror("Nie utworzono gniazda");

/*  Utwórz gniazdo klienta: IPv4, UDP  */
int gniazdo_klienta;
if ( (gniazdo_klienta = socket(PF_INET, SOCK_DGRAM, 0)) == -1)
  perror("Nie utworzono gniazda");
```

- **`close()`** - zamknięcie gniazda (klient, serwer)

SKŁADNIA

```
#include <unistd.h>
int close (int socket);
```

OPIS

- Funkcja close zamyka połączenie (w obu kierunkach) i usuwa gniazdo.
- Parametry:
  - *`socket`* – deskryptor zwalnianego gniazda
- Zwraca wartość 0, gdy operacja zakończona powodzeniem, w przeciwnym wypadku zwraca –1 i ustawia `errno`.
- Przykład:
  - W programie klienta:
    `close(gniazdo_klienta); //utworzone za pomocą socket()`
  - W programie serwera:
    `close(gniazdo_polaczone_z_klientem);  // utworzone za pomocą accept()`

Uwagi:

Funkcja `close` oznacza gniazdo o deskryptorze *`socket`* jako zamknięte, zmniejsza licznik odniesień do gniazda o 1 i natychmiast wraca do procesu. Proces nie może się już posługiwać tym gniazdem, ale warstwa TCP spróbuje wysłać dane z bufora wysyłkowego, po czym zainicjuje wymianę segmentów kończących połączenie TCP. Jednakże, jeśli po zmniejszeniu liczby odniesień do gniazda nadal jest ona > 0, nie jest inicjowana sekwencja zamykania połączenia TCP (wysłanie segmentu FIN).

- **`shutdown ()`** - zakończenie połączenia w wybranym kierunku (klient, serwer)

SKŁADNIA

```
#include <sys/socket.h>
int shutdown(int socket, int howto);
```

OPIS

- Funkcja oznacza gniazdo *`socket`* jako zamknięte w kierunku określonym drugim parametrem. Inicjuje sekwencję zamykania połączenia TCP bez względu na liczbę odniesień do deskryptora gniazda. Parametr *`howto`* może przyjmować wartości:
  - `SHUT_RD` (0) - proces nie może pobierać z gniazda danych (funkcja `read` zwróci 0), może nadal wysyłać dane przez gniazdo; kolejka danych wejściowych jest czyszczona, odebranie nowych danych do tego gniazda będzie potwierdzone, po czym dane zostaną odrzucone bez powiadamiania procesu; nie ma wpływu na bufor wysyłkowy (*Uwaga*: Winsock w tym przypadku działa inaczej - odnawia połączenie, jeśli przyjdą nowe dane)
  - `SHUT_WR` (1) - proces nie może wysyłać danych do gniazda (funkcja `write` zwróci kod błędu), może wciąż pobierać dane; dane znajdujące się w buforze wysyłkowym zostaną wysłane, po czym zainicjowana zostanie sekwencja kończąca połączenie TCP; nie ma wpływu na bufor odbiorczy
  - `SHUT_DRWR` (2) - zamykana jest zarówno część czytająca jak i część pisząca; równoważne kolejnemu wywołaniu `shutdown` z parametrem `how` równym 0 a następnie 1, nie jest równoważne wywołaniu `close`.
- Zwraca wartość 0, gdy operacja zakończona powodzeniem, w przeciwnym wypadku zwraca –1 i ustawia `errno`.

Uwagi: Funkcja `shutdown` jest przeznaczona tylko dla gniazd.

- **`connect()` -** nawiązanie połączenia z serwerem

SKŁADNIA

```
  #include <sys/socket.h>
  #include <sys/types.h>
int connect (int socket, struct sockaddr *serv_addr, unsigned int addrlen);
```

OPIS

- Funkcja `connect` nawiązuje połączenie z odległym serwerem. W przypadku połączenia TCP inicjuje trójfazowe uzgadnianie.
- Parametry:
  - *`socket`*– deskryptor gniazda, które będzie używane do połączenia
  - *`serv_addr`* – struktura adresowa, która zawiera adres serwera
  - *`addrlen`* – rozmiar adresu serwera
- Zwraca wartość 0, gdy operacja zakończona powodzeniem, w przeciwnym wypadku zwraca –1 i ustawia `errno`.

Struktury adresowe

- Każda rodzina protokołów posiada własne rodziny adresów. Adres gniazda ma znaczenie tylko w kontekście wybranej przestrzeni nazw rodziny adresów.
- Przykłady:
  - rodzina protokołów `PF_INET` : rodzina adresów `AF_INET` – 32 bitowy numer IP, 16 bitowy numer portu
  - rodzina protokołów `PF_UNIX` : rodzina adresów `AF_UNIX` – nazwa pliku zmiennej długości

Fragment przykładowego pliku zawierającego definicje stałych opisujących rozpoznawane rodziny protokołów i rodziny adresów `/usr/include/bits/socket.h` :

```
/* Protocol families.  */
#define PF_UNSPEC       0       /* Unspecified.  */
#define PF_LOCAL        1       /* Local to host (pipes and file-domain).  */
#define PF_UNIX         PF_LOCAL /* Old BSD name for PF_LOCAL.  */
#define PF_FILE         PF_LOCAL /* Another non-standard name for PF_LOCAL.  */
#define PF_INET         2       /* IP protocol family.  */
...
#define PF_INET6        10      /* IP version 6.  */
...
#define PF_BLUETOOTH    31      /* Bluetooth sockets.  */
#define PF_MAX          32      /* For now..  */

/* Address families.  */
#define AF_UNSPEC       PF_UNSPEC
#define AF_LOCAL        PF_LOCAL
#define AF_UNIX         PF_UNIX
#define AF_FILE         PF_FILE
#define AF_INET         PF_INET
...
#define AF_INET6        PF_INET6
...
#define AF_BLUETOOTH    PF_BLUETOOTH
#define AF_MAX          PF_MAX
```

- Adres reprezentowany jest za pomocą gniazdowej struktury adresowej.
- Gniazdowa struktura adresowa dla rodziny AF_INET:

```
#include <netinet/in.h>
struct sockaddr_in {
   unsigned short int  sin_family; /* typ adresu - stała */
   unsigned short int  sin_port;   /* numer portu, 16 bitów,
                                     w sieciowym porządku bajtów */
   struct in_addr      sin_addr;   /* adres IP */
   char sin_zero[8];               /* nie wykorzystywane */
};
struct in_addr {
     unsigned long int  s_addr; /* adres IP, 32 bity, w sieciowym porządku */
};
```

- Przykład dla rodziny AF_UNIX

```
#include <sys/un.h>
#define UNIX_PATH_MAX   108
struct sockaddr_un {
  unsigned short int sun_family; /* rodzina: AF_UNIX */
  char sun_path[UNIX_PATH_MAX]; /* nazwa ścieżkowa pliku */
};
```

Uogólniona struktura adresowa

- Problem:
  - Do funkcji działających na gniazdach trzeba przekazać wskaźnik do gniazdowej struktury adresowej właściwej dla danej rodziny protokołów.
  - Funkcje muszą działać dla dowolnej rodziny protokołów obsługiwanych przez system operacyjny.
  - Jak definiować typ wskaźnika przekazywanego do funkcji?

- Rozwiązanie: zdefiniowano ogólną gniazdową strukturę adresową:

```
#include <sys/socket.h>
struct sockaddr {
  unsigned short sa_family;     /* typ adresu AF_xxx */
  char sa_data[14];             /* adres właściwy dla protokołu */
}
```

  - W wywołaniu funkcji rzutuje się wskaźnik do właściwej dla danego protokołu gniazdowej struktury adresowej na wskaźnik do ogólnej gniazdowej struktury adresowej:
  - Jądro systemu może określić rodzaj struktury na podstawie wartości składowej `sa_family`.

	sa_family	sa_data		
sockaddr	rodzina adresów	zestaw bitów, których znaczenie zależy od rodziny adresów		
	2 bajty	2 bajty	4 bajty	8 bajtów
sockaddr_in	rodzina adresów	port	adres IP	nie wykorzystywane
	sin_family	sin_port	sin_addr	sin_zero

Przykład wypełnienia struktury adresowej i nawiązania połączenia

- Klient: łączy się z serwerem 127.0.0.1 na porcie 9001

```
int gniazdo_klienta;              /* deskryptor gniazda */
struct sockaddr_in serwer_adres;  /* adres serwera */

gniazdo_klienta=socket(PF_INET,SOCK_STREAM,0);

memset(&adres_serwera,0,sizeof(adres_serwera)); /* zerowanie struktury */
serwer_adres.sin_family = AF_INET;
serwer_adres.sin_addr.s_addr = inet_addr("127.0.0.1"); /* zamiana na binarny */
serwer_adres.sin_port = 9001; /* o ile sieciowa kolejność bajtów */
connect( gniazdo_klienta, (struct sockaddr*) &serwer_adres,
                           sizeof(serwer_adres));
```

Sieciowa kolejność bajtów

- Komputery stosują dwie różne metody wewnętrznej reprezentacji liczb całkowitych:
  - *najpierw starszy bajt* (ang. *big endian*) – najbardziej znaczący bajt słowa ma najniższy adres (czyli adres samego słowa)
  - *najpierw młodszy bajt* (ang. *little endian*) – najmniej znaczący bajt słowa ma najniższy adres (czyli adres samego słowa)
- Przykład: Mamy liczbę 17 998 720 (0x112A380)

1	18	163	128

Komputer A - *big endian*

128	163	18	1

Komputer B - *little endian*

- Przykłady:
  - sun, sunos4.1.4: big-endian
  - sun, solaris2.5.1: big-endian
  - hp, hp-ux10.20: big-endian
  - pc, linux: little-endian

- Rozwiązanie przyjęte dla informacji przesyłanych w sieci:
  - Kolejność bajtów właściwą dla danego systemu operacyjnego nazwano *systemową kolejnością bajtów*.
  - Do informacji przesyłanych w sieci (np. w nagłówkach protokołów TCP/IP) przyjęto standardową kolejność bajtów: najpierw starszy bajt (*big endian*). Kolejność tę nazwano *sieciową kolejnością bajtów*.
  - Funkcje sieciowe działają na liczbach (np. adres IP, numery portów) zapisanych w sieciowej kolejności bajtów.

Funkcje konwersji porządku szeregowania bajtów

- **Liczby całkowite krótkie (funkcje działają na liczbach 16 bitowych)**

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

/* host to network */
unsigned short htons (unsigned short host16);
uint16_t htons (uint16_t host16);

/* network to host */
unsigned short ntohs (unsigned short network16);
uunit16_t ntohs (uint16_t network16);
```

- **Liczby całkowite długie (funkcje działają na liczbach 32 bitowych)**

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

/* host to network */
unsigned long htonl (unsigned long host32)
uint32_t htonl (uint32_t host32)

/* network to host */
unsigned long ntohl (unsigned long network32)
uint32_t ntohl (uint32_t network32)
```

- **`inet_ntoa ()`**- konwersja adresu zapisanego binarnie na adres w notacji kropkowej

  SKŁADNIA

```
#include <sys/socket.h>
  #include <netinet/in.h>
  #include <arpa/inet.h>
  char *inet_ntoa(struct in_addr addr)
```

  OPIS
  - Funkcja `inet_ntoa` służy do konwersji adresu w postaci binarnej (sieciowy porządek bajtów) na odpowiadający mu napis (np. "187.78.66.23")
  - Parametry:
    - *`addr`* – struktura zawierająca 32-bitowy adres IP
  - Zwraca wskaźnik do napisu zawierającego adres w postaci kropkowej

  Struktura wykorzystywana w `inet_ntoa` ma postać:

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

- **`inet_addr()`** - konwersja adresu zapisanego w kropkowej notacji dziesiętnej na równoważną mu postać binarną

  SKŁADNIA

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
unsigned long inet_addr(const char *str)
```

  OPIS
  - Funkcja **`inet_addr`** służy do konwersji adresu IP podanego w kropkowej notacji dziesiętnej na równoważną mu postać binarną, uporządkowaną w sieciowej kolejności bajtów.
  - Parametry:
    - *`str`* – wskaźnik do napisu z adresem w notacji kropkowej
  - Zwraca binarną reprezentację adresu IP, lub –1 w przypadku błędu.

  - Problem:
    Funkcja ta zwraca stałą `INADDR_NONE` (zazwyczaj –1 czyli 32 jedynki), jeśli argument przesłany do tej funkcji nie jest poprawny. Oznacza to, że funkcja ta nie przekształci adresu 255.255.255.255 (ograniczone rozgłaszanie). Rozwiązanie: używanie funkcji `inet_aton`.

- **`inet_aton()`** - konwersja adresu zapisanego w kropkowej notacji dziesiętnej na równoważną mu postać binarną

  SKŁADNIA

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
  int inet_aton(const char *str, struct in_addr *addr);
```

  OPIS
  - Parametry:
    - *`str`* – wskaźnik do napisu z adresem w notacji kropkowej
    - *`addr`* - wskaźnik do struktury, w której zapisany zostanie binarny adres
  - Zwraca wartość różną od zera, jeśli konwersja się powiedzie, lub 0 w przypadku błędu.

- **`read()`** - odbieranie danych z gniazda (klient, serwer)

SKŁADNIA

```
#include <unistd.h>
  int read(int socket, void* buf, int buflen);
```

OPIS

- Funkcja **`read`** służy do odebrania danych wejściowych z gniazda.

ARGUMENTY

- *socket* - deskryptor gniazda
- *buf* - wskaźnik do bufora, do którego będą wprowadzone dane
- *buflen* – liczba bajtów w buforze *buf*

WARTOŚĆ ZWRACANA

- 0 - koniec pliku (zakończono przesyłanie),
- >0 - liczba przeczytanych bajtów
- –1 – wystąpił błąd; kod błędu jest umieszczany w `errno`

Przykład:

```
/* Połączenie TCP: odpowiedź może przyjść podzielona na części */
while ((n=read(gniazdo,bptr,bufdl))>0) {
  bptr +=n;  /* przesuń wskaźnik za wczytane dane */
  bufdl -=n; /* zmniejsz licznik wolnych miejsc */
}

/* Połączenie UDP */
n=read(gniazdo, bptr, bufdl);
```

- **`write()`** - przesyłanie danych do odległego komputera (klient, serwer)

SKŁADNIA

```
#include <unistd.h>
  int write(int socket, char* buf, int buflen);
```

OPIS

- Funkcja write służy do przesłania danych do komputera odległego.

ARGUMENTY

- *socket* - deskryptor gniazda
- *buf* - wskaźnik do bufora, który zawiera dane
- *buflen* – liczba bajtów w buforze `buf`

WARTOŚĆ ZWRACANA

- 0 - nie zostało wysłane
- >0 - liczba poprawnie przesłanych bajtów
- –1 - wystąpił błąd; kod błędu jest umieszczany w errno.

Przykład:

```
write(gniazdo,bptr,strlen(bufdl));
```

- **`bind()`** powiązanie adresu protokołu z gniazdem (serwer)

SKŁADNIA

```
#include <sys/types.h>
#include <sys/socket.h>
int bind (int socket, struct sockaddr *my_addr, int addrlen);
```

OPIS

- Funkcja `bind` przypisuje gniazdu lokalny adres protokołowy.
  - *`socket`* – deskryptor gniazda utworzonego przez funkcję `socket`
  - *`my_addr`* – adres lokalny (struktura), z którym gniazdo ma być związane
  - *`addrlen`* – rozmiar adresu lokalnego (struktury adresowej)
- Zwraca wartość 0, gdy operacja zakończona powodzeniem, w przeciwnym wypadku zwraca –1. Zmienna `errno` zawiera kod błędu.

Przykłady:

```
int gniazdo_serwera;
struct sockaddr_in serwer_adres;

/* Utwórz gniazdo */
gniazdo_serwera = socket(PF_INET, SOCK_STREAM, 0);

/* Ustal lokalny adres IP i numer portu  */
memset(&serwer_adres,0,sizeof(serwer.adres));
serwer_adres.sin_family = AF_INET;
serwer_adres.sin_addr.s_addr = htonl(INADDR_ANY);
serwer_adres.sin_port = htons(9001);

/* Przypisz gniazdu lokalny adres protokołowy */
bind(gniazdo_serwera, (struct sockaddr *) &serwer_adres,
                      (sizeof(serwer_adres));
```

- Zamiast adresu IP można użyć stałą `INADDR_ANY`. Pozwoli to akceptować serwerowi połączenie przez dowolny interfejs.

- **`listen()`** - przekształcenie gniazda niepołączonego w gniazdo bierne i założenie kolejki połączeń

SKŁADNIA

```
#include <sys/socket.h>
  int listen (int socket, int queuelen);
```

OPIS

- Funkcja `listen` ustawia tryb gotowości gniazda do przyjmowania połączeń (gniazdo bierne). Pozwala również ustawić ograniczenie liczby zgłoszeń połączeń przechowywanych w kolejce do tego gniazda wtedy, kiedy serwer obsługuje wcześniejsze zgłoszenie.
- Parametry:
  - *`socket`* – deskryptor gniazda związanego z lokalnym adresem
  - *`queuelen`* – maksymalna liczba oczekujących połączeń dla danego gniazda
- Zwraca wartość 0, gdy operacja zakończona powodzeniem, w przeciwnym wypadku zwraca –1. Zmienna `errno` zawiera kod błędu.
- Przykład:
  `listen(gniazdo_serwera, 5);`

- **`accept()`** - przyjęcie połączenia

SKŁADNIA

```
#include <sys/socket.h>
  int accept  (int socket, struct sockaddr *addr, int *addrlen);
```

OPIS

- Funkcja `accept` nawiązuje połączenie z klientem. W tym celu pobiera kolejne zgłoszenie połączenia z kolejki (albo czeka na nadejście zgłoszenia), tworzy nowe gniazdo do obsługi połączenia (gniazdo połączone) i zwraca deskryptor nowego gniazda.
- Parametry:
  - *`socket`* – deskryptor gniazda, przez które serwer przyjmuje połączenia (utworzonego za pomocą funkcji `socket`), tzw. gniazdo nadsłuchujące.
  - Serwer ma tylko jedno gniazdo nadsłuchujące (ang. *listening socket*)., które istnieje przez cały okres życia serwera.
  - Każde zaakceptowane połączenie z klientem jest obsługiwane przez nowe gniazdo połączone (ang. *connected socket*). Gdy serwer zakończy obsługę danego klienta, wtedy gniazdo połączone zostaje zamknięte.
  - *`addr`* – adres, który funkcja `accept` wypełnia adresem odległego komputera (klienta)
  - *`addrlen`* – rozmiar adresu klienta, wypełnia funkcja `accept`.
- Funkcja zwraca deskryptor nowego gniazda, które będzie wykorzystywane dla połączenia z klientem, zaś gdy operacja nie zakończy się powodzeniem wartość –1. Zmienna `errno` zawiera kod błędu.

```
int gniazdo_klienta;  /* z kolejki */
struct sockaddr_in klient_adres;
unsigned int dl_adres_klienta;
dl_adres_klienta=sizeof(klient_adres);
gniazdo_klienta=accept( gniazdo_serwera,
                        (struct sockaddr *) &gniazdo_klienta,
                        &dl_adres_klienta);
```

- **`sendto()`** - wysyłanie datagramu pobierając adres z odpowiedniej struktury

---

SKŁADNIA

```
int sendto(int sockfd, char *buff, int nbytes, int flags,
           struct sockaddr *to, int adrlen);
```

OPIS

Znaczenie argumentów:

- `sockfd` - deskryptor gniazda,
- `buff` - adres bufora zawierającego dane do wysłania,
- `nbytes` - liczba bajtów danych w buforze,
- `flags`- opcje sterowania transmisja lub opcje diagnostyczne,
- `to` - wskaźnik do struktury adresowej zawierającej adres punktu końcowego, do którego datagram ma być wysłany,
- `adrlen` - rozmiar struktury adresowej.

Funkcja zwraca liczbę wysłanych bajtów, lub -1 w przypadku błędu.

- **`recvfrom()`** - odbieranie datagramu wraz z adresem nadawcy

SKŁADNIA

```
int recvfrom(int sockfd, char *buff, int nbytes, int flags,
             struct sockaddr *from, int *adrlen);
```

OPIS

Znaczenie argumentów:

- `sockfd` - deskryptor gniazda,
- `buff` - adres bufora, w którym zostaną umieszczone otrzymane dane,
- `nbytes` - liczba bajtów w buforze,
- `flags`- opcje sterowania transmisja lub opcje diagnostyczne,
- `from` - wskaźnik do struktury adresowej, w której zostanie wpisany adres nadawcy datagramu,
- `adrlen` - rozmiar struktury adresowej.

Funkcja zwraca liczbę otrzymanych bajtów, lub -1 w przypadku błędu.

- **`recv()`** – odbieranie danych z gniazda (klient, serwer)

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```

Odbiera komunikat do połączonego hosta. Podobna do **`read`**, ale daje możliwość określenia opcji dla połączenia.

- **`send()`** – wysyłanie danych do zdalnego hosta (klient, serwer)

```
ssize_t send(int s, const void *msg, size_t len, int flags);
```

Wysyła komunikat do połączonego hosta. Podobna do **`write`**, ale daje możliwość określenia opcji dla połączenia.

2.5. Przykład klienta TCP usługi echo

```
#include <stdio.h>        /* printf(), fprintf(), perror() */
#include <sys/socket.h>   /* socket(), connect() */
#include <arpa/inet.h>    /* sockaddr_in, inetd_addr() */
#include <stdlib.h>       /* atoi(), exit() */
#include <string.h>       /* memset() */
#include <unistd.h>       /* read(), write(), close() */

#define BUFWE 32   /* Rozmiar bufora we */

int main(int argc, char *argv[]){
  int gniazdo;
  struct sockaddr_in echoSerwAdr;
  unsigned short echoSerwPort;
  char *serwIP;
  char *echoTekst;
  char echoBufor[BUFWE];
  unsigned int echoTekstDl;
  int bajtyOtrz, razemBajtyOtrz;

  if ((argc < 3) || (argc > 4)) {
    fprintf(stderr,
     "Uzycie: %s <Serwer IP> <Tekst> [<Echo_Port>]\n",
        argv[0]);
    exit(1);
  }

  serwIP = argv[1];
  echoTekst= argv[2];
  if (argc == 4)
    echoSerwPort = atoi(argv[3]);
  else
    echoSerwPort = 7;  /* standardowy port usługi echo */

  /* Utwórz gniazdo strumieniowe TCP */
  if ((gniazdo = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    { perror("socket() - nie udalo sie"); exit(1); }

  /* Zbuduj strukturę adresową serwera */
  memset(&echoSerwAdr, 0, sizeof(echoSerwAdr));
  echoSerwAdr.sin_family      = AF_INET;
  echoSerwAdr.sin_addr.s_addr = inet_addr(serwIP);
  echoSerwAdr.sin_port        = htons(echoSerwPort);

  /* Nawiąż połączenie z serwerem usługi echo */
  if (connect(gniazdo, (struct sockaddr *) &echoSerwAdr, sizeof(echoSerwAdr)) < 0)
  {
    perror("connect() - nie udalo sie");
    exit(1);
  }

  echoTekstDl = strlen(echoTekst);

  /* Prześlij tekst do serwera */
  if (write(gniazdo, echoTekst, echoTekstDl) != echoTekstDl)
  {
    perror("write() - przeslano zla liczbe bajtow");
    exit(1);
  }
```

```
  /* Odbierz ten sam tekst od serwera */
  razemBajtyOtrz = 0;
  printf("Otrzymano: ");
  while (razemBajtyOtrz < echoTekstDl) {
    if ((bajtyOtrz = read(gniazdo, echoBufor, BUFWE - 1)) <= 0)
    {
      perror("read() - nie udalo się lub polaczenie przedwczesnie zamknieto");
      exit(1);
     }
    razemBajtyOtrz += bajtyOtrz;
    echoBufor[bajtyOtrz] = '\0';
    printf(echoBufor);
  }

  printf("\n");

  /* Zamknij gniazdo */
  close(gniazdo);
  exit(0);
}
```

2.6. Przykład serwera połączeniowego usługi echo

```
#include <stdio.h>      /* printf(), fprintf() */
#include <sys/socket.h> /* socket(), bind(), connect() */
#include <arpa/inet.h>  /* sockaddr_in, inet_ntoa() */
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* read(), write(), close() */

#define MAXKOLEJKA 5
#define BUFWE 80

void ObslugaKlienta(int klientGniazdo);

int main(int argc, char *argv[])
{
  int serwGniazdo;
  int klientGniazdo;
  struct sockaddr_in echoSerwAdr; /* adres lokalny */
  struct sockaddr_in echoKlientAdr; /* adres klienta */
  unsigned short echoSerwPort;
  unsigned int klientDl; /* długość struktury adresowej */

  if (argc != 2)  {
    fprintf(stderr, "Użycie: %s <Serwer Port>\n", argv[0]);
    exit(1);
  }

  echoSerwPort = atoi(argv[1]);

/* Utwórz gniazdo dla przychodzących połączeń */
  if ((serwGniazdo = socket(PF_INET, SOCK_STREAM, 0)) < 0)
    { perror("socket() - nie udalo się"); exit(1); }

/* Zbuduj lokalną strukturę adresową */
  memset(&echoSerwAdr, 0, sizeof(echoSerwAdr));
  echoSerwAdr.sin_family = AF_INET;
  echoSerwAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  echoSerwAdr.sin_port = htons(echoSerwPort);

/* Przypisz gniazdu lokalny adres */
  if (bind(serwGniazdo,(struct sockaddr *) &echoSerwAdr,sizeof(echoSerwAdr)) < 0)
   { perror("bind() - nie udalo się"); exit(1); }

/* Ustaw gniazdo w trybie biernym - przyjmowania połączeń*/
  if (listen(serwGniazdo, MAXKOLEJKA) < 0)
   { perror("listen() - nie udalo się"); exit(1); }

/* Obsługuj nadchodzące połączenia */
  for (;;)   {
    klientDl= sizeof(echoKlientAdr);
    if ((klientGniazdo =  accept(serwGniazdo, (struct sockaddr *) &echoKlientAdr,
                             &klientDl)) < 0)
     { perror("accept() - nie udalo się"); exit(1); }

    printf("Przetwarzam klienta %s\n", inet_ntoa(echoKlientAdr.sin_addr));

    ObslugaKlienta (klientGniazdo);
  }
}
```

```
void ObslugaKlienta(int klientGniazdo)
{
  char echoBufor[BUFWE];
  int otrzTekstDl;

  /* Odbierz komunikat od klienta */
  otrzTekstDl = read(klientGniazdo, echoBufor, BUFWE);
  if (otrzTekstDl < 0)
  { perror("read() - nie udalo się"); exit(1); }

  /* Odeślij otrzymany komunikat i odbieraj kolejne
     komunikaty do zakończenia transmisji przez klienta */
  while (otrzTekstDl > 0)
  {
    /* Odeślij komunikat do klienta */
    if (write(klientGniazdo, echoBufor, otrzTekstDl) != otrzTekstDl)
     { perror("write() - nie udalo się"); exit(1); }

    /* Sprawdź, czy są nowe dane do odebrania */
    if ((otrzTekstDl = read(klientGniazdo,echoBufor,BUFWE)) < 0)
     { perror("read() - nie udalo się"); exit(1); }
  }

  close(klientGniazdo);
}
```

2.7. Przykład klienta UDP usługi echo

```
#include <stdio.h>      /* printf(),fprintf() */
#include <sys/socket.h> /* socket(), connect(), sendto(), recvfrom() */
#include <arpa/inet.h>  /* sockaddr_in, inet_addr() */
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* close() */
#define ECHOMAX 255     /* Najdluzszy przesylany tekst */

int main(int argc, char *argv[]) {
  int gniazdo;
  struct sockaddr_in echoSerwAdr;
  struct sockaddr_in echoOdpAdr;
  unsigned short echoSerwPort;
  unsigned int odpDl;
  char *serwIP;
  char *echoTekst;
  char echoBufor[ECHOMAX+1];
  int echoTekstDl;
  int odpTekstDl;

  if ((argc < 3) || (argc > 4))     {
    fprintf(stderr,"Użycie: %s <Seraer IP> <Tekst> [<Echo Port>]\n", argv[0]);
    exit(1);
  }
  serwIP = argv[1];
  echoTekst = argv[2];

  if ((echoTekstDl = strlen(echoTekst)) > ECHOMAX)
    { printf("Tekst zbyt dlugi\n"); exit(1); }

  if (argc == 4)
    echoSerwPort = atoi(argv[3]);
  else
    echoSerwPort = 7;

  if ((gniazdo = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    { perror("socket()"); exit(1);  }

  memset(&echoSerwAdr, 0, sizeof(echoSerwAdr));
  echoSerwAdr.sin_family = AF_INET;
  echoSerwAdr.sin_addr.s_addr = inet_addr(serwIP);
  echoSerwAdr.sin_port   = htons(echoSerwPort);

  if (sendto(gniazdo, echoTekst, echoTekstDl, 0, (struct sockaddr *)
             &echoSerwAdr, sizeof(echoSerwAdr)) != echoTekstDl)
   { perror("sendto()"); exit(1); }

  odpDl = sizeof(echoOdpAdr);
  if ((odpTekstDl = recvfrom(gniazdo, echoBufor, ECHOMAX, 0,
                       (struct sockaddr *) &echoOdpAdr, &odpDl)) != echoTekstDl)
    { perror("revcfrom()"); exit(1); }
  if (echoSerwAdr.sin_addr.s_addr != echoOdpAdr.sin_addr.s_addr)
    { fprintf(stderr,"Blad: pakiet z nieznanego zrodla.\n");
      exit(1);
    }

  echoBufor[odpTekstDl] = '\0';
  printf("Otrzymano: %s\n", echoBufor);

  close(gniazdo);
  exit(0);
}
```

2.8. Przykład serwera bezpołączeniowego usługi echo

```
#include <stdio.h>      /* printf(), fprintf() */
#include <sys/socket.h> /* socket(), bind() */
#include <arpa/inet.h>  /* sockaddr_in, inet_ntoa() */
#include <stdlib.h>     /* atoi() */
#include <string.h>     /* memset() */
#include <unistd.h>     /* close() */

#define ECHOMAX 255     /* Najdluzszy przesylany tekst */

int main(int argc, char *argv[]) {
  int gniazdo;
  struct sockaddr_in echoSerwAdr;
  struct sockaddr_in echoKlientAdr;
  unsigned int klientDl;
  char echoBufor[ECHOMAX];
  unsigned short echoSerwPort;
  int otrzTekstDl;

  if (argc != 2) {
     fprintf(stderr,"Użycie:  %s <UDP SERWER PORT>\n", argv[0]);
     exit(1);
  }

  echoSerwPort = atoi(argv[1]);

  if ((gniazdo = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    { perror("socket()"); exit(1); }

  memset(&echoSerwAdr, 0, sizeof(echoSerwAdr));
  echoSerwAdr.sin_family = AF_INET;
  echoSerwAdr.sin_addr.s_addr = htonl(INADDR_ANY);
  echoSerwAdr.sin_port = htons(echoSerwPort);

  if (bind(gniazdo, (struct sockaddr *) &echoSerwAdr, sizeof(echoSerwAdr)) < 0)
    { perror("bind()"); exit(1); }

  for (;;)  {
    klientDl = sizeof(echoKlientAdr);

    if ((otrzTekstDl = recvfrom(gniazdo, echoBufor, ECHOMAX, 0,
                        (struct sockaddr *) &echoKlientAdr, &klientDl)) < 0)
      { perror("recvfrom()"); exit(1); }

    printf("Przetwarzam klienta %s\n", inet_ntoa(echoKlientAdr.sin_addr));

    if (sendto(gniazdo, echoBufor, otrzTekstDl, 0,
         (struct sockaddr *) &echoKlientAdr, sizeof(echoKlientAdr)) !=
                                                              otrzTekstDl)
      { perror("sendto()"); exit(1); }
  }

}
```

2.8. Gniazda domeny uniksowej

- Rodzina protokołów dla gniazd domeny unksowej to: `PF_UNIX` (lub `PF_LOCAL`). Pozwala ona na łączenie gniazd na tym samym komputerze.
- Rodzina adresów dla tej domeny to: `AF_UNIX` (lub `AF_LOCAL`)
- Adresem gniazda jest nazwa ścieżkowa pliku. Jest to plik specjalny. Plik taki powstaje w momencie związywania gniazda z nazwą ścieżki (funkcja `bind`). Jeśli plik o podanej nazwie istnieje, `bind` zwraca błąd `EADDRINUSE`.
- Postać struktury adresowej dla rodziny `AF_UNIX`:

```
#include <sys/un.h>
#define UNIX_MAX_PATH   108
struct sockaddr_un {
  unsigned short int sun_family; /* rodzina: AF_UNIX */
  char sun_path[UNIX_PATH_MAX]; /* nazwa ścieżkowa pliku */
};
```

Uwaga: rozmiar adresu przekazywanego do funkcji gniazd, które tego wymagają powinna być równa liczbie znaków, z których składa się nazwa ścieżki, powiększonej o rozmiar pola `sun_family`. Istnieje makro SUN_LEN, które ten rozmiar oblicza.

- Informacje na temat gniazd domeny uniksowej: `man 7 unix`

Przykład:

```
#include <stddef.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <sys/un.h>
int make_named_socket (const char *filename)
{
  struct sockaddr_un name;
  int sock;
  size_t size;
  sock = socket (PF_LOCAL, SOCK_STREAM, 0); // lub PF_UNIX
  if (sock < 0)
  { perror ("socket");  exit (EXIT_FAILURE); }
/* Przypisz nazwe do gniazda */
  name.sun_family = AF_LOCAL;
  strncpy (name.sun_path, filename, sizeof (name.sun_path));
/* Oblicz rozmiar adresu
   size = (offsetof (struct sockaddr_un, sun_path)
                + strlen (name.sun_path) + 1);
   lub skorzystaj z makra SUN_LEN
*/
 size = SUN_LEN (&name);
  if (bind (sock, (struct sockaddr *) &name, size) < 0)
  { perror ("bind"); exit (EXIT_FAILURE); }
  return sock;
}
```

Przykład:

(M. Johnson, E. Troan: Oprogramowanie użytkowe w systemie Linux, WNT, 2000; str. 342-345)

Serwer iteracyjny domeny uniksowej. Serwer tworzy gniazdo domeny uniksowej `./gniazdo`. Pobiera dane z gniazda przesłane przez klienta i wyświetla je na standardowym wyjściu.

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

#include "sockutil.h"          /* funkcje pomocnicze */

int main(void) {
  struct sockaddr_un address;
  int sock, conn;
  size_t addrLength;

  if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
    die("socket");

  /* Usuń poprzednie gniazdo */
  unlink("./gniazdo");

  /* Utworz nowe gniazdo */
  address.sun_family = AF_UNIX;
  strcpy(address.sun_path, "./gniazdo");
  addrLength=SUN_LEN(&address);

  if (bind(sock, (struct sockaddr *) &address, addrLength))
    die("bind");
  if (listen(sock, 5))
    die("listen");

  while ((conn=accept(sock,(struct sockaddr *)&address,&addrLength)) >=0) {
    printf("---- odczyt danych\n");
    copyData(conn, 1);
    printf("---- koniec\n");
    close(conn);
  }

  if (conn < 0)
    die("accept");

  close(sock);
  return 0;
}
```

Klient domeny uniksowej. Klient łączy się z gniazdem domeny uniksowej `./gniazdo`. Pobiera dane wprowadzane przez klienta na STDIN i przesyła je do gniazda.

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

#include "sockutil.h"

int main(void) {
  struct sockaddr_un address;
  int sock;
  size_t addrLength;

  if ((sock = socket(PF_UNIX, SOCK_STREAM, 0)) < 0)
    die("socket");

  address.sun_family = AF_UNIX;
  strcpy(address.sun_path, "./gniazdo");
  addrLength = SUN_LEN(&address);

  if (connect(sock, (struct sockaddr *) &address, addrLength))
     die("connect");

  copyData(0, sock);

  close(sock);

  return 0;
}
```

Plik `sockutil.h`

```
/* sockutil.h */

void die(char * message);
void copyData(int from, int to);
```

Plik `sockutil.c`

```
/* sockutil.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "sockutil.h"

void die(char * message) {
  perror(message);
  exit(1);
}

void copyData(int from, int to) {
  char buf[1024];
  int amount;

  while ((amount = read(from, buf, sizeof(buf))) > 0) {
    if (write(to, buf, amount) != amount) {
       die("write");
    }
  }
  if (amount < 0)
    die("read");
}
```

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 72-87

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 24-52, 82-129, 248-265

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 87-88, 425-444