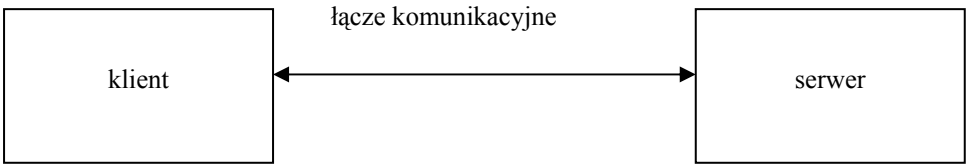
1. Model klient-serwer

1.1. Model komunikacji w sieci



Tradycyjny podział zadań:
- ***Klient*** – strona żądająca dostępu do danej usługi lub zasobu
- ***Serwer*** – strona, która świadczy usługę lub udostępnia zasoby
- Komunikacja między klientem i serwerem realizowana jest w warstwie aplikacji.

Typowy schemat pracy klienta:
1. Nawiązuje kontakt z serwerem
2. Wysyła do serwera żądanie wykonania usługi i czeka na odpowiedź
3. Po otrzymaniu odpowiedzi od serwera kontynuuje działanie.

Typowy schemat pracy serwera:
1. Rozpoczyna pracę i zasypia czekając na klienta, który się z nim skontaktuje.
2. Gdy otrzyma zlecenie klienta budzi się i wykonuje usługę.
3. Po zakończeniu wykonywania usługi zasypia i czeka na nadejście następnego żądania.

Rodzaje usług
- standardowe – zdefiniowane w standardach RFC
- niestandardowe – wszystkie inne

1.2. Klient

Charakterystyka klienta

- wywoływany przez użytkownika, który chce skorzystać z usługi
- aktywnie inicjuje kontakt z serwerem
- działa lokalnie na komputerze użytkownika
- nie wymaga specjalnych uprawnień systemowych
- może kontaktować się z wieloma serwerami, ale w danej chwili aktywnie komunikuje się tylko z jednym

Wymagania dla klienta

- Przykłady usług standardowych:
  - zdalnie działający terminal (ang. *remote terminal client*) - protokół TELNET
  - poczta elektroniczna (ang. *electronic mail client*) – protokół SMTP
  - przesyłanie plików między komputerami (ang. *file transfer client*) protokół FTP
- Parametryzacja - dostęp do serwerów wielu usług
- Przykład:

```
telnet oceanic.wsisiz.edu.pl        (telnet)
telnet oceanic.wsisiz.edu.pl 25     (smtp)
telnet oceanic.wsisiz.edu.pl 22     (ssh)
```

1.3. Serwer

Charakterystyka serwera

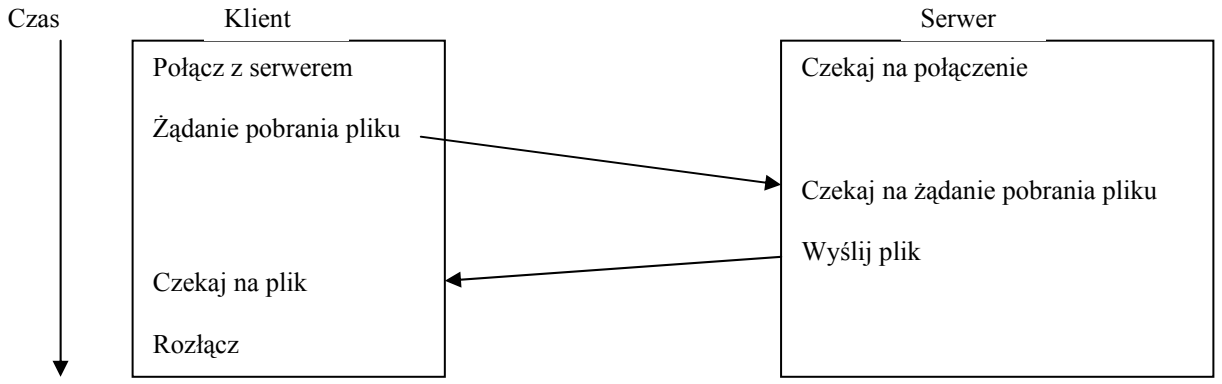
- uruchamiany automatycznie przy starcie systemu
- czeka pasywnie na zgłoszenia od dowolnych klientów
- działa na publicznie dostępnym komputerze
- jest specjalizowanym, uprzywilejowanym programem, którego zadaniem jest świadczenie konkretnej usługi
- zwykle świadczy jedną usługę, ale może obsługiwać wielu klientów

Wymagania dla serwera

- Działa w trybie uprzywilejowanym, jeśli musi mieć dostęp do zasobów chronionych
- Obsługa wielu klientów
  - sekwencyjna (serwer iteracyjny)
  - kilka zgłoszeń jednocześnie (serwer współbieżny)
- Zapewnienie bezpiecznej pracy:
  - **uwierzytelnianie** (ang. *authentication*) - sprawdzenie tożsamości klienta
  - **kontrola uprawnień** (ang. *autorization*) - sprawdzenie, czy dany klient ma prawo dostępu do usługi realizowanej przez serwer
  - **ochrona danych** (ang. *security*) - zabezpieczenie danych przez niezamierzonym naruszeniem lub ujawnieniem
  - **poufność** (ang. *privacy*) - zabezpieczenie przed dostępem nieupoważnionych użytkowników
  - **ochrona zasobów** (ang. *protection*) - zabezpieczenie zasobów systemowych przed niewłaściwym użyciem programów użytkowych działających w sieci

1.4. Protokół warstwy aplikacji

- Protokół jest to zbiór reguł, których muszą przestrzegać klient i serwer, aby mogły się ze sobą komunikować. Przykład:



- Przykład: protokół SMTP

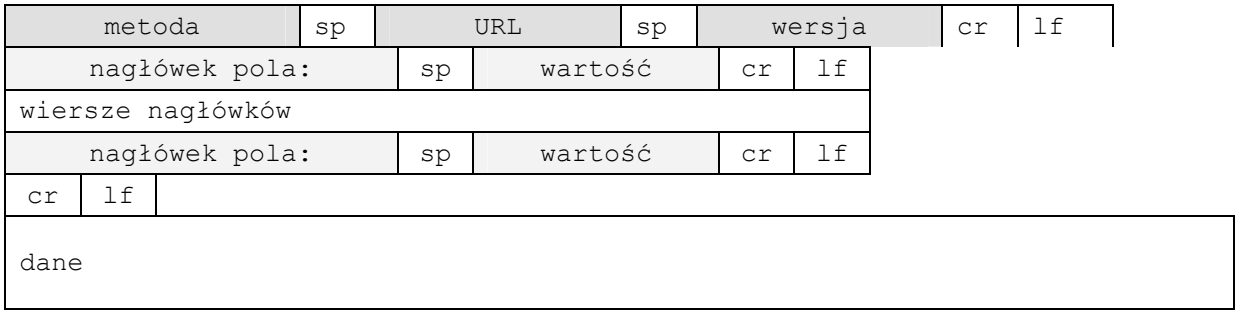
Klient	**Serwer**
(otwiera połączenie)	
	220 poczta.onet.pl Wed, 10 Sep 2004 20:45:41 +0200
EHLO wp.pl	
	250-poczta.onet.pl Hello wp.pl [213.135.45.70], pleased to meet you 250-ENHANCEDSTATUSCODES 250-8BITMIME 250-SIZE 250-DSN 250-ONEX 250-ETRN 250-XUSR 250 HELP
MAIL FROM:adam@wp.pl	
	250 2.1.0 <adam@wp.pl>... Sender ok
RCPT To: dorota@poczta.onet.pl	250 2.1.5 <dorota@poczta.onet.pl>... Recipient ok
DATA	354 Enter mail, end with "." on a line by itself
From: adam@wp.pl To: dorota@poczta.onet.pl Subject: Test Czesc!. .	
	250 2.0.0 h8AIjfe01432 Message accepted for delivery
QUIT	221 2.0.0 poczta.onet.pl closing connection
(Połączenie zamknięte)	

Każdy przesłany wiersz kończy się znakami CRLF (carriage return, line feed)

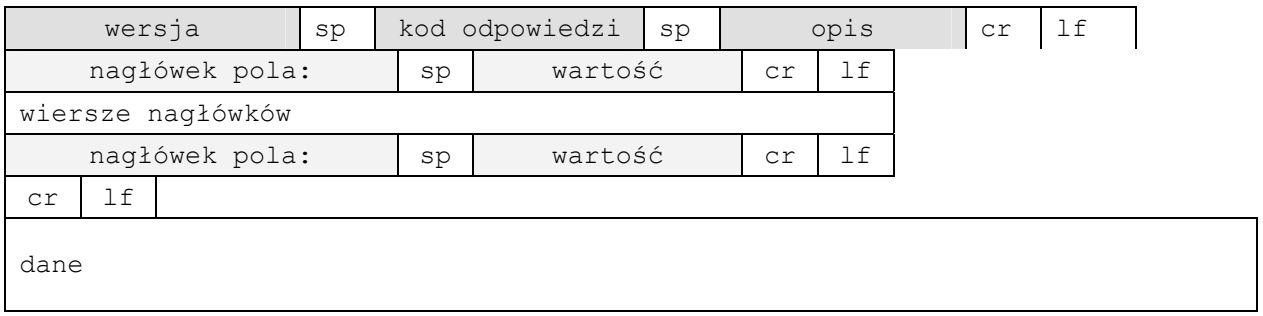
- Przykład: protokół HTTP – żądanie i odpowiedź

```
GET /projekt/index.html HTTP/1.1.
Host: oceanic.wsisiz.edu.pl
Connection: close

HTTP/1.1 200 OK
Date: Sat, 25 Feb 2006 09:09:01 GMT
Server: Apache
Last-Modified: Tue, 15 Nov 2005 14:54:42 GMT
Accept-Ranges: bytes
Content-Length: 3230
Connection: close
Content-Type: text/html

dane dane dane ....

Connection closed by foreign host.
```

- Ogólna postać żądania:



- Ogólna postać odpowiedzi:



1.5. Komunikacja - protokół transportowy: TCP czy UDP

Typy komunikacji

- Wyróżnia się następujące typy komunikacji:
  - strumieniowa (*stream*) – pomiędzy dwoma punktami końcowymi przesyłany jest strumień bajtów (w obydwu kierunkach). Na strumień ten nie jest nakładana żadna określona struktura. Wysyłane dane mogą być agregowane lub dzielone. Przykład protokołu strumieniowego: TCP
  - datagramowa (*datagram*) – wysyłany jest pojedynczy komunikat od nadawcy do odbiorcy, bez nawiązywania połączenia. Przykład protokołu datagramowego: UDP
  - rozgłoszeniowa (*broadcast*) – komunikat jest wysyłany do wszystkich jednostek w sieci, nie trzeba ustalać odbiorcy, oparty jest na UDP.
  - rozgłoszeniowa ograniczona (*multicast*) – komunikat jest wysyłany do grupy odbiorców, trzeba taką grupę zdefiniować, oparty jest na UDP.

Protokoły warstwy transportowej

- Komunikacja jest realizowana w oparciu o protokoły warstwy transportowe, takie jak. TCP i UDP.
  - Protokół TCP – połączeniowy, niezawodny, dane są przesyłane w postaci strumienia bajtów
  - Protokół UDP – bezpołączeniowy, zawodny, dane przesyłane w postaci datagramów określonej długości, możemy je traktować tak jak rekordy
- Serwer:
  - ***połączeniowy*** (ang. *connection-oriented*) – oparty na TCP
  - ***bezpołączeniowy*** (ang. *connectionless*) – oparty na UDP
- O wyborze decyduje protokół aplikacyjny:
  - niezawodność
  - uporządkowanie pakietów
  - sterowanie przepływem
  - pełny duplex
  - narzuty czasowe
- Przykład: kiedy zazwyczaj stosuje się protokół UDP
  - wymaga tego protokół aplikacji – ale wtedy zawiera mechanizmy zapewniające niezawodność
  - czasochłonność obsługi lub opóźnienia uniemożliwiają poprawne działanie aplikacji
  - protokół aplikacji korzysta z trybu rozgłaszania

  - Przykłady:

Aplikacja	Protokół warstwy aplikacji	Protokół transportowy
Poczta	SMTP (RFC 2821)	TCP
Zdalny dostęp do terminala	Telnet (RFC 854)	TCP
WWW	HTTP(RFC 2616)	TCP
Zdalne przesyłanie plików	FTP (RFC 959)	TCP
Zdalny serwer plików	NFS (McKusnik 1996)	UDP lub TCP
Strumieniowe usługi multimedialne	Często własność producenta	UDP lub TCP

1.6. Protokół warstwy aplikacji

- Projektowanie własnego protokołu wymaga podjęcia takich decyzji jak:
  - format wymienianych komunikatów,
  - czy serwer będzie bierny czy aktywny,
  - czy serwer ma przetwarzać polecenia pojedynczo,
  - czy protokół będzie wymagał nawiązania sesji,
  - czy protokół będzie tekstowy czy binarny ,
  - czy będzie uwzględniał uwierzytelnianie,
  - czy będzie zapewniał poufność.

Należy przeczytać:

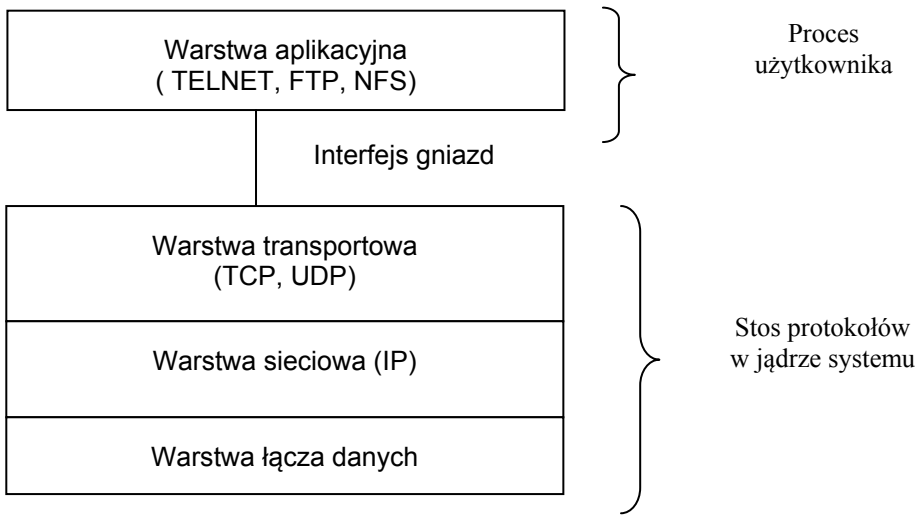
Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 35-48

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 82-137

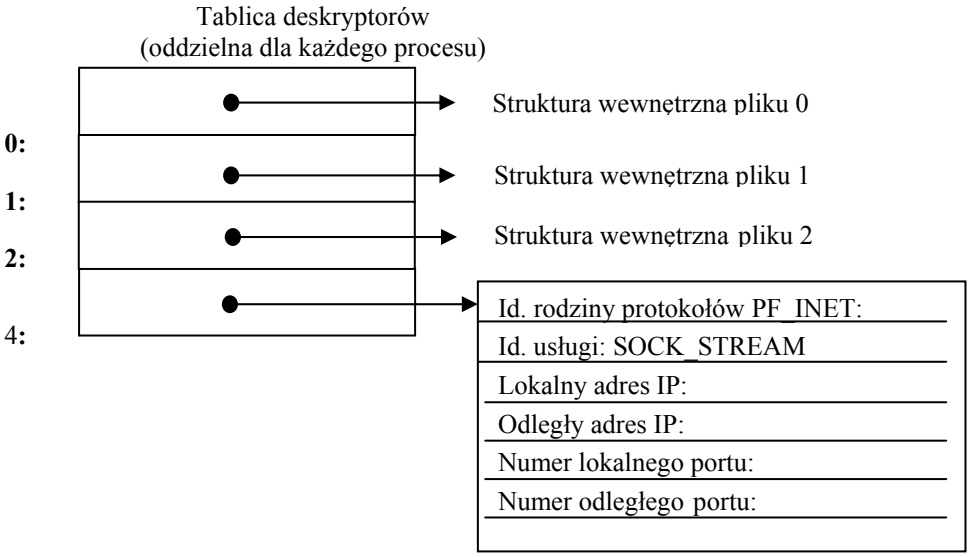
2. Interfejs gniazd

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.



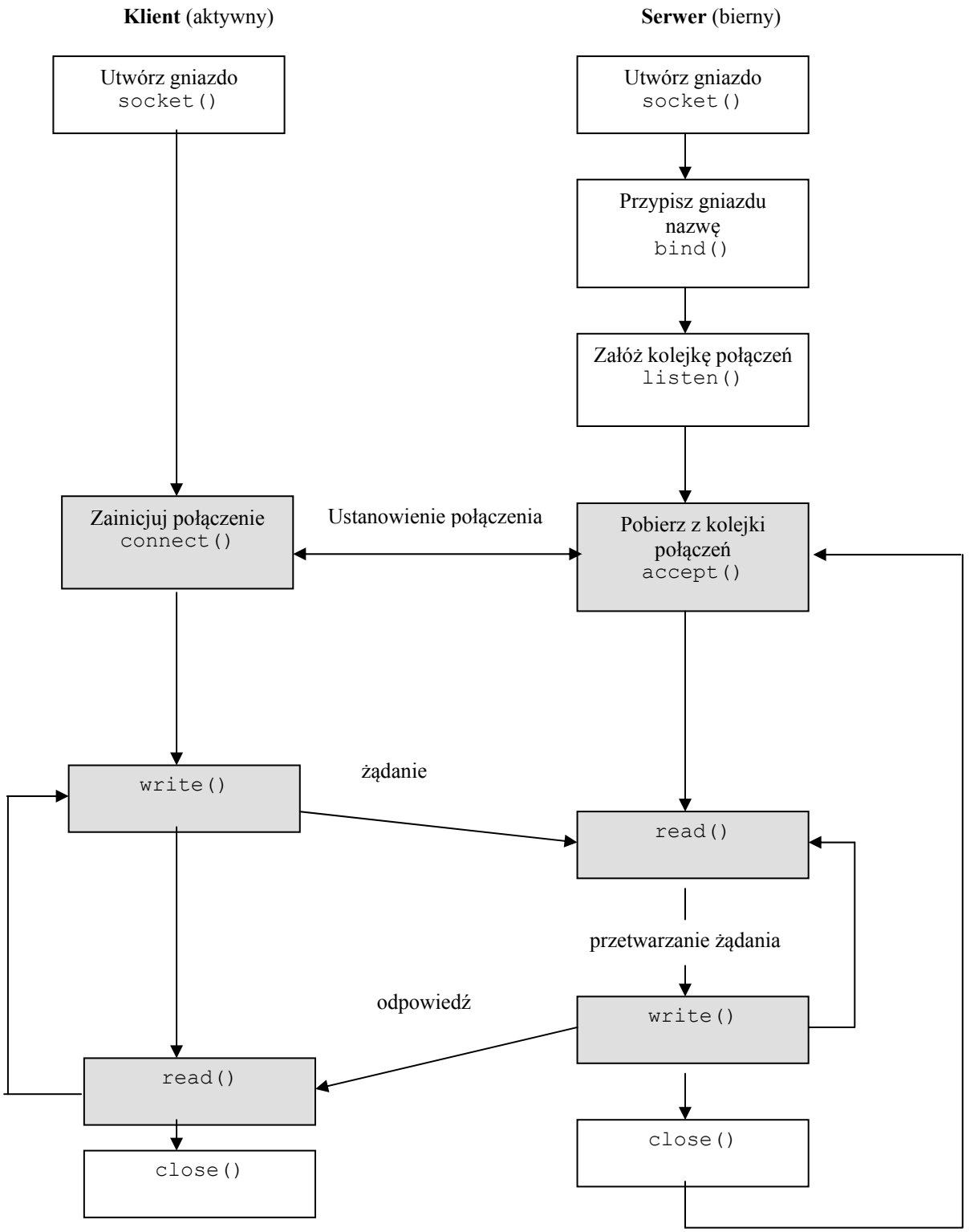
2.2. Gniazdo jako obiekt systemowy



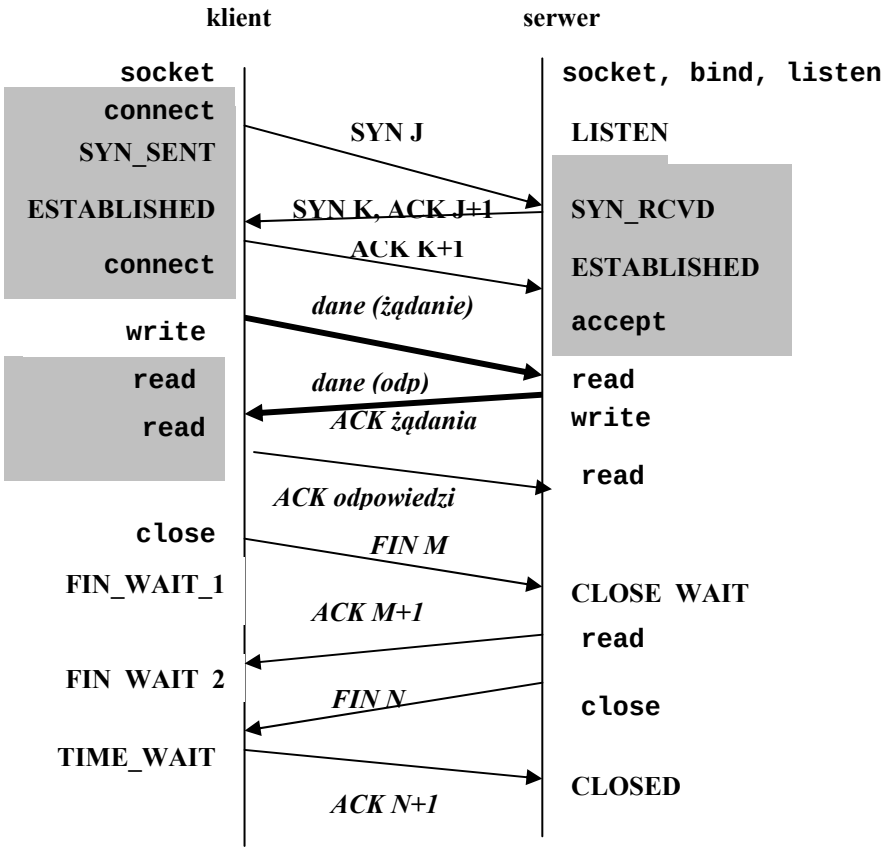
Gniazdo: struktura danych opisująca gniazdo rodziny PF_INET do obsługi połączenia TCP

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

Serwer połączeniowy



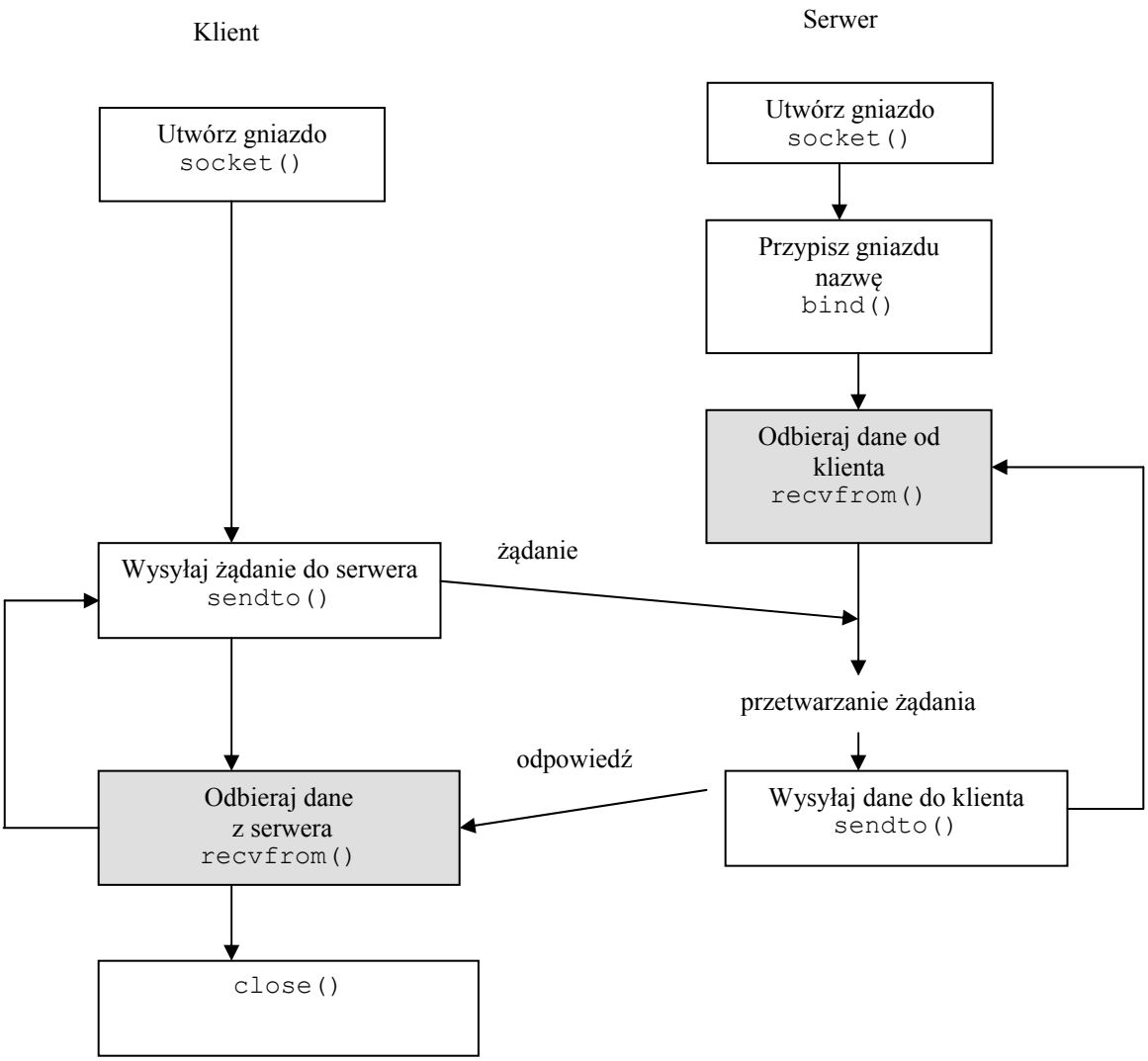
Wymiana pakietów przez połączenie TCP



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`	protokół IPv4
`PF_INET6`	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 */
};
```

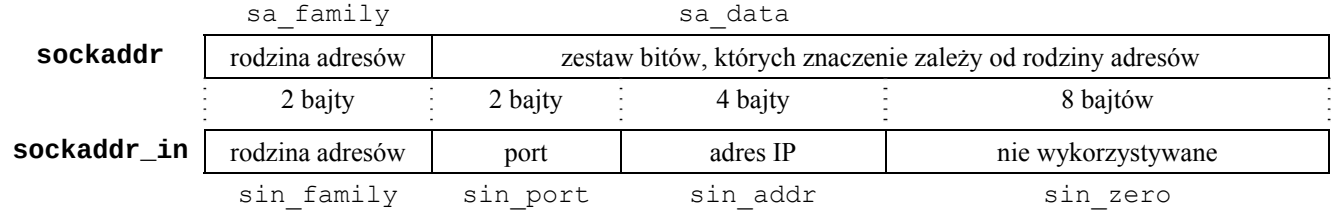
Uogólniona struktura adresowa

- 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`.



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

3. Identyfikacja

3.1. Określanie adresu połączonego hosta

SKŁADNIA

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

OPIS

- Funkcja `getpeername` dostarcza adresu drugiej strony połączenia.
- Parametry:
  - *`socket`* – deskryptor gniazda, przez które obsługiwane jest połączenia.
  - *`addr`* – adres, który funkcja `getpeername` wypełnia adresem odległego komputera
  - *`addrlen`* – rozmiar adresu, wypełnia funkcja `getpeername`.
- Funkcja zwraca 0, jeśli operacja zakończy się powodzeniem, zaś gdy operacja nie powiedzie się wartość –1. Zmienna `errno` zawiera kod błędu.

```
struct sockaddr_in adres;
int dl_adres;
dl_adres=sizeof(adres);
if (getpeername(gniazdo, (struct sockaddr *) &adres, &dl_adres) != 0)
  perror("getpeername()");
else
  printf("Dane partnera: %s:%d\n",
          inet_ntoa(adres.sin_addr),ntohs(adres.sin_port));
```

3.2. Określanie adresu lokalnego hosta

SKŁADNIA

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

OPIS

- Funkcja `getsockname` dostarcza adresu lokalnej strony połączenia.
- Parametry:
  - *`socket`* – deskryptor gniazda, przez które obsługiwane jest połączenia.
  - *`addr`* – adres, który funkcja `getsockname` wypełnia adresem lokalnego komputera
  - *`addrlen`* – rozmiar adresu, wypełnia funkcja `getsockname`.
- Funkcja zwraca 0, jeśli operacja zakończy się powodzeniem, zaś gdy operacja nie powiedzie się wartość –1. Zmienna `errno` zawiera kod błędu.

```
struct sockaddr_in adres;
int dl_adres;
dl_adres=sizeof(adres);
if (getsockname(sock, (struct sockaddr *) &adres, &dl_adres) != 0)
  perror("getsockname()");
else
  printf("Dane lokalne: %s:%d\n",
          inet_ntoa(adres.sin_addr),ntohs(adres.sin_port));
```

Uwaga: w kliencie należy wywołać funkcję po `connect()`.

3.3. Określanie nazwy lokalnego hosta

SKŁADNIA

```
#include <unistd.h>
int gethostname(char *name, size_t len);
```

OPIS

- Funkcja `gethostname` dostarcza nazwę lokalnego hosta.
- Parametry:
  - *`name`* – bufor, w którym będzie umieszczona nazwa
  - *`len`* – rozmiar bufora
  - *`addrlen`* – rozmiar adresu, wypełnia funkcja `getpeername`.
- Funkcja zwraca 0, jeśli operacja zakończy się powodzeniem, zaś gdy operacja nie zakończy się powodzeniem wartość –1. Zmienna `errno` zawiera kod błędu.

```
char nazwa[50];
if (gethostname(nazwa,sizeof(nazwa)) != 0)
  perror("gethostname()");
else
  printf("Nazwa mojego hosta: %s\n",nazwa);
```

3.4. Odwzorowanie nazwy domenowej na adres IP - `gethostbyname()`

SKŁADNIA

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

OPIS

- Funkcja `gethostbyname` pobiera ciąg znaków ASCII reprezentujący nazwę domenową komputera (parametr *name*) i zwraca wskaźnik do wypełnionej struktury `hostent` zawierającej między innymi 32-bitowy adres komputera. Jeśli funkcja `gethostbyname` zostanie wywołana z adresem IP zamiast nazwy, to w strukturze IP wypełniona zostanie tylko jedna składowa – pierwsza pozycja `h_addr_list`.
- Funkcja po poprawnym wykonaniu zwraca wskaźnik do struktury `hostent`, w przypadku wystąpienia błędu zwracana jest wartość NULL i ustawiana jest zmienna globalna `h_errno`. Informację o błędzie można uzyskać za pomocą funkcji `herror()`.
- Definicja struktury `hostent` (plik `netdb.h`):

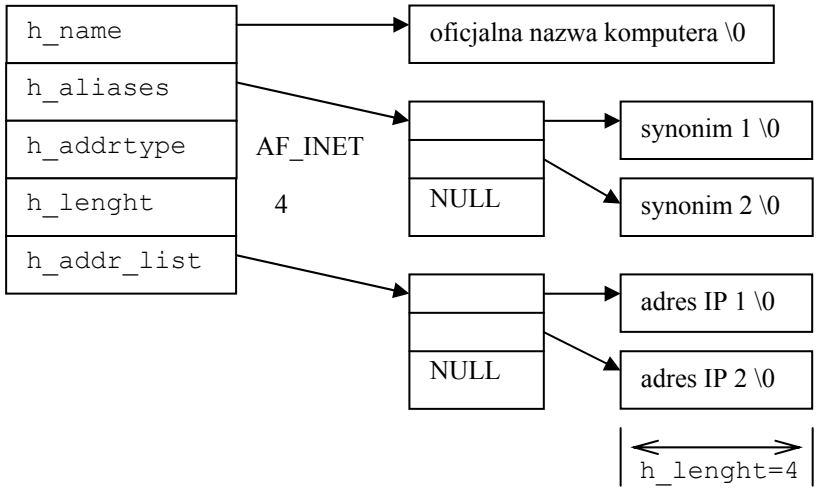
```
struct hostent {
  char *h_name;       /* oficjalna nazwa domenowa komputera */
  char **h_aliases;   /* synonimy */
  int h_addrtype;     /* typ adresu  */
  int    h_length;    /* długość adresu */
  char **h_addr_list; /* lista adresów */
};
/* pierwszy adres w tablicy adresów */
#define h_addr h_addr_list[0]
```

gdzie:

`h_name` wskazuje napis zawierający oficjalną nazwę domenową hosta
`h_aliases` jest wskaźnikiem do tablicy wskaźników zawierających inne nazwy hosta
`h_addrtype` jest określa typ adresu (stała `AF_INET` dla IPv4)
`h_length` określa długość (w bajtach) adresów w `h_addr_list` (dla IPv4 będzie zawsze równy 4)
`h_addr_list` zawiera listę adresów IP związanych z oficjalną nazwą, adresy przechowywane są w postaci sieciowej kolejności bajtów. Adresy pobierane są z pliku `/etc/hosts`, bazy NIS lub DNS – w zależności od konfiguracji.

W pliku nagłówkowym zdefiniowano również nazwę `h_addr` jako odwołanie do pierwszego elementu listy adresów IP hosta, wykorzystywane w starszym oprogramowaniu.

***Struktura* `hostent`**



- Przykład

```
struct hostent *hptr;
char *nazwa = "oceanic.wsisiz.edu.pl";
if (hptr = gethostbyname(nazwa))
{
  /* umieszczono adres IP w hptr->h_addr */
}
else
{
/* błąd w nazwie - odpowiednie działania */
}
```

- Przykład funkcji, która zwraca adres IP w postaci binarnej

```
unsigned long OdwzorujNazwe(char nazwa[])
{
  struct hostent *host; // funkcja gethostbyname zwraca
                        // wskaźnik
  if ((host = gethostbyname(nazwa)) == NULL) {
    fprintf(stderr, "gethostbyname(): nie udało się");
    exit(1);
  }
  // zwroc adres IP w postaci binarnej
   return *((unsigned long *) host->h_addr_list[0]);
}
```

- Przykład funkcji, która wstawia adres IP do składowej struktury adresowej sockaddr_in

```
int resolve_name( struct sockaddr_in *addr, char *hostname )
{
  addr->sin_family = AF_INET;
  addr->sin_addr.s_addr = inet_addr(hostname);
  if ( addr->sin_addr.s_addr == 0xffffffff ) {
    struct hostent *hp;
    hp = (struct hostent *)gethostbyname( hostname );
    if (hp == NULL) return -1;
    else {
      memcpy( (void *)&addr->sin_addr,
              (void *)hp->h_addr_list[0],
              sizeof(addr->sin_addr) );
    }
  }
  return 0;
}
```

3.5. Odwzorowanie adresu IP na nazwę hosta

SKŁADNIA

```
#include <netdb.h>
#include <sys/socket.h>        /* for AF_INET */
struct hostent *gethostbyaddr(const char *addr, int len, int type);
```

OPIS

- Funkcja `gethostbyaddr` pobiera adres IP (parametr *`addr`*) i zwraca wskaźnik do wypełnionej struktury `hostent` zawierającej między innymi nazwę domenową hosta.
- Argumenty:
  - `addr` - wskaźnik do tablicy zawierającej adres IP (uwaga: adres jest liczbą binarną)
  - `len` - rozmiar adresu (4 dla IPv4)
  - `type` - typ adresu (AF_INET dla adresu IPv4)

- Przykład:

```
#include <netdb.h>

struct hostent *host;
struct in_addr in;
inet_aton("213.135.44.33", &in);
if ( (host=gethostbyaddr((char *)&in.s_addr,
                         sizeof(in.s_addr),AF_INET))) {
  printf("Host name is %s\n",host->h_name);
```

3.6. Nowe funkcje odwzorowujące

- Standard Posix wprowadza nowe funkcje odwzorowujące:
  - `getaddrinfo()` - dostarcza struktury adresowej właściwej dla protokołu (nazwa na adres IP)
  - `getnameinfo()` - łączy funkcjonalność `gethostbyaddr()` i `gethostbyport()`(adres IP na nazwę)

- Przykład 1 - Informacje uzyskiwane z `gethostbyname`

```
#include <stdio.h>          //printf() perror()
#include <string.h>         //strcmp()
#include <unistd.h>         //gethostname()
#include <netdb.h>          //gethostbyname() gethostbyaddr()
#include <sys/socket.h>     //AF_INET inet_aton() inet_ntoa()
#include <netinet/in.h>
#include <arpa/inet.h>

int main()
{
  struct hostent *host;
  struct in_addr in;
  int i;

  host=gethostbyname("www.microsoft.com");
  if (host == NULL) return 1;
  else {
    printf("h_name is %s\n", host->h_name);
    printf("h_addrtype is %d\n", host->h_addrtype);

    i=0;
    printf("Aliases:\n");
    while (1) {
      if (host->h_aliases[i]) {
        printf("h_aliases[%d] = %s\n",i,host->h_aliases[i]);
        i++;
      } else break;
    }

    i=0;
    printf("Addresses:\n");
    while (1) {
      if (host->h_addr_list[i]) {
        memcpy(&in.s_addr,host->h_addr_list[i],sizeof(in.s_addr));
        printf("h_addr_list[%d] = %s\n",i,inet_ntoa(in));
        i++;
      } else break;
    }
  }
  return 0;
}
```

- Przykład 2: Zamiana nazwy domenowej na adres IP

```
#include <netdb.h>
#include <stdio.h>          //printf() perror()
#include <string.h>         //strcmp()
#include <unistd.h>         //gethostname()
#include <netdb.h>          //gethostbyname() gethostbyaddr()
#include <sys/socket.h>           //AF_INET inet_aton() inet_ntoa()
       #include <netinet/in.h>
       #include <arpa/inet.h>
int resolve_name( struct sockaddr_in *addr, char *hostname );
int main(int argc, char *argv[])
{
  struct sockaddr_in addr;
  int ret;
  ret = resolve_name(&addr, argv[1]);
  if (ret==0) {
    printf("address is %s\n", inet_ntoa(addr.sin_addr));
  } else return 1;
  return 0;
}

int resolve_name( struct sockaddr_in *addr, char *hostname )
{
  addr->sin_family = AF_INET;
  addr->sin_addr.s_addr = inet_addr( hostname );
  if ( addr->sin_addr.s_addr == 0xffffffff ) {
    struct hostent *hp;
    hp = (struct hostent *)gethostbyname( hostname );
    if (hp == NULL) return -1;
    else {
      memcpy( (void *)&addr->sin_addr,
              (void *)hp->h_addr_list[0],
              sizeof( addr->sin_addr ) );
    }
  }
  return 0;
}
```

3.7. Odwzorowanie nazwy usługi na numer portu - `getservbyname()`

SKŁADNIA

```
#include <netdb.h>
  struct servent *getservbyname(const char *name, const char *proto);
  struct servent *getservbyport(int port, const char *proto);
```

OPIS

- Funkcja `getservbyname` pobiera dwa napisy reprezentujące odpowiednio nazwę usługi oraz nazwę protokołu komunikacyjnego warstwy transportowej i zwraca wskaźnik do struktury `servent`. W przypadku wystąpienia błędu (brak definicji usługi w pliku `/etc/services`) zwracana jest wartość NULL.
- Funkcja `getservbyport` dokonuje odwzorowania numery portu na nazwę usługi.
- Definicja struktury `servent` (plik `netdb.h`):

```
struct servent
{
   char  *s_name;    /* oficjalna nazwa usługi */
   char **s_aliases; /* synonimy */
   int    s_port;    /* port dla tej usługi w sieciowym porządku
                        bajtów */
   char  *s_proto;   /* protokół, którego należy użyć */
};
```

- Przykład

```
struct servent *sptr;
if  (sptr = getservbyname("smtp","tcp"))
{
  /* numer portu umieszczono sptr->s_port */
}
else {
  /* błąd – odpowiednie działania */
}
```

- UWAGA: Numer portu w strukturze `servent` reprezentowany jest w sieciowym porządku bajtów. Aby poprawnie odczytać jego wartość należy dokonać konwersji do postaci obowiązującej na lokalnym komputerze (funkcja `ntohs`).
- Przykład funkcji, która zwraca numer portu w postaci binarnej

```
unsigned short OdwzorujUsluge(char usluga[],
                              char protokol[])
{
  struct servent *usl;
  unsigned short port;

/* Czy port podany jako liczba?  */
  if ((port = atoi(usluga)) == 0)  {
    if ((usl = getservbyname(usluga, protokol)) == NULL)
    {
      fprintf(stderr, "getservbyname() failed");
      exit(1);
    }
    else
      port = usl->s_port;
  }
  else
    port = htons(port);
  return port;
}
```

3.8. Odwzorowanie nazwy protokołu na jego numer - `getprotobyname()`

SKŁADNIA

```
#include <netdb.h>
  struct protoent *getprotobyname(const char *name);
  struct protoent *getprotobynumber(int proto);
```

OPIS

- Funkcja `getprotobyname` pobiera napis reprezentujący nazwę protokołu i zwraca wskaźnik do struktury `protoent` zawierającej między innymi liczbę całkowitą przypisaną temu protokołowi. W przypadku wystąpienia błędu (brak nazwy protokołu w pliku `/etc/protocols`) zwracana jest wartość NULL.
- Funkcja `getprotobynumber` dokonuje odwzorowania numeru protokołu na nazwę.
- Definicja struktury `protoent` (plik `netdb.h`):

```
struct protoent
{
   char  *p_name;     /* oficjalna nazwa protokołu */
   char **p_aliases;  /* synonimy */
   int    p_proto;    /* oficjalny numer protokołu */
};
```

- Przykład

```
struct protoent *pptr;
if  (pptr = getprotobyname("udp"))
{
  /* numer protokołu umieszczono w pptr->p_proto */
}
else
{
  /* błąd - odpowiednie działania */
}
```

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 89-96

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 277-299

4. Algorytmy serwera

4.1. Typy serwerów

- Serwer iteracyjny (ang. *iterative server*) obsługuje zgłoszenia klientów sekwencyjnie, jedno po drugim.
- Serwer współbieżny (ang. *concurrent server*) obsługuje wiele zgłoszeń jednocześnie.

- Serwer połączeniowy (ang. *connection-oriented server*) używa protokołu TCP.
- Serwer bezpołączeniowy (ang. *connectionless server*) używa protokołu UDP.

- Serwer wielostanowy (ang. *stateful server*) pamięta stany interakcji z klientami.
- Serwer bezstanowy (ang. *stateless server*) nie przechowuje informacji o stanie interakcji z klientem.

TCP a UDP

- Protokół TCP:
  - zapewnia połączenie typu punkt-punkt
  - gwarantuje niezawodność połączenia - klient albo ustanawia połączenie z serwerem albo gdy żądanie połączenia nie może być zrealizowane otrzymuje zwrotny komunikat
  - gwarantuje niezawodność przesyłania danych - po nawiązaniu połączenia dane są dostarczane w takiej samej kolejności w jakiej zostały wysłane, nie są gubione ani powielane; jeśli połączenie zostanie zerwane, nadawca jest o tym informowany.
  - steruje szybkością przepływu danych - nadawca nie może szybciej wysyłać danych niż odbiorca jest w stanie je odebrać
  - działa w trybie full-duplex - pojedynczy kanał może służyć do równoczesnego przesyłania danych w dwóch kierunkach, klient może przesyłać dane do serwera zaś serwer do klienta
  - jest protokołem strumieniowym - do odbiorcy przesyłany jest strumień bajtów, nie musi być on tak samo zgrupowany jak był wysłany, możliwe jest za to przesyłanie dużych bloków danych

- Protokół UDP:
  - nie wymaga utworzenia połączenia, nie potrzebuje narzutu czasu związanego z tworzeniem i utrzymywaniem połączenia
  - umożliwia połączenia typu wiele-wiele
  - jest protokołem zawodnym - komunikat może być zagubiony, powielony, dostarczony w innej kolejności
  - nie ma mechanizmu kontroli przesyłania danych - jeśli datagramy napływają szybciej niż mogą być przetworzone, są odrzucane bez powiadamiania o tym użytkownika
  - jest protokołem datagramowym - nadawca określa liczbę bajtów do wysłania i taką samą liczbę bajtów otrzymuje odbiorca w jednym komunikacie; rozmiar komunikatu jest ograniczony.

Serwer iteracyjny a serwer współbieżny

- Czas przetwarzania zgłoszenia przez serwer (T_P, ang. *request processing time*) to całkowity czas trwania obsługi jednego, wyizolowanego zgłoszenia.
- Obserwowany czas odpowiedzi dla klienta (T_R, ang. *observed processing time*) to całkowity czas upływający od wysłania głoszenia przez klienta do chwili uzyskania odpowiedzi serwera.
- T_R nigdy nie jest krótszy od T_P , a może być dużo dłuższy, gdy serwer ma do obsłużenia kolejkę zgłoszeń.

- Przykładowe kryteria wyboru:

  - Średni obserwowany czas odpowiedzi serwera iteracyjnego:

    $$T_R = (N/2 + 1)\ T_P$$

    gdzie N - średnia długość kolejki zgłoszeń. Można założyć, że jeśli mała kolejka w serwerze iteracyjnym nie wystarczy, to należy użyć serwera współbieżnego.

  - Czas przetwarzania zgłoszenia T_P dla serwera iteracyjnego powinien być mniejszy niż:

    $$T_{P\,MAX} = 1/KR$$

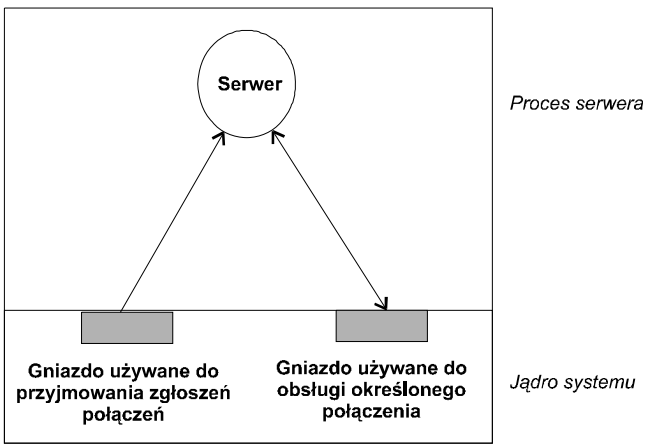
    gdzie: K - liczba klientów, R - liczba zgłoszeń nadsyłanych przez pojedynczego klienta w ciągu sekundy. Jeśli warunek ten nie zostanie spełniony, należy rozważyć serwer współbieżny.

  - Serwer współbieżny poprawi czas odpowiedzi, jeśli:
    - przygotowanie odpowiedzi wymaga wielu operacji we-wy
    - czas przetwarzania jest zróżnicowany dla różnych żądań
    - serwer jest uruchomiony na maszynie wieloprocesorowej

4.2. Algorytm działania iteracyjnego serwera połączeniowego (TCP)

Zasada: *jeden proces kolejno obsługuje połączenia z poszczególnymi klientami*.

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnianej przez serwer (funkcje `socket`, `bind`)
2. Ustaw bierny tryb pracy gniazda (funkcja `listen`).
3. Przyjmij kolejne zgłoszenie połączenia nadesłane na adres tego gniazda i uzyskaj przydział nowego gniazda do obsługi tego połączenia (funkcja `accept`).
4. Odbieraj kolejne zapytania od klienta, konstruuj odpowiedzi i wysyłaj je do klienta zgodnie z protokołem zdefiniowanym w warstwie aplikacji.
5. Po zakończeniu obsługi danego klienta zamknij połączenie i wróć do kroku 3, aby przyjąć następne połączenie.



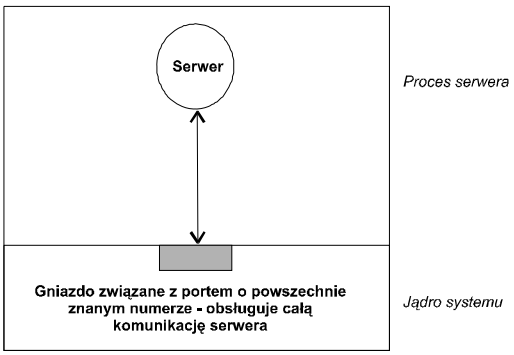
- Serwer połączeniowy:
  - wykorzystuje wszystkie zalety protokołu TCP

  ale
  - potrzebuje czasu na nawiązanie połączenia TCP i jego zakończenie
  - dla każdego połączenia tworzy oddzielne gniazdo
  - nie przesyła pakietów przez połączenie, które jest bezczynne

4.3. Algorytm działania iteracyjnego serwera bezpołączeniowego (UDP)

Zasada: *jeden proces kolejno obsługuje zapytania od poszczególnych klientów.*

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnianej przez serwer (funkcje `socket`, `bind`).
2. Odbieraj kolejne zapytania od klientów, konstruuj odpowiedzi i wysyłaj je zgodnie z protokołem warstwy aplikacji.



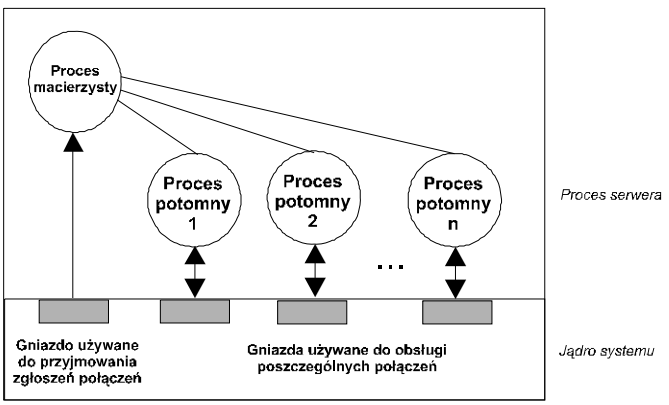
- Serwer bezpołączeniowy
  - nie jest narażony na wyczerpanie zasobów
  - pozwala na pracę w trybie rozgłaszania

  ale
  - trzeba wbudować mechanizmy gwarantujące niezawodność (część może być po stronie klienta)

4.4. Algorytm działania współbieżnego serwera połączeniowego (TCP)

Zasada: proces główny serwera przyjmuje zgłoszenia połączeń i tworzy nowe procesy potomne lub wątki do obsługi każdego połączenia; proces potomny po wykonaniu usługi zamyka połączenie.

Proces główny, krok 1.	Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze realizowanej przez serwer..
Proces główny, krok 2.	Ustaw bierny tryb pracy gniazda, tak aby mogło być używane przez serwer.
Proces główny, krok 3.	Przyjmuj kolejne zgłoszenia połączeń od klientów posługując się funkcją `accept`; dla każdego połączenia utwórz nowy proces potomny lub wątek, który przygotuje odpowiedź.
Proces potomny/wątek, krok 1.	Rozpoczynając działanie, przejmij od procesu głównego nawiązane połączenie (tzn. gniazdo przeznaczone dla tego połączenia).
Proces potomny/wątek, krok 2.	Korzystając z tego połączenia, prowadź interakcję z klientem; odbieraj zapytania i wysyłaj odpowiedzi.
Proces potomny/wątek, krok 3.	Zamknij połączenie i zakończ się. Proces potomny/wątek kończy działanie po obsłużeniu wszystkich zapytań od jednego klienta.



- Implementacja współbieżności za pomocą procesów potomnych:
  - Kod procesu macierzystego i potomnego może być:
    - zawarty w jednym programie
    - kod procesu potomnego może być zawarty w odrębnym programie uruchamianym za pomocą funkcji z rodziny `exec`.
  - Należy zwrócić uwagę na czyszczenie po procesach potomnych i nie pozostawianie zombi.
  - Zalety: łatwy w implementacji, żaden z klientów nie może zmonopolizować serwera, załamanie jednego procesu potomnego nie wpływa na inne.
  - Wady: utworzenie nowego procesu jest czasochłonne, trudne porozumiewanie się procesów między sobą (brak wspólnej pamięci), duży program zużywa znaczące zasoby.

- Implementacja współbieżności za pomocą wątków
  - Zalety: mniejszy czas potrzebny na utworzenie nowego wątku, współdzielenie pamięci, różne metody synchronizacji,
  - Wady: mniejsza stabilność w porównaniu do procesów potomnych – błąd w jednym wątku może mieć wpływ na cały serwer, ograniczenie liczby wątków dla jednego programu

4.5. Algorytm działania wieloprocesowego współbieżnego serwera bezpołączeniowego (UDP)

Zasada: proces macierzysty (główny) serwera przyjmuje zapytania od klientów i tworzy nowe procesy potomne/wątki do obsługi każdego zapytania.

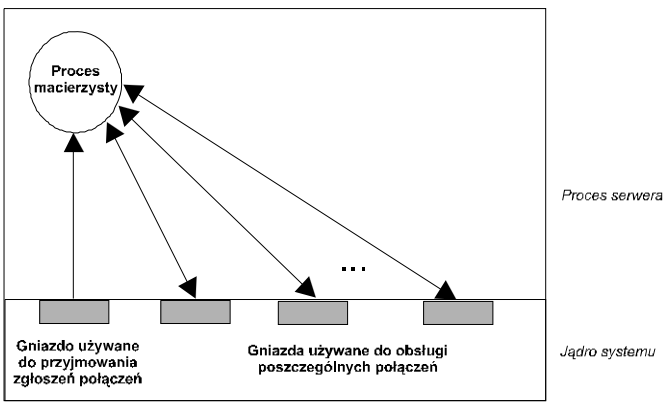
Proces macierzysty, krok 1.	Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze realizowanej przez serwer.
Proces macierzysty, krok 2.	Odbieraj kolejne zapytania od klientów posługując się funkcją `recvfrom`; dla każdego zapytania utwórz nowy proces potomny, który przygotuje odpowiedź.
Proces potomny/wątek, krok 1.	Rozpoczynając działanie, przejmij określone zapytanie od klienta i przejmij dostęp do gniazda.
Proces potomny/watek, krok 2.	Skonstruuj odpowiedź zgodnie z protokołem warstwy aplikacji i wyślij ją do klienta posługując się funkcja `sendto`.
Proces potomny/wątek, krok 3.	Zakończ się (proces potomny/wątek kończy więc działanie po obsłużeniu jednego zapytania).

- Wady: czas potrzebny do utworzenia procesu potomnego/wątku

4.6. Współbieżność pozorna - algorytm działania jednoprocesowego, współbieżnego serwera połączeniowego (mulipleksacja)

Zasada: proces serwera czeka na gotowość któregoś z gniazd - nadejście nowego żądania połączenia lub nadejście zapytania od klienta przez gniazdo już istniejące.

1. Utwórz gniazdo i zwiąż je z portem o powszechnie znanym numerze odpowiadającym usłudze realizowanej przez serwer. Dodaj gniazdo do listy gniazd, na których są wykonywane operacje we/wy.
2. Wywołaj funkcję `select`, aby czekać na zdarzenie we/wy dotyczące istniejących gniazd.
3. W razie gotowości pierwotnie utworzonego gniazda, wywołaj funkcje `accept`, w celu przyjęcia kolejnego połączenia i dodaj nowe gniazdo do listy gniazd, na których są wykonywane operacje we/wy.
4. W razie gotowości innego gniazda, wywołaj funkcję `read`, aby odebrać kolejne zapytanie nadesłane przez klienta, skonstruuj odpowiedź i wywołaj funkcję `write`, aby przesłać odpowiedź klientowi.
5. Przejdź do kroku 2.



- Zalety: jeden proces, współdzielenie pamięci, szybka obsługa nowych połączeń
- Wady: może być bardzo złożony i trudny w pielęgnacji, klient może zmonopolizować serwer

- **`select()`** - asynchroniczne wykonywanie we-wy

SKŁADNIA

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

int select(int maxfdpl, fd_set *readfds, fd_set *writefds,
                        fd_set *exceptfds, struct timeval *timeout);
```

OPIS

- Funkcja `select` umożliwia procesowi asynchroniczne wykonywanie operacji we-wy. Proces czeka na zgłoszenie gotowości przez którykolwiek z deskryptorów z podanego zbioru. Deskryptory są implementowane jako wektory bitów.
- Parametry na we:
- `maxfdpl` - maksymalna liczba deskryptorów, które będą sprawdzane
- `readfds` - deskryptory sprawdzane dla danych wejściowych
- `writefds` - deskryptory sprawdzane dla danych wyjściowych
- `excepfds` - deskryptory sprawdzane dla sytuacji wyjątkowych
- `timeout` - pozwala określić maksymalny czas oczekiwania na wystąpienie zdarzenia.
- Wartość zwracana przez funkcję `select`:

> 0 – liczba deskryptorów, które zgłosiły gotowość
0 – upłynął czas oczekiwania
-1 – błąd.

Argumenty deskryptorów po powrocie z funkcji `select` zawierają tylko, te deskryptory, które były aktywne. Uwaga: w niektórych implementacjach zmianie ulega również argument `timeout`.

- Struktura `timeval` zdefiniowana jest w pliku `time.h` następująco:

```
struct timeval
{
   long tv_sec;      /* sekundy  */
   long tv_usec;     /* mikrosekundy */
};
```

W zależności od wartości argumentu `timeout` w działaniu funkcji `select` wyróżnić można trzy przypadki:

- Oba pola struktury `timeval` są równe 0 - funkcja kończy się natychmiast po sprawdzeniu wszystkich deskryptorów; mówimy wtedy o odpytywaniu deskryptorów (ang. *polling*)
- W strukturze `timeval` określono niezerowy czas oczekiwania – funkcja czeka nie dłużej niż `timeout` na gotowość któregoś z deskryptorów;
- Argument `timeout` jest równy `NULL` – powrót z funkcja następuje dopiero wtedy, gdy jeden z deskryptorów gotowy jest do wykonania operacji we-wy.

- Zdefiniowano następujące makrodefinicje do obsługi zestawów deskryptorów:

```
/* zeruj wszystkie bity w fdset */
FD_ZERO(fd_set fdset);

/* umieść 1 w bicie dla fd w fdset */
FD_SET(int fd, fd_set *fdset);

/* zeruj bit dla fd w fdset */
FD_CLR(int fd, fd_set *fdset);

/* sprawdź bit dla fd w fdset */
FD_ISSET(int fd, fd_set *fdset);
```

- Przykład:

Istnieje stała `FD_SETSIZE`, która określa maksymalną liczbę deskryptorów obsługiwanych przez `select`:

```
int sock;       // deskryptor gniazda
fd_set rfds;    // zbiór deskryptorów do czytania

FD_ZERO(&rfds);
FD_SET(sock, &rfds);
select(FD_SETSIZE,rfds,NULL,NULL,NULL);
if (FD_ISSET(sock, &rfds)) printf("Gniazdo gotowe do czytania\n");
```

4.7. Przykładowa biblioteka podstawowych funkcji dla programów serwera

- Biblioteka zaproponowana w Comer, Stevens "Sieci komputerowe", tom 3.

- Podstawowe funkcje: utworzenie gniazda biernego

```
socket = passiveTCP(usługa, dkol); // serwer połączeniowy
socket = passiveUDP(usługa);       // serwer bezpołączeniowy
```

- Implementacja funkcji `passiveTCP` (plik `passiveTCP.c`)

```
/*
 *------------------------------------------------------------
 * passiveTCP - utwórz gniazdo bierne dla serwera
 *              używającego protokołu TCP.
 *------------------------------------------------------------
 */
#include "passivesock.h"

int passiveTCP(char *service, int qlen)
{
  return passivesock(service, "tcp", qlen);
}
```

- Implementacja funkcji `passiveUDP` (plik `passiveUDP.c`)

```
/*
 *------------------------------------------------------------
 * passiveUDP - utworz gniazdo bierne dla serwera
 *              używajacego protokolu UDP.
 *------------------------------------------------------------
 */
#include "passivesock.h"

int passiveUDP(char *service)
{
  return passivesock(service, "udp", 0);
}
```

- Implementacja funkcji tworzącej gniazdo bierne `passivesock` (plik `passivesock.c`)

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

u_short portbase=0;/* przesuniecie bazowe portu
                      dla serwera nieuprzywilejowanego */

/*
 *------------------------------------------------------
 * passivesock - ustaw gniazdo dla serwera używającego TCP
 *               lub UDP i przypisz mu adres
 *------------------------------------------------------
 */
int passivesock(char *service, char *transport, int qlen)
{
  struct servent *pse;  /* wskaznik do struktury opisu usługi */
  struct protoent *ppe;  /* wskaznik do struktury opisu protokołu  */
  struct sockaddr_in sin;  /* internetowy adres punktu końcowego */
  int s, type;  /* deskryptor, typ gniazda  */

  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;

  /* Odwzoruj nazwę usługi na numer portu */
  if(pse = getservbyname(service, transport))
    sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
  else if((sin.sin_port = htons((u_short)atoi(service))) == 0 )
    errexit("can't get \" %s \" service entry \n",service);

  /* Odwzoruj nazwę protokołu na jego numer */
  if ( (ppe = getprotobyname(transport)) == 0)
    errexit("can't get \" %s \" protocol entry \n",protocol);

 /* Wybierz typ gniazda odpowiedni dla protokołu */
  if (strcmp(transport, "udp") == 0)
    type = SOCK_DGRAM;
  else
    type = SOCK_STREAM;

  /* Utwórz gniazdo */
  s = socket(PF_INET, type, ppe->p_proto);
  if (s < 0)
    errexit("can't create socket: %s\n", sys_errlist[errno]);

  /* Przypisz adres gniazdu */
  if (bind(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't bind to %s port: %s\n", service, sys_errlist[errno]);

  if (type == SOCK_STREAM && listen(s, qlen) < 0)
    errexit("can't listen on %s port: %s\n", service, sys_errlist[errno]);

  return s;
}
```

- Przykłady serwerów TCP i UDP napisanych z użyciem proponowanej biblioteki:
  Comer, Stevens "Sieci komputerowe", tom 3
  - iteracyjny serwer usługi DAYTIME, wersja TCP, str. 163-165
  - iteracyjny serwer usługi TIME, wersja UDP, str-157-158
  - współbieżny wieloprocesowy serwer usługi ECHO , wersja TCP, str.171-174
  - współbieżny jednoprocesowy serwer usługi ECHO, wersja TCP, str. 180-183

4.8. Przykład wieloprocesowego serwera współbieżnego echo

Serwer współbieżny echa - wersja 1

Plik TCPEchoSerwer.c

```
#include "TCPEchoSerwer.h"  /* prototypy funkcji */
#include <sys/wait.h>          /* waitpid() */

unsigned int potomekLicz=0;  /* liczba potomków */

void sig_child(int sig)
{
/* wait nieblokujące */
  while (waitpid(-1,NULL,WNOHANG)>0)
   potomekLicz--;
}

int main(int argc, char *argv[]) {
  int serwGniazdo;
  int klientGniazdo;
  unsigned short echoSerwPort;
  pid_t procesID; /* numer procesu */
  unsigned int potomekLicz=0;  /* liczba potomków */

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

  signal(SIGCHLD,sig_child);

  echoSerwPort = atoi(argv[1]);
  serwGniazdo = UtworzGniazdoTCP(echoSerwPort);
  for (;;) {
    klientGniazdo=AkceptujPolaczenieTCP(serwGniazdo);
    if ((procesID=fork()) < 0) /* błąd */
      Zakoncz("fork()");
    else if (procesID==0) {   /* proces potomka */
      close(serwGniazdo);  /*zamyka gniazdo serwera*/
      PrzetwarzajKlienta(klientGniazdo);
      close(klientGniazdo); /*zamyka gniazdo połączenia */
      exit(0);  /* zakończenie procesu potomka */
    }

    /* proces serwera */
    printf("Uruchomiono proces potomny %d\n",procesID);
    close(klientGniazdo);
    potomekLicz++;
  }
}
```

Plik TCPEchoSerwer.h

```
#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>     /* close() */
void Zakoncz(char *komunikat);  /* Funkcja bledu */
void PrzetwarzajKlienta(int klientGniazdo);
int UtworzGniazdoTCP(unsigned short port);
int AkceptujPolaczenieTCP(int serwGiazdo);
```

Plik UtworzGniazdoTCP.c

```
#include <sys/socket.h> /*socket(),bind(),connect()*/
#include <arpa/inet.h>  /* sockaddr_in, inet_ntoa()*/
#include <string.h>     /* memset() */

#define MAXKOLEJKA 5
void Zakoncz(char *komunikat);

int UtworzGniazdoTCP(unsigned short port)
{
  int gniazdo;
  struct sockaddr_in echoSerwAdr;
/* Utwórz gniazdo dla przychodzących połączeń */
  if ((gniazdo =
    socket(PF_INET, SOCK_STREAM, 0)) < 0)
      Zakoncz("socket()");

/* 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(port);

/* Przypisz gniazdu lokalny adres */
  if (bind(gniazdo,(struct sockaddr *) &echoSerwAdr,
           sizeof(echoSerwAdr)) < 0)
    Zakoncz("bind()");

/* Ustaw gniazdo w trybie biernym - przyjmowania
   połączeń*/
  if (listen(gniazdo, MAXKOLEJKA) < 0)
    Zakoncz("listen()");
  return gniazdo;
}
```

Plik Zakoncz.c

```
#include <stdio.h>   /* perror() */
#include <stdlib.h>  /* exit() */
void Zakoncz(char *komunikat)
{
  perror(komunikat);
  exit(1);
}
```

Plik AkceptujPolaczenieTCP.c

```
#include <stdio.h>      /* printf() */
#include <sys/socket.h> /* accept() */
#include <arpa/inet.h>  /* sockaddr_in,inet_ntoa()*/

void Zakoncz(char *komunikat);

int AkceptujPolaczenieTCP(int serwGniazdo)
{
  int klientGniazdo;
  struct sockaddr_in echoKlientAdr;
  unsigned int klientDl;

  klientDl= sizeof(echoKlientAdr);
  if((klientGniazdo=accept(serwGniazdo,
   (struct sockaddr *)&echoKlientAdr, &klientDl)) < 0)
     Zakoncz("accept()");
  printf("Przetwarzam klienta %s\n",
      inet_ntoa(echoKlientAdr.sin_addr));
  return klientGniazdo;
}
```

Plik PrzetwarzajKlienta.c

```
#include <stdio.h>  /* printf(),fprintf(),perror() */
#include <unistd.h>     /* read(), write(), close() */
#include <sys/socket.h> /* sendto(), recvfrom() */

#define BUFWE 32

void Zakoncz(char *komunikat);

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

/* Odbierz komunikat od klienta */
  otrzTekstDl=recv(klientGniazdo,echoBufor,BUFWE,0);
  if (otrzTekstDl < 0)
    Zakoncz("recv()");

/* Odeslij komunikat do klienta i pobierz nastepny */
  while(otrzTekstDl > 0)
  {
    if (send(klientGniazdo,echoBufor,otrzTekstDl,0)
      != otrzTekstDl)
    Zakoncz("send()");
    otrzTekstDl=recv(klientGniazdo,echoBufor,BUFWE,0);
    if (otrzTekstDl < 0)
      Zakoncz("recv()");
  }
  close(klientGniazdo);
}
```

4.9. Przykład wielowątkowego serwera współbieżnego

Wielowątkowy serwer współbieżny echo, wersja 1

- Porównaj z wieloprocesowym serwerem współbieżnym echa w 5.8. W przykładzie wykorzystane są te same funkcje

```
#include "TCPEchoSerwer.h"
#include <pthread.h>       /* Wątki POSIX */

void *Wykonaj(void *arg);  /* Główna funkcja wątku */

/* Argument przesyłany do głównej funkcji wątku */
struct WatekArg {
  int klientGniazdo; /* Deskryptor gniazda klienta */
};

int main(int argc, char *argv[])
{
  int serwGniazdo;   /* Deskryptor gniazda serwera */
  int klientGniazdo; /* Deskryptor gniazda klienta */
  unsigned short echoSerwPort;    /* Port serwera */

  pthread_t watekID;    /* Identyfikator wątku dla pthread_create() */
  struct WatekArg *watekArg;   /* Wskaźnik do argumentu
                                  przekazywanego do wątku */

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

  echoSerwPort = atoi(argv[1]);
  serwGniazdo = UtworzGniazdoTCP(echoSerwPort);
  for (;;) {
    klientGniazdo = AkceptujPolaczenieTCP(serwGniazdo);
    /* przydziel pamięć dla argumentu wątku */
    if ((watekArg =
       (struct WatekArg *) malloc(sizeof(struct WatekArg)))
                == NULL)
       Zakoncz("malloc()");
    watekArg -> klientGniazdo = klientGniazdo;

    /* Utwórz wątek obsługujący klienta */
    if (pthread_create(&watekID, NULL, Wykonaj, (void *) watekArg) != 0)
      Zakoncz("pthread_create()");
    printf("watek %ld ", (long int) watekID);
    printf("proces %ld\n", (long int) getpid());
  }
}
```

```
/* Główna funkcja wątku */
void *Wykonaj(void *watekArg)
{
  int klientGniazdo;  /* Deskryptor gniazda klienta */

  /* Odłączenie wątku */
  pthread_detach(pthread_self());

  /* Wybranie deskryptora gniazda klienta
     z argumentu funkcji */
  klientGniazdo =
      ((struct WatekArg *) watekArg) -> klientGniazdo;
  free(watekArg);  /*Zwolnienie pamięci dla argumentu*/

  PrzetwarzajKlienta(klientGniazdo);

  return (NULL);
}
```

Przykład wielowątkowego serwera współbieżnego - echo, wersja 2

Porównaj z poprzednim przykładem.

```
/* Comer, Sieci komputerowe TCP/IP t.III - wersja Linux */

/* TCPmtechod.c - main, TCPechod, prstats */

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

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>

#define  QLEN      32  /* maximum connection queue length  */
#define  BUFSIZE    4096

#define  INTERVAL  5  /* secs */

struct {
  pthread_mutex_t  st_mutex;
  unsigned int  st_concount;
  unsigned int  st_contotal;
  unsigned long  st_contime;
  unsigned long  st_bytecount;
} stats;

void  prstats(void);
int  TCPechod(int fd);
int  errexit(const char *format, ...);
int  passiveTCP(const char *service, int qlen);

/*------------------------------------------------------------------------
 * main - Concurrent TCP server for ECHO service
 *------------------------------------------------------------------------
 */
int main(int argc, char *argv[])
{
  pthread_t  th;
  pthread_attr_t  ta;
  char  *service = "echo";  /* service name or port number  */
  struct  sockaddr_in fsin;  /* the address of a client  */
  unsigned int  alen;    /* length of client's address  */
  int  msock;      /* master server socket    */
  int  ssock;      /* slave server socket    */
```

```
  switch (argc) {
  case  1:
    break;
  case  2:
    service = argv[1];
    break;
  default:
    errexit("usage: TCPechod [port]\n");
  }

  msock = passiveTCP(service, QLEN);

  (void) pthread_attr_init(&ta);
  (void) pthread_attr_setdetachstate(&ta, PTHREAD_CREATE_DETACHED);
  (void) pthread_mutex_init(&stats.st_mutex, 0);

  if (pthread_create(&th, &ta, (void * (*)(void *))prstats, 0) < 0)
    errexit("pthread_create(prstats): %s\n", strerror(errno));

  while (1) {
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);
    if (ssock < 0) {
      if (errno == EINTR)
        continue;
      errexit("accept: %s\n", strerror(errno));
    }
    if (pthread_create(&th, &ta, (void * (*)(void *))TCPechod,
        (void *)ssock) < 0)
      errexit("pthread_create: %s\n", strerror(errno));
  }
}

/*------------------------------------------------------------------------
 * TCPechod - echo data until end of file
 *------------------------------------------------------------------------
 */
int TCPechod(int fd) {
  time_t  start;
  char  buf[BUFSIZ];
  int  cc;

  start = time(0);
  (void) pthread_mutex_lock(&stats.st_mutex);
  stats.st_concount++;
  (void) pthread_mutex_unlock(&stats.st_mutex);
  while (cc = read(fd, buf, sizeof buf)) {
    if (cc < 0)
      errexit("echo read: %s\n", strerror(errno));
    if (write(fd, buf, cc) < 0)
      errexit("echo write: %s\n", strerror(errno));
    (void) pthread_mutex_lock(&stats.st_mutex);
    stats.st_bytecount += cc;
    (void) pthread_mutex_unlock(&stats.st_mutex);
  }
  (void) close(fd);
  (void) pthread_mutex_lock(&stats.st_mutex);
  stats.st_contime += time(0) - start;
  stats.st_concount--;
  stats.st_contotal++;
  (void) pthread_mutex_unlock(&stats.st_mutex);
  return 0;
}
```

```
/*------------------------------------------------------------------------
 * prstats - print server statistical data
 *------------------------------------------------------------------------
 */
void prstats(void)
{
  time_t  now;

  while (1) {
    (void) sleep(INTERVAL);

    (void) pthread_mutex_lock(&stats.st_mutex);
    now = time(0);
    (void) printf("--- %s", ctime(&now));
    (void) printf("%-32s: %u\n", "Current connections", stats.st_concount);
    (void) printf("%-32s: %u\n", "Completed connections",
      stats.st_contotal);
    if (stats.st_contotal) {
      (void) printf("%-32s: %.2f (secs)\n",
        "Average complete connection time",
        (float)stats.st_contime / (float)stats.st_contotal);
      (void) printf("%-32s: %.2f\n",
        "Average byte count",
        (float)stats.st_bytecount / (float)(stats.st_contotal +
        stats.st_concount));
    }
    (void) printf("%-32s: %lu\n\n", "Total byte count",
      stats.st_bytecount);
    (void) pthread_mutex_unlock(&stats.st_mutex);

  }
}
```

4.10. Przykład jednoprocesowego serwera współbieżnego

Jednoprocesowy współbieżny serwer echo – wersja 1

```
#include "TCPEchoSerwer.h"
#include <sys/time.h>       /* dla struct timeval {} */
#include <fcntl.h>          /* dla fcntl() */

int main(int argc, char *argv[])
{
  int serwGniazdo;
  int klientGniazdo;
  int maxDeskryptor;  /* Liczba sprawdzanych deskryptorów */
  fd_set gniazdoC;    /* Zbiór przeglądanych deskryptorów */
  fd_set gniazdoS;    /* Zbiór deskryptorów dla select() */
  long timeout;       /* Timeout */
  struct timeval selTimeout;   /* Timeout dla select() */
  int wykonuj = 1;  /* 1 jesli serwer ma działać, 0 w przeciwnym wypadku */
  unsigned short portNo;    /* Port serwera */

  int x;
  if (argc < 3) {
    fprintf(stderr, "Uzycie:  %s <Timeout (sek.)> <Port>\n", argv[0]);
    exit(1);
  }
  timeout = atol(argv[1]);    /* Pierwszy arg: Timeout */
  portNo = atoi(argv[2]);     /* Drugi arg: Port */
  serwGniazdo = UtworzGniazdoTCP(portNo);

/* Przygotuj początkowy zbiór deskryptorów dla select() */
  FD_ZERO(&gniazdoC);
  FD_SET(STDIN_FILENO, &gniazdoC);
  FD_SET(serwGniazdo, &gniazdoC);
  maxDeskryptor = FD_SETSIZE;      /* z dużym nadmiarem */

  printf("Uruchamiam serwer:  nacisnij Return aby zakonczyc prace\n");

  while (wykonuj)
  {
   /* Trzeba na nowo zdefiniwoać przed każdym wywołaniem select() */
    selTimeout.tv_sec = timeout;       /* timeout (sek.) */
    selTimeout.tv_usec = 0;            /* 0 mikrosek. */
    gniazdoS=gniazdoC;

    /* Blokuj, dopóki nie jest gotowy deskryptor lub upłynął timeout */
    if (select(maxDeskryptor, &gniazdoS, NULL, NULL, &selTimeout) == 0)
      printf("Nic sie nie dzieje od %ld sek... Serwer zyje!\n", timeout);
    else
    {
      if (FD_ISSET(STDIN_FILENO, &gniazdoS)) /* sprawdź klawiature */
      {
        printf("Zamykam serwer\n");
        getchar();
        wykonuj = 0;
        continue;
      }
```

```
      /* sprawdź deskryptory gniazd */
      for (x= 0; x< FD_SETSIZE; x++)
        if ((x != STDIN_FILENO) && FD_ISSET(x, &gniazdoS))
        {
          if (x==serwGniazdo) {   /* zgłosił się nowy klient */
            klientGniazdo=accept(serwGniazdo,NULL,NULL);
            FD_SET(klientGniazdo,&gniazdoC);
            printf("Nowy klient otrzymał gniazdo %d\n", klientGniazdo);
          } else {
          printf("Czytam z gniazda %d\n",x);
          if ((PrzetwarzajKlienta(x)== 0)) {
            close (x);
            FD_CLR(x,&gniazdoC);
            printf("Klient z gniazda %d odłączony\n", klientGniazdo);
          }
        }
      } /* koniec for */
    }
  } /
  close(serwGniazdo);
  exit(0);
}
```

- Jak należy zmienić funkcję `PrzetwarzajKlienta` z poprzednich przykładów, aby klienci byli obsługiwani współbieżnie?

Jednoprocesowy serwra współbieżny echo, wersja 2

```
/* Comer, Sieci komputerowe TCP/IP t.III, s.180-182 */

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

#define QLEN 5 // maksymalna długość kolejki połączeń
#define BUFSIZE  4096 // bufor wejściowy

extern int errno;
int  errexit(const char *format, ...);
int  passiveTCP(const char *service, int qlen);
int  echo(int fd);

int main(int argc, char *argv[])
{
  char *service = "echo"; // nazwa usługi
  struct sockaddr_in fsin; // adres klienta
  int msock;   // gniazdo nasłuchujące serwera
  fd_set rfds;   // zbiór deskryptorów do czytania
  fd_set afds;   // zbiór deskryptorów aktywnych
  int alen;      // długość adresu nadawcy
  int i, nfds;  // zmienne pomocnicze

  switch (argc) {
  case 1:
    break;
  case 2:
    service = argv[1];
    break;
  default:
    errexit("użycie: TCPmechod [port]\n");
  }
// utwórz gniazdo bierne dla serwera typu TCP
// (opis funkcji: Comer t.III, str.161
// argumenty: nazwa lub numer usługi (w postaci tekstu)
//            długość kolejki zgłoszeń klientów
// wynik: deskryptor gniazda nasłuchującego
  msock = passiveTCP(service, QLEN);

// określ maksymalną liczbę przeglądanych deskryptorów
  nfds = getdtablesize();
// przypisz wszystkim deskryptorom wartość 0
  FD_ZERO(&afds);
// ustaw bit
// związany z deskryptorem gniazda nasłuchującego
  FD_SET(msock, &afds);

  while (1) {

// utwórz maskę rdfs przeglądanych deskryptorów
    memcpy(&rfds, &afds, sizeof(rfds));
```

```
// sprawdź, czy są deskryptory gotowe do przetwarzania
    if (select(nfds, &rfds, NULL, NULL, NULL) < 0)
      errexit("select: %s\n", strerror(errno));
// po wykonaniu select rfds zawiera tylko
// deskryptory gotowe do czytania

// sprawdź, czy zgłosił się nowy klient i utwórz
// dla niego gniazdo połączeniowe
    if (FD_ISSET(msock, &rfds)) {
      int ssock;
      alen = sizeof(fsin);
      ssock = accept(msock, (struct sockaddr *)&fsin,
                                              &alen);
      if (ssock < 0)
        errexit("accept: %s\n", strerror(errno));
      // dodaj gniazdo do zbioru aktywnych deskryptorów
      FD_SET(ssock, &afds);
    }

// Przeglądaj kolejne gniazda i dla każdego gotowego
// gniazda odbierz zapytanie, przetwórz je i odeślij
    for (i=0; i<nfds; ++i)
      if ( (i != msock) && FD_ISSET(i, &rfds))
        // echo() przetwarza zapytanie skierowane
        // do i-tego klienta; zwrócenie 0 oznacza,
        // że klient zakończył połączenie
        if (echo(i) == 0) {
          close(i);
          FD_CLR(i, &afds);
        }
  } // pętla while
}

/*-------------------------------------------------
      echo - odsyła echo przesłanych danych,
      zwraca liczbę przeczytanych bajtów
*-------------------------------------------------*/
int echo(int fd)
{
  char buf[BUFSIZ];
 int cc;

  cc = read(fd, buf, sizeof buf);
  if (cc < 0)
    errexit("echo read: %s\n", strerror(errno));
  if (cc && write(fd, buf, cc) < 0)
    errexit("echo write: %s\n", strerror(errno));
  return cc;
}
```

4.11. Przykład serwera wieloprotokołowego

- Wykorzystanie mechanizmów zwielokrotnionego wejścia-wyjścia w serwerze wieloprotokołowym - funkcja `select` (usługa DAYTIME, protokół TCP lub UDP w jednym programie): Comer, Stevens "Sieci komputerowe", tom 3, str. 187-190)

4.12. Przykład serwera wielousługowego

- Wykorzystanie mechanizmów zwielokrotnionego wejścia-wyjścia w serwerze wielousługowym (usługi echo, chargen, daytime, time w jednym programie): Comer, Stevens "Sieci komputerowe", tom 3, str. 200-206)

4.13. Inne właściwości serwera

Uruchomienie jako demon

- Demon jest to program, który jest wykonywany w tle i nie jest związany z żadnym terminalem. Ma działać bez żadnej interakcji z użytkownikiem i zazwyczaj jest uruchomiony tak długo, jak długo działa system. Aby zrealizować te funkcje program:
    - powinien zamknąć wszystkie odziedziczone deskryptory, aby zapobiec niepotrzebnemu przetrzymywaniu zasobów
    - musi odłączyć się od terminala sterującego, aby na jego działanie nie miały wpływu sygnały generowane przez terminal użytkownika
    - powinien zmienić katalog bieżący na taki, w którym będzie mógł działać nieograniczenie długo nie utrudniając zarządzania systemem
    - powinien zmienić wartość maski umask zgodnie ze swoimi wymaganiami
    - musi odłączyć się od grupy, aby nie otrzymywać sygnałów przeznaczonych dla tej grupy

- Przykład prostej funkcji demon:

```
#include <sys/types.h>
#include <unistd.h>
#include <syslog.h>
#include <sys/stat.h>
#include <stdlib.h>

int demon() {
  pid_t pid;
  long ldes;
  int i;

/* Krok 1 - wywołaj funkcję fork() i zakończ proces macierzysty */
  if ((pid = fork()) != 0) { exit(0); }

/* Krok 2 - utwórz nową grupę procesów i nową sesji. Uczyń proces jedynym
   członkiem tych grup i jednocześnie przywódcą tych grup. W ten sposób
   pozbędziesz się terminala sterującego */

  setsid();

/* Krok 3 - ponownie wywołaj funkcję fork() i zakończ proces macierzysty.
   Nie ma już przywódcy grupy, nie można zostać przyłączonym
   do terminala sterującego */

  if ((pid = fork()) != 0) { exit(0);  }

/* Krok 4 - wybierz jako katalog bieżący "bezpieczny" katalog */

  chdir("/");

/* Krok 5 - ustal wartość maski */

  umask(0);

/* Krok 6  - zamknij odziedziczone po procesie macierzystym deskryptory
   plików */
  ldes = sysconf(_SC_OPEN_MAX);
  for (i = 0; i < ldes; i++) {
    close(i);
  }

  return 1;
}
```

Zapisy do logów

- Serwer może posiadać własne pliki logów lub korzystać z logów systemowych.
- W systemach uniksowych tradycyjnym programem obsługi logów jest `syslog`. Przykład zapisu za pośrednictwem syslog'a:

```
int main(int argc, char **argv)
{
  demon();

/* otwórz połączenie */
  openlog("test_sewera", LOG_PID, LOG_USER);

/* wyślij komunikat */
  syslog(LOG_INFO, "%s", "Uruchomiono!");

/* zamknij połączenie */
  closelog();

  return 1;
}
```

Uzyskany wpis w `/var/log/messages`:

```
Mar 10 18:31:34 blade-runner test_serwera[10289]: Uruchomiono!
```

Zmniejszenie uprawnień

- Zasada: takie przywileje, jakie są niezbędne do wykonywania pracy.
- Jeśli serwer potrzebuje uprawnień root'a tylko do wykonania czynności początkowych po uruchomieniu, można po ich wykonaniu odebrać mu te uprawnienia.
- Do jednoczesnej zmiany wszystkich identyfikatorów użytkownika (rzeczywistego, efektywnego) służy funkcja `setuid()`. Musi być wywołana, gdy właścicielem procesu jest jeszcze root.

```
int main(int argc, char **argv)
{
  struct passwd *pws;
  const char *user = "nopriv";

  pws = getpwnam(user);
  if (pws == NULL) {
    printf("Nieznany użytkownik: %s\n", user);
    return 0;
  }

  demon();

/* Zmień ID uzytkownika na nopriv */
  setuid(pws->pw_uid);

  while (1) {
    sleep(1);
  }

  return 1;
}
```

- Można również ograniczyć dostęp do systemu plików. Za pomocą funkcji `chroot()` można określić katalog, który staje się katalogiem głównym dla danego procesu. Proces będzie miał dostęp tylko do plików znajdujących się poniżej nowego głównego katalogu. Funkcja wymaga uprawnień roota. Uwaga: funkcja `chroot()` nie zmienia bieżącego katalogu.

Wzajemne wykluczanie egzemplarzy serwera

- Zasada: nie powinno się inicjować działania więcej niż jednej kopii serwera w danym czasie; ewentualna współbieżność powinna być realizowana przez sam serwer.

- Przykład:

```
#define LOCKF /var/spool/serwer.lock
lf=open(LOCKF, O_RWDR|O_CREAT, 0640);
if (lf < 0)  /* błąd podczas otwierania pliku */
  exit(1);
if (flock(lf, LOCK_EX|LOCK_NB))
  exit(0);   /* nie udało się zablokować pliku */
```

Zarejestrowanie identyfikatora procesu serwera

- Serwer często zapamiętuje identyfikator swojego procesu w pliku o ustalonej nazwie. Dzięki temu można szybko odnaleźć ten identyfikator, bez potrzeby przeglądania listy wszystkich procesów działających w systemie.
- Takim plikiem może być na przykład plik blokady serwera.
- Przykład:

```
char pbuf[10]; /* pid w postaci napisu */
/* zamień liczbę binarną na dziesiętną */
sprintf(pbuf,"%6d\n",getpid());
/* plik blokady jest już otwarty */
write(lf,pbuf,strlen(pbuf));
```

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 130-206

W. Richard Stevens: *Unix,* programowanie *usług sieciowych, tom 1: API gniazda i XTI*: str. 129-135, 182-189, 671-707

5. Rozbudowany interfejs gniazd

5.1. Funkcje wejścia-wyjścia

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);

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

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.

```
ssize_t sendto(int  s,  const  void *msg, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
```
Wysyła komunikat do określonego hosta. Wykorzystywana w połączeniach UDP.

```
ssize_t sendmsg(int s, const struct msghdr *msg, int flags);
```
Najbardziej ogólna funkcja. Wysyła komunikat zbudowany z wielu bloków. Daje możliwość określenia opcji dla połączenia.

```
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.

```
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```
Odbiera komunikat od hosta. Adres hosta jest zwracany w argumencie `from`. Wykorzystywana w połączeniach UDP.

```
ssize_t recvmsg(int s, struct msghdr *msg, int flags);
```
Najbardziej ogólna funkcja Odbiera komunikat zbudowany z wielu bloków. Daje możliwość określenia opcji dla połączenia.

- Przykłady opcji w funkcjach wysyłających (argument flags):

MSG_OOB	Wyślij dane poza pasmowe (ang. *urgent*, *out-of-band*)
MSG_DONTWAIT	Wyślij bez blokowania. Zwraca w `errno` wartość `EWOULDBLOCK`, jeśli funkcja nie może być od razu wykonana. Dotyczy tylko jednego wywołania. Nie trzeba wtedy ustawiać gniazda w trybie nieblokującym.

Przykłady opcji w funkcjach odbierających (argument flags):

MSG_OOB	Odbierz dane poza pasmowe, jeśli nie są one umieszczone w normalnym strumieniu danych.
MSG_DONTWAIT	Odbierz bez blokowania. Zwraca w `errno` wartość `EWOULDBLOCK`, jeśli funkcja nie może być od razu wykonana. Dotyczy tylko jednego wywołania. Nie trzeba wtedy ustawiać gniazda w trybie nieblokującym.
MSG_WAITALL	Czekaj, aż odebrane zostaną wszystkie dane. Uwaga: funkcja może przekazać mniejszą od żądanej liczbę bajtów, gdy przechwycono sygnał lub połączenie zostało zakończone.
MSG_PEEK	Podgląd komunikatu – odbierz dane bez usuwania ich z bufora wejściowego.

5.2. Funkcja connect() a UDP

- Działanie funkcji connect() zależy od typu gniazda. W przypadku gniazda TCP jej zadaniem jest nawiązanie połączenia z serwerem. W przypadku gniazda UDP funkcja connect() przekazuje do jądra dane serwera. Mówimy wtedy o gnieździe UPD połączonym, co oznacza, że nie musimy określać adresu docelowego IP ani numeru portu.

5.3. Sprawdzanie zamknięcia gniazda partnera

- Wykrywanie zamkniętego gniazda podczas czytania

```
  int gniazdo, wynik;
  char bufor[BUFWE];

  gniazdo = socket(PF_INET, SOCK_STREAM, 0));

/ * Klient łączy się */

  wynik = recv(gniazdo, bufor, BUFWE, 0);
  if (wynik > 0) {
  /* Otrzymano dane, przetwarzaj je */
  }
  else if (wynik < 0 ) {
  /* wystąpił błąd, sprawdź jaki */
  }
  else if (wynik == 0) {
  /* partner zamknął połączenie */
  close(gniazdo);
}
```

- Wykrywanie zamkniętego gniazda podczas zapisu

```
  int serwGniazdo, klientGniazdo;
  int rozmiar;
  char bufor[BUFWE];
  int wynik;

  serwGniazdo = socket(PF_INET, SOCK_STREAM, 0));

/* ustawienie parametrów serwera */

  klientGniazdo =  accept(serwGniazdo, NULL, NULL);

...

  wynik=send(klientGniazdo, bufor, rozmiar, 0);
  if (wynik == 0) {
  /* dane wysłano */
  }
  else if (wynik < 0 ) {
  /* wystąpił błąd, sprawdź jaki */
    if (errno == EPIPE) {
  /* Partner zamknął gniazdo  */
  close(klientGniazdo);
  }
}
```

5.4. Dane pozapasmowe

- Dane pozapasmowe (ang. *out-of-band date*) to dane, które są przesyłane z wyższym priorytetem. Każdy rodzaj warstwy transportowej obsługuje te dane w inny sposób.
- Do obsługi danych pozapasmowych w protokole TCP wykorzystywany jest tryb pilny (ang *urgent mode*). Można w ten sposób przesłać jeden znak jako dane pilne.

- Przesyłanie danych pozapasmowych:

```
send(socket,"?",1,MSG_OOB);
```

- Odczytywanie danych pzapasmowych:
  a) dane odbierane są w specjalnym jednobajtowym buforze danych pozapasmowych (domyślne działanie gniazda); do odczytu można użyć wtedy `recv` z ustawioną flagą `MSG_OOB` :

```
recv(socket,buf,1,MSG_OOB);
```

  b) dane odbierane są przemieszane z danymi zwykłymi (gniazdo ma ustawioną opcję `SO_OOBINLINE`); należy wtedy odszukać dane pilne w zwykłych danych

- Powiadomienie o nadejściu danych pozapasmowych:
  - proces odbierający jest powiadamiany o nadejściu danych pozapasmowych za pomocą sygnału `SIGURG`; proces musi być właścicielem gniazda
  - jeśli proces korzysta z funkcji `select`, to nadejście danych pozapasmowych jest sygnalizowane pojawieniem się sytuacji wyjątkowej (trzeci zestaw badanych deskryptorów)

Przykład: wykorzystanie danych pozapasmowych do śledzenie aktywności połączenia

- Klient echa

```
#define CZEKAJ 5

int sock, odp=1;

void obsluga_syg(int sygnal)
{
  if ( sygnal == SIGURG )
  {
    char c;
    recv(sock, &c, sizeof(c), MSG_OOB);
    odp = ( c == 'T' );              /* żyję */
    fprintf(stderr,"[jest]");
  }
  else if ( sygnal == SIGALRM )
  {
    if ( odp )
    {
      send(sock, "?", 1, MSG_OOB);   /* żyjesz?? */
      alarm(CZEKAJ);
      odp = 0;
    }
    else
      fprintf(stderr, "Brak polaczenia!");
  }
}

...

  struct sigaction act;
  memset(&act, 0, sizeof(act));
  act.sa_handler = obsluga_syg;
  act.sa_flags = SA_RESTART;
  sigaction(SIGURG, &act, 0);
  sigaction(SIGALRM, &act, 0);
...
  sock = socket(PF_INET, SOCK_STREAM, 0);
// ustal właściciela gniazda
  if ( fcntl(sock, F_SETOWN, getpid()) != 0 )
    { perror("F_SETOWN"); exit(1); }
...
  if ( connect(sock, (struct sockaddr*) &adr,
       sizeof(adr)) == 0 )
  {
// ustaw okres oczekiwania
    alarm(CZEKAJ);
    do
    {
      // pobierz dane z klawiatury i prześlij do serwera
...   // czekaj na odpowiedź
    }
    while ( ... );
  }
...
```

- Serwer echa

```
int sock;
struct sigaction act;

void obsluga_syg(int sygnal)
{
  if ( sygnal == SIGURG )
  {
    char c;
    recv(sock, &c, sizeof(c), MSG_OOB);
    if ( c == '?' )                          /* Czy żyjesz? */
    send(sock, "T", 1, MSG_OOB);             /* TAK! */
  }
  else {if ( sygnal == SIGCHLD )
    wait(0);
  }
}

...

  bzero(&act, sizeof(act));
  act.sa_handler = sig_handler;
  act.sa_flags = SA_RESTART;
  sigaction(SIGURG, &act, 0);
  if ( fcntl(sock, F_SETOWN, getpid()) != 0 )
    perror("F_SETOWN");
  do
  {
    bytes = recv(sock, buffer, sizeof(buffer), 0);
    if ( bytes > 0 )
      send(sock, buffer, bytes, 0);
  }
  while ( bytes > 0 );
  close(sock);
  exit(0);
}
```

5.5 Opcje gniazd

```
int getsockopt(int socket,     // deskryptor gniazda
               int level,      // kto ma przetwarzać opcję
               int optName,    // nazwa opcji
               void *optVal,   // wartość opcji
               unsigned int *optLen);
                               // rozmiar zmiennej z opcją

int setsockopt(int socket,     // deskryptor gniazda
               int level,      // kto ma przetwarzać opcję
               int optName,    // nazwa opcji
               const void *optVal,  // wartość opcji
               unsigned int optLen);
                               // rozmiar zmiennej z opcją
```

W przypadku poprawnego wykonania funkcje zwracają 0, w przypadku błędu -1 i kod błędu w zmiennej `errno`.

- Opcje mogą dotyczyć różnych poziomów oprogramowania sieciowego (parametr `level`):
  `SOL_SOCKET` - oprogramowanie poziomu gniazd - dotyczy wszystkich gniazd
  `IPPROTO_IP` - oprogramowanie IPv4
  `IPPROTO_IPV6` - oprogramowanie IPv6
  `IPPROTO_TCP` - oprogramowanie TCP

- Opis opcji:
  ```
  man 7 socket
  man 7 tcp
  man 7 ip
  ```

- Dwa typy opcji:
  - opcje, które włączają lub wyłączają pewną właściwość
  - opcje, które pobierają lub przekazują specjalne wartości

Przykłady opcji

Nazwa		Typ	Wartość
Poziom **SOL_SOCKET**			
SO_BROADCAST	zezwolenie na wysyłanie w trybie rozgłaszania	int	0, 1
SO_KEEPALIVE	testowanie okresowe, czy połączenie żyje	int	0, 1
SO_LINGER	zwlekanie z zamykaniem, jeśli w buforze są dane do wysłania	struct linger	czas
SO_RVCBUF	rozmiar bufora odbiorczego	int	bajty
SO_SNDBUF	rozmiar bufora wysyłkowego	int	bajty
SO_RCVLOWAT	znacznik dolnego ograniczenia bufora odbiorczego	int	bajty
SO_SNDLOWAT	znacznik dolnego ograniczenia bufora wysyłkowego	int	bajty
SO_RCVTIMEO	czas oczekiwania na pobranie	struct timeval	czas
SO_SNDTIMEO	czas oczekiwania na wysłanie	struct timeval	czas
SO_REUSEADDR	zezwolenie współdzielenie przez dwa gniazda pary adres lokalny port	int	0, 1
SO_TYPE	pobranie typu gniazda (tylko getsockname))	int	liczba
SO_OOBLINE	wykorzystywane podczas przetwarzania danych poza pasmowych	int	0,1

- Gniazda połączone TCP dziedziczą niektóre opcje po gnieździe nasłuchującym. Należą do nich SO_KEEPALIVE, SO_LINGER, SO_RVCBUF, SO_SNDBUF.

- **Opcja SO_BROADCAST**

```
# Nadawca

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


int main(int argc, char *argv[])
{
  int gniazdo;
  struct sockaddr_in rozglAdr;
  char *rozglIP;
  unsigned short rozglPort;
  char *tekst;
  int rozglaszanie;
  unsigned int tekstDl;

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

  rozglIP = argv[1];          /* adres rozgloszeniowy */
  rozglPort = atoi(argv[2]);  /* port rozgloszeniowy */
  tekst = argv[3];            /* tekst rozglaszany */

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

  rozglaszanie = 1;
  if (setsockopt(gniazdo, SOL_SOCKET, SO_BROADCAST,
                 &rozglaszanie, sizeof(rozglaszanie)) < 0)
    { perror("setsockopt()"); exit(1); }

  memset(&rozglAdr, 0, sizeof(rozglAdr));
  rozglAdr.sin_family = AF_INET;
  rozglAdr.sin_addr.s_addr = inet_addr(rozglIP);
  rozglAdr.sin_port = htons(rozglPort);

  tekstDl= strlen(tekst);

for (;;)
  {
   /* Rozglaszaj co 3 sekundy */
    if (sendto(gniazdo,tekst,tekstDl,0,
               (struct sockaddr *)&rozglAdr,
                sizeof(rozglAdr)) != tekstDl) {     perror("sendto() wyslal
inna liczbe bajtow niz powinien");
      exit(1); }
      sleep(3);
  }
}
```

```
# Odbiorca
#
#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 MAXTEKST 255 /* najdluszy odbierany tekst */

int main(int argc, char *argv[])
{
  int gniazdo;
  struct sockaddr_in rozglAdr;
  unsigned int rozglPort;
  char tekst[MAXTEKST+1];
  int tekstDl;

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

  rozglPort = atoi(argv[1]);   /* port rozgloszeniowy */
  if ((gniazdo= socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
  { perror("socket()"); exit(1); }

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

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

  if ((tekstDl = recvfrom(gniazdo, tekst, MAXTEKST, 0,
                          NULL, 0)) < 0)
  { perror("recvfrom()"); exit(1); }

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

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

- **Opcja `SO_REUSEADDR`**

```
  int serwGniazdo, wynik;
  struct sockaddr_in serwAdr;
  unsigned short serwPort;
  int opcja;

  serwGniazdo = socket(PF_INET, SOCK_STREAM, 0);

  opcja=1;
  wynik = setsockopt(sock,SOL_SOCKET,SO_REUSERADDR,
                   (void *)&opcja, sizeof(opcja));

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

/* Przypisz gniazdu lokalny adres */
  wynik =bind(serwGniazdo,(struct sockaddr *) &serwAdr,
         sizeof(serwAdr));
```

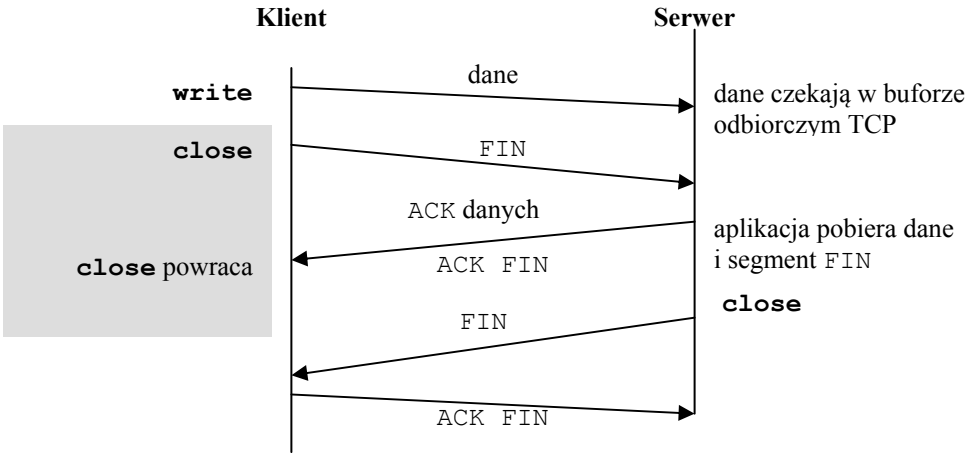
- **Opcja `SO_LINGER`**

```
struct linger {
  int l_onoff;   /* 0 - wyłączone, niezero - włączone */
  int l_linger;  /* czas zwlekania */
};
```

Jeśli:

- `l_onoff` jest równe 0 - ignorowana jest druga składowa i działanie funkcji `close` pozostaje niezmienione
- `l_onoff` jest różne od 0, `l_linger` jest równe 0 - połączenie zostanie natychmiast zerwane przez warstwę TCP
- `l_onoff` jest różne od 0, `l_linger` jest różne od 0 - proces będzie uśpiony dopóty, dopóki albo wszystkie dane będą wysłane i nadejdzie potwierdzenie od partnera, albo upłynie czas zwlekania (ang. *linger*).
- Jeśli wróci się z funkcji w wyniku upłynięcia czasu zwlekania, to zwrócony będzie kod błędu `EWOULDBLOCK` i wszystkie dane pozostawione w buforze wysyłkowym zostaną zniszczone.

- Włączone zwlekanie



Otrzymaliśmy potwierdzenie danych i przesłanego do partnera segmentu FIN. Nadal nie wiemy, czy aplikacja partnera przeczytała dane. Jak uzyskać tę informację?

- **Opcje `SO_RVCBUF` i `SO_SNDBUF`**

- Każde gniazdo ma bufor wysyłkowy i odbiorczy.
- Bufor odbiorczy wykorzystywany jest przez oprogramowanie warstwy TCP i UDP do przechowywania danych zanim przeczyta je aplikacja.
  - Wielkość bufora odbiorczego TCP jest równa rozmiarowi okna oferowanego partnerowi..
  - W przypadku UDP jeśli datagram nie mieści się w buforze odbiorczym gniazda, zostanie odrzucony.
- Każde gniazdo TCP ma bufor wysyłkowy. Do niego kopiowane są dane z bufora użytkowego aplikacji. Jeśli gniazdo jest gniazdem blokującym (ustawienie domyślne), powrót z funkcji `write` będzie oznaczał, że wszystkie dane z bufora aplikacji zostały umieszczone w tym buforze. Dane są usuwane z tego bufora dopiero po otrzymaniu potwierdzenia ACK.
- Gniazdo UDP nie ma bufora wysyłkowego. Posługuje się tylko jego rozmiarem do określenia maksymalnego rozmiaru datagramu, który można wysłać poprzez to gniazdo.

- Przykład: chcemy zwiększyć rozmiar bufora odbiorczego gniazda

```
int rvcBufferSize;
int sockOptSize;
sockOptSize=sizeof(rvcBufferSize);
if (getsockopt(sock,SOL_SOCKET,SO_RCVBUF,&rvcBufferSize, &sockOptSize) < 0)
  error("getsockopt");
printf("Poczatkowa wielkosc bufora: %d\n", rvcBufferSize);

/* Podwajamy wielkość bufora */
rvcBufferSize *=2;
if (setsockopt(sock,SOL_SOCKET,SO_RCVBUF,
               &rvcBufferSize,sizeof(rcvBufferSize)) < 0)
  error("setsockopt");
```

- **Opcje `SO_RCVLOWAT` i `SO_SNDLOWAT`**

- Funkcja `select` do stwierdzenia gotowości gniazda do czytania lub pisania wykorzystuje znaczniki dolnego ograniczenia bufora wysyłkowego i odbiorczego (ang. *low-water mark*).
- Znacznik dolnego ograniczenia bufora odbiorczego jest to niezbędna liczba bajtów w buforze odbiorczym gniazda potrzebna do tego aby `select` przekazała informację, że gniazdo nadaje się do pobierania danych.
- Znacznik dolnego ograniczenia bufora wysyłkowego jest to niezbędna wielkość dostępnej przestrzeni w buforze wysyłkowym gniazda potrzebna do tego aby `select` przekazała informację, że gniazdo nadaje się do wysyłania danych.
- W przypadku UDP znacznik ten oznacza górną granicę maksymalnego rozmiaru datagramów UDP, które można odsyłać do tego gniazda. Gniazdo to nie ma bufora wysyłkowego, ma tylko rozmiar bufora wysyłkowego.

- Przykład: chcemy otrzymać 48 bajtów, zanim nastąpi powrót z operacji czytania

```
int lowat;
int lowatSize;
sockOptSize=sizeof(rvcBufferSize);
int wynik;

lowat=48;
wynik=setsockopt(sock,SOL_SOCKET,SO_RCVLOWAT,&lowat,sizeof(rcvBufferSize));
```

- **Opcja `SO_KEEPALIVE`**

  - Opcja włącza i wyłącza sondowanie połączenie TCP (ang. keepalive probe). Sondowanie polega na wysyłaniu segmentu ACK, na który partner musi odpowiedzieć.

5.6. Gniazda nieblokujące

- Modele wejścia-wyjścia:
  - wejście-wyjście blokujące - domyślnie każde gniazdo jest blokujące, program czeka aż pojawią się dane lub będzie można je wysłać
  - wejście-wyjście nieblokujące - powróć z funkcji natychmiast, jeśli nie można zakończyć operacji wejścia-wyjścia zwróć błąd (`EWOULDBLOCK`); program musi odpytywać (ang. *polling*) taki deskryptor, czy operacja jest już gotowa do wykonania
  - wejście-wyjście zwielokrotnione - specjalna funkcja systemowa (`select`, `poll`), w której proces się blokuje, zamiast w funkcji wejścia-wyjścia; zaletą jest możliwość oczekiwania na wiele deskryptorów
  - wejście-wyjście sterowane sygnałami - jądro systemu może generować sygnał `SIGIO`, który poinformuje proces o gotowości deskryptora do rozpoczęcia operacji wejścia-wyjścia; proces nie jest blokowany
  - wejście-wyjście asynchroniczne (Posix) - proces zleca jądru rozpoczęcie operacji wejścia-wyjścia i późniejsze zawiadomienie o jej zakończeniu

- **Przykład:** Wejście-wyjście sterowane sygnałami (tradycyjna nazwa - wejście-wyjście asynchroniczne)

- Czynności związane z korzystaniem z gniazda w trybie wejścia-wyjścia sterowanego sygnałami
  1. Ustanowienie procedury obsługi sygnału `SIGIO`
  2. Ustalenie właściciela gniazda
  3. Włączenie obsługiwania gniazda w trybie wejścia-wyjścia sterowanego sygnałami

ad 1.

Protokół UDP - sygnał `SIGIO` jest generowany m.in. wtedy, kiedy

- nadejdzie datagram przeznaczony do gniazda

Protokół TCP - sygnał `SIGIO` jest generowany m.in. wtedy, kiedy

- zakończono obsługiwanie żądania połączenia z gniazdem nasłuchującym
- zainicjowano obsługiwanie żądania rozłączenia
- zakończono obsługiwanie żądania rozłączenia
- jedna ze stron połączenia została zamknięta
- do gniazda dostarczono dane
- z gniazda wysłano dane (zwolniono miejsce w buforze wysyłkowym)

Ze względu na częste występowanie sygnału SIGIO w przypadku połączenia TCP, jest on rzadko wykorzystywany do sterowania we-wy.

ad 2.

Do wykonywania operacji na deskryptorze pliku służy funkcja `fcntl()`:

```
#include <fcntl>
int fcntl(int fd, int cmd, ... /* int arg */);
```

Ustanowienie właściciela gniazda wymaga polecenia `F_SETOWN`, pozwala przypisać gniazdo procesowi o ustalonym identyfikatorze:

```
fcntl(gniazdo,F_SETOWN,getpid())
```

ad. 3.

Włączenie obsługiwania gniazda w trybie wejścia-wyjścia sterowanego sygnałami można również zrealizować za pomocą funkcji `fcntl()`:

```
int flagi;
flagi=fcntl(gniazdo,F_GETFL,0);
flagi |= FASYNC; // lub flagi |= O_ASYNC
fcntl(gniazdo,F_SETFL,flagi)
```

```
/* Serwer echa */

#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>    /* fcntl() */
#include <sys/file.h> /* O_NONBLOCK i FASYNC */
#include <signal.h>
#include <errno.h>

#define ECHOMAX 255

void ObslugaBledu(char *komunikat);
void CzasDoWykorzystania();
void ObslugaSIGIO(int typSygnalu);

int sock;

int main(int argc, char *argv[]) {
  struct sockaddr_in SerwAdr;
  unsigned short SerwPort;
  struct sigaction obsluga;

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

  SerwPort = atoi(argv[1]);
  if ((sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
    ObslugaBledu("socket()");

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

  if (bind(sock,(struct sockaddr *)&SerwAdr,
                sizeof(SerwAdr)) < 0)
    ObslugaBledu("bind()");

/* Ustanowienie procedury obsługi sygnału SIGIO */
  obsluga.sa_handler = ObslugaSIGIO;
  if (sigfillset(&obsluga.sa_mask) < 0)
    ObslugaBledu("sigfillset()");
  obsluga.sa_flags = 0;

  if (sigaction(SIGIO, &obsluga, 0) < 0)
    ObslugaBledu("sigaction() SIGIO");

/* Ustalenie właściciela gniazda */
if (fcntl(sock, F_SETOWN, getpid()) < 0)
   ObslugaBledu("Nie mozna ustawic wlasciciela procesu ");

/* Włączenie obsługiwania gniazda w trybie wejścia-wyjścia
   sterowanego sygnałami */
  if (fcntl(sock,F_SETFL,O_NONBLOCK|FASYNC)< 0)
```

```
    ObslugaBledu("Nie mozna ustawic gniazda klienta w trybie
                  O_NONBLOCK|FASYNC");

for (;;)
  CzasDoWykorzystania();
}

void CzasDoWykorzystania() {
  printf(".\n");
  sleep(3);
}

void ObslugaSIGIO(int typSygnalu){
  struct sockaddr_in KlientAdr;
  unsigned int KlientDl;
  int rozmiar;
  char bufor[ECHOMAX];

  do {
  KlientDl = sizeof(KlientAdr);
  if ((rozmiar = recvfrom(sock, bufor, ECHOMAX, 0,
         (struct sockaddr *)&KlientAdr, &KlientDl)) < 0) {
    if (errno != EWOULDBLOCK)
      ObslugaBledu("recvfrom()");
  }
  else {
    printf("Przetwarzam klienta %s\n",
            inet_ntoa(KlientAdr.sin_addr));
    if (sendto(sock, bufor, rozmiar, 0,
               (struct sockaddr *)&KlientAdr,
               sizeof(KlientAdr)) != rozmiar)
       ObslugaBledu("sendto()");
    }
  }  while (rozmiar >= 0);
}
```

Pytania:

- Dlaczego w funkcji obsługi sygnału występuje pętla?

6. Sterowanie współbieżnością

- Zalety stosowania współbieżności:
  - skrócenie obserwowanego czasu odpowiedzi na zgłoszenie, w konsekwencji zwiększenie ogólnej przepustowości serwera
  - uniknięcie ryzyka zakleszczenia
- Realizacja serwerów współbieżnych:
  - poziom współbieżności – liczba procesów serwera działających w danej chwili; pytanie: czy wprowadzić maksymalny poziom współbieżności
  - współbieżność sterowana zapotrzebowaniem (ang. *demand-driven concurrency*) – poziom współbieżności wzrasta na żądanie, odpowiada liczbie zgłoszeń, które serwer przyjął, a których obsługa nie została jeszcze zakończona.
  - alokacja wstępna procesów podporządkowanych

6.1. Modyfikacje podstawowych algorytmów serwerów

- *Serwer wyprzedzająco wieloprocesowy* (ang. *preforking*): po uruchomieniu serwera przygotowywana jest z góry pewna liczba procesów potomnych. Po ustanowieniu połączenia, klientowi przypisywany jest jeden proces z puli. Można to uzyskać, na przykład poprzez umieszczenie funkcji `accept()` w procesie potomnym. Wszystkie procesy potomne wywołują funkcję `accept()` dla tego samego gniazda nasłuchującego, jądro systemu wybiera jeden proces, któremu przekazuje połączenie.
  Zalety: koszty uruchomienia procesów potomnych ponoszone tylko raz, na początku działania programu.
  Wady: Konieczność oszacowania z góry liczby uruchamianych procesów potomnych; w niektórych systemach mogą wystąpić narzuty czasu związane z budzeniem wszystkich procesów potomnych i ponownym usypianiem, po przekazaniu jednemu z nich obsługi klienta. Pewne rozwiązanie to dynamiczna pula procesów.
- *Serwer wyprzedzająco wielowątkowy (ang. prethreading)*: zasada podobna do serwera wyprzedzająco wieloprocesowego.
- *Serwer wyprzedzająco wieloprocesowy i wyprzedzająco wielowątkowy*: tworzona jest pula procesów, w ramach każdego procesu tworzona jest pula wątków. Klient jest obsługiwany przez wątek.
- *Serwer wyprzedzająco wieloprocesowy i wyprzedzająco wielowątkowy z mulipleksacją.*

Przykłady

- Przykład A. Serwer wieloprocesowy, brak ograniczenia na procesy potomne, zróżnicowana obsługa błędów

```
void sig_child(int s)
{
  while ( waitpid(-1, 0, WNOHANG) > 0 )
    ;
}

...

int listensock;
struct sockaddr_in addr;
struct sigaction act;

...
if ( sigaction(SIGCHLD, &act, 0) != 0 )
  PANIC("sigaction");                                  // błąd krytyczny

...

if ( (listensock = socket(PF_INET, SOCK_STREAM, 0)) < 0 )
  PANIC("socket");                                     // błąd krytyczny
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_port = htons(mport);
addr.sin_addr.s_addr = INADDR_ANY;

if ( bind(listensock, &addr, sizeof(addr)) != 0 )
  PANIC("bind");                                       // błąd krytyczny
if ( listen(listensock, 5) != 0 )
  PANIC("listen");                                     // błąd krytyczny
for(;;)
{
  int sock_cli, adr_size = sizeof(addr);
  sock_cli = accept(listensock, &addr, &adr_size);
  if (sock_cli > 0 )
  {
    int pid;
    if ( (pid=fork()) == 0 )
      close(listensock);
      child(sock_cli);              // obsługa nowego klienta
    }
    else if (pid > 0)
      close(sock_cli);
    else
      perror("fork");  // wracamy do początku pętli
  }
}
```

- Przykład B. Ograniczenie liczby procesów potomnych

```
#define MAXCLIENTS 20   // maksymalna liczba klientów
int childCount=0;       // licznik procesów potomnych

void sig_child(int s)
{
  while ( waitpid(-1, 0, WNOHANG) > 0 )
    childCount --;
}

...

for(;;)
{
  int sock_cli, adr_size = sizeof(adr);
  while (childCount >= MAXCLIENTS)
    sleep(1);
  sock_cli = accept(listensock, &addr, &addr_size);
  if (sock_cli > 0 )
  {
    int pid;
    if ( (pid=fork()) == 0 )        // potomek
      close(listensock);
      child(sock_cli);               // obsługa nowego klienta
    }
    else if (pid > 0)               // proces macierzysty
    {
      childCount ++;
      close(sock_cli);
    }
    else                            // proces macierzysty
      perror("fork");  // wracamy do początku pętli
  }
}
```

- Przykład C. Tworzenie procesów z wyprzedzeniem

```
#define MAXCLIENTS 20   // maksymalna liczba klientów
int childCount =0; // licznik procesów potomnych

void sig_potomek(int s)
{
  while ( waitpid(-1, 0, WNOHANG) > 0 )
    childCount --;
}

...

for(;;)
{
  if (childCount < MAXCLIENTS)
  {
    if ( (pid=fork()) == 0 )        // potomek
      for (;;)
      {
        int sock_cli;
        sock_cli = accept(listensock, 0, 0);
        Child(sock_cli);             // obsłuż nowego klienta
      }
    else if (pid > 0)               // proces macierzysty
      childCount ++;
    else
      perror("fork");
  }
  else
    sleep(1);
}
```

- Przykład D. Zmienianie liczby procesów w zależności od zastosowania: tworzenie dodatkowych procesów ze zmienną częstotliwością

```
int okres=MAXOKRES;
time_t ostatni;

void sig_child(int signum)
{
    wait(0);               /* Czekaj na zakończenie potomka */
    time(&ostatni);        /* Aktualizuj czas */
    okres = MAXOKRES;      /* Ustaw okres domyślny */
}


...
  time(&ostatni);           /* Inicjalizacja znacznika czasu */
  for (;;)
  {
  if ( !fork() )
    child();               /* obsługa klienta (musi mieć exit()) */
    sleep(okres);
    /* Jeśli żaden proces potomny nie zakończył się,
       zwiększ częstotliwość*/
    if ( time(0) - ostatni >= okres )
      if ( okres > MINOKRES )   /* nie poniżej minimum */
        okres /= 2;        /* podwój częstotliwość */
  }
```

Założenie:
- liczba połączeń stabilna: tyle samo procesów kończy się, ile powstaje
- liczba połączeń wzrasta: zwiększenie częstotliwości tworzenia dodatkowych procesów

- Przykład E. Problemy z funkcją `select()`
  - `select` działa na zbiorach deskryptorów obejmujących wszystkie deskryptory do i włączając ten najwyższy będący przedmiotem zainteresowania
  - `select` nie wie, który z deskryptorów spowodował powrót z funkcji, każdy z deskryptorów musi być sprawdzony

Rozwiązanie - utrzymywanie małej tablicy deskryptorów: `select` w procesie potomnym

```
void child(int listensock) {
  fd_set set;
  int maxfd = listensock;
  int count=0;

  FD_ZERO(&set);
  FD_SET(listensock, &set);
  for (;;) {
    struct timeval timeout={2,0}; /* 2 sekundy */
    if ( select(maxfd+1, &set, 0, 0, &timeout) > 0 ) {

    /*--- Jeśli nowe połączenie, dodaj do listy --- */
      if ( FD_ISSET(listensock, &set) ) {
        if ( count < MAXCONNECTIONS ) {
          int sock_cli = accept(listensock, 0, 0);
          if ( maxfd < sock_cli) maxfd = sock_cli;
          FD_SET(sock_cli, &set);
          count++;
        }
      } /* koniec if - nowe połączenie */

      /*--- Jeśli żądanie klienta, przetwarzaj ---*/
      else {
        int i;
        for ( i = 0; i < maxfd+1; i++ ) {
          if ( FD_ISSET(i, &set) ) {
            char buffer[1024];
            int bytes;
            bytes = recv(i, buffer, sizeof(buffer), 0);
            if ( bytes < 0 ) { /* czy zamknięto połączenie */
              close(i);
              FD_CLR(i, &set);
              licznik--;
            }
            else  /* przetwarzaj żłdanie */
              send(i, buffer, bytes, 0);
          } /* koniec if - przetwarzania deskryptora */
        } /* koniec for - przegadanie gotowych deksryptorów */
      } /* koniec if - żądań klientów */

    } /* koniec if - select */
  } /* pętla główna */
  exit(0);
}
```

6.2. Klient współbieżny

Zalety:

- możliwość interakcji z użytkownikiem podczas przesyłania danych
- możliwość łączenia się z wieloma serwerami jednocześnie

Implementacja współbieżnego programu klienckiego

- funkcje klienta wykonywane są przez dwa lub więcej procesów (wątków)
- klient obsługuje zdarzenia na wielu wejściach i wyjściach w trybie asynchronicznym posługując się funkcją typu `select`

- **Przykład: program do pomiaru przepustowości sieci**

Comer, Stevens "Sieci komputerowe", tom 3, str. 228-234)

```
#include <sys/types.h>
#include <sys/param.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <errno.h>

int  TCPtecho(fd_set *pafds, int nfds, int ccount, int hcount);
int  reader(int fd, fd_set *pfdset);
int  writer(int fd, fd_set *pfdset);
int  errexit(const char *format, ...);
int  connectTCP(const char *host, const char *service);
long  mstime(u_long *);

#define  BUFSIZE    4096
#define  CCOUNT    64*1024
#define  USAGE  "usage: TCPtecho [ -c count ] host1 host2...\n"

char  *hname[NOFILE];
int  rc[NOFILE], wc[NOFILE];
char  buf[BUFSIZE];

int main(int argc, char *argv[]) {
  int  ccount = CCOUNT;
  int  i, hcount, maxfd, fd;
  int  one = 1;
  fd_set  afds;
  hcount = 0;
  maxfd = -1;
  for (i=1; i<argc; ++i) {
    if (strcmp(argv[i], "-c") == 0) {
      if (++i < argc &&
          (ccount = atoi(argv[i])))
        continue;
      errexit(USAGE);
    }
    /* else, a host */
    fd = connectTCP(argv[i], "echo");
/* gniazdo nieblokujące, można również użyć fcntl()*/
    if (ioctl(fd, FIONBIO, (char *)&one))
      errexit("can't mark socket nonblocking: %s\n",
        strerror(errno));
    if (fd > maxfd)
      maxfd = fd;
    hname[fd] = argv[i];
    ++hcount;
    FD_SET(fd, &afds);
  }
  TCPtecho(&afds, maxfd+1, ccount, hcount);
  exit(0);
}
```

```
int TCPtecho(fd_set *pafds, int nfds, int ccount, int hcount)
{
  fd_set  rfds, wfds;
  fd_set  rcfds, wcfds;
  int  fd, i;

  for (i=0; i<BUFSIZE; ++i)
    buf[i] = 'D';
  memcpy(&rcfds, pafds, sizeof(rcfds));
  memcpy(&wcfds, pafds, sizeof(wcfds));
  for (fd=0; fd<nfds; ++fd)
    rc[fd] = wc[fd] = ccount;

  (void) mstime((u_long *)0);

  while (hcount) {
    memcpy(&rfds, &rcfds, sizeof(rfds));
    memcpy(&wfds, &wcfds, sizeof(wfds));

    if (select(nfds, &rfds, &wfds, (fd_set *)0,(struct timeval *)0) < 0)
      errexit("select failed: %s\n", strerror(errno));
    for (fd=0; fd<nfds; ++fd) {
      if (FD_ISSET(fd, &rfds))
        if (reader(fd, &rcfds) == 0)
          hcount--;
      if (FD_ISSET(fd, &wfds))
        writer(fd, &wcfds);
    }
  }
}

int reader(int fd, fd_set *pfdset) {
  u_long  now;
  int  cc;

  cc = read(fd, buf, sizeof(buf));
  if (cc < 0)
    errexit("read: %s\n", strerror(errno));
  if (cc == 0)
    errexit("read: premature end of file\n");
  rc[fd] -= cc;
  if (rc[fd])
    return 1;
  (void) mstime(&now);
  printf("%s: %d ms\n", hname[fd], now);
  (void) close(fd);
  FD_CLR(fd, pfdset);
  return 0;
}

int writer(int fd, fd_set *pfdset) {
  int  cc;

  cc = write(fd, buf,MIN(sizeof(buf), wc[fd]));
  if (cc < 0)
    errexit("read: %s\n", strerror(errno));
  wc[fd] -= cc;
  if (wc[fd] == 0) {
    (void) shutdown(fd, 1);
    FD_CLR(fd, pfdset);
  }
}
```

```
long mstime(u_long *pms) {
  static struct timeval  epoch;
  struct timeval    now;

  if (gettimeofday(&now, (struct timezone *)0))
    errexit("gettimeofday: %s\n", strerror(errno));
  if (!pms) {
    epoch = now;
    return 0;
  }
  *pms = (now.tv_sec - epoch.tv_sec) * 1000;
  *pms +=(now.tv_usec-epoch.tv_usec+500)/1000;
  return *pms;
}
```

- Pytanie: dlaczego gniazdo jest ustawiane w trybie nieblokującym?

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 208-221, 223-235

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 806-845

Uzupełnienia

http://www.kegel.com/c10k.html
http://www.atnf.csiro.au/people/rgooch/linux/docs/io-events.html
http://bulk.fefe.de/scalable-networking.pdf

7. Krótkie wprowadzenie do korzystania z OpenSSL

Literatura:

http://www.openssl.org

E. Rescola, "An introduction to OpenSSL Programming (PartI)"
(http://www.linuxjournal.com/article/4822)
"An introduction to OpenSSL Programming (PartII)"
(http://www.linuxjournal.com/article/5487)
http://www.rtfm.com/openssl-examples/

Biblioteka OpenSSL

Pliki nagłówkowe

```
#include <openssl/ssl.h>
#include <openssl/error.h>
```

Kompilacja programu

```
gcc -o serwer serwer.c -lcrypto -lssl
gcc -o klient klient.c -lcrypto -lssl
```

Inicjalizacja biblioteki

```
#include <openssl/ssl.h>
#include <openssl/err.h>

SSL_library_init();
SSL_load_error_strings();
OpenSSL_add_all_algoritms();
```

Struktury wykorzystywane przez bibliotekę

SSL_METHOD

Pozwala określić metodę kryptograficzną wykorzystywana do komunikacji: SSLv1, SSLv2, ... , TLSv1

```
SSL_METHOD *my_ssl_method;
my_ssl_method = TLSv1_method();
```

SSL_CTX

Określa kontekst komunikacji serwera, czyli jakiej konfiguracji oczekujemy. Wykorzystywana jest do tworzenia obiektu reprezentującego każde połączenie.

```
SSL_CTX *my_ssl_ctx;

// utwórz nowy kontekst
my_ssl_ctx = SSL_CTX_new(my_ssl_method);

// lokalizacja plików z kluczem prywatnym i certyfikatów
SSL_CTX_use_certificate_file(my_ssl_ctx,"server.pem",SSL_FILETYPE_PEM);
SSL_CTX_usePrivateKey_file(my_ssl_ctx,"server.pem",SSL_FILETYPE_PEM);

//Weryfikacja klucza prywatnego
if (SSL(SSL_CTX_check_private_key(my_ssl_ctx)
// klucz działa
else
// niepoprawny klucz
```

BIO

Interfejs pozwalający czyta/zapisywać dane z różnych źródeł we/wy (gniazd, terminala, buforów pamięci, itd.)

SSL

Struktura zarządzająca danymi niezbędnymi do korzystania z bezpiecznego połączenia. Jest tworzona dla każdego połączenia.

```
SSL *my_ssl;
// połączenie struktury z kontekstem
my_ssl = SSL_new(my_ssl_ctx);

// połączenie struktury z gniazdem opisanym za pomocą deskryptora fd
SSL_set_fd(my_ssl, fd)

//server
if (SSL_accept(my_ssl) <= 0)
  // wystapiły błędy
else
  // nawiązano bezpieczne połączenie

//klient
if (SSL_connect(my_ssl) <= 0)
  // wystapiły błędy
else
  // nawiązano bezpieczne połączenie

// uzyskanie informacji o połączeniu
printf("[%s,%s]\n", SSL_get_version(my_ssl), SSL_get_cipher(my_ssl));
```

A. Przykład klienta

Działamy w oparciu o powiązanie z deskryptorami plików.

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>
/*
    SSL includes
 */
#include <openssl/ssl.h>
#include <openssl/err.h>

int main(int argc, char *argv[]) {
    SSL_METHOD *my_ssl_method;
    SSL_CTX *my_ssl_ctx;
    SSL *my_ssl;

    int my_fd;
    struct sockaddr_in server;
    int error = 0, read_in = 0;
    char buffer[512];

    memset(buffer,'\0',sizeof(buffer));

    OpenSSL_add_all_algorithms();
    SSL_library_init();
    SSL_load_error_strings();

    my_ssl_method = TLSv1_client_method();

    if((my_ssl_ctx = SSL_CTX_new(my_ssl_method)) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    if((my_ssl = SSL_new(my_ssl_ctx)) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    my_fd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&server,sizeof(server));
    server.sin_family = AF_INET;
    server.sin_port = htons(5353);
    inet_aton("127.0.0.1",&server.sin_addr);
    bind(my_fd, (struct sockaddr *)&server, sizeof(server));
    connect(my_fd,(struct sockaddr *)&server, sizeof(server));


    SSL_set_fd(my_ssl,my_fd);
```

```
    if(SSL_connect(my_ssl) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    printf("Connection made with [version,cipher]:
            [%s,%s]\n",SSL_get_version(my_ssl),SSL_get_cipher(my_ssl));

    for( read_in = 0; read_in < sizeof(buffer); read_in += error ) {
        error = SSL_read(my_ssl,buffer+read_in,sizeof(buffer) - read_in);
        if(error <= 0)
            break;
    }

    SSL_shutdown(my_ssl);
    SSL_free(my_ssl);
    SSL_CTX_free(my_ssl_ctx);
    close(my_fd);

    printf("%s",buffer);

    return 0;
}
```

A. Przykład serwera

Działamy w oparciu o powiązanie z deskryptorami plików.

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

/* SSL */
#include <openssl/ssl.h>
#include <openssl/err.h>

int main(int argc, char *argv[]) {
    SSL_METHOD *my_ssl_method;
    SSL_CTX *my_ssl_ctx;
    SSL *my_ssl;
    int my_fd,client_fd;
    struct sockaddr_in server, client;
    int client_size;
    int error = 0, wrote = 0;
    char buffer[] = "Hello there! Welcome to the SSL test server.\n\n";

    OpenSSL_add_all_algorithms();
    SSL_library_init();
    SSL_load_error_strings();

    my_ssl_method = TLSv1_server_method();

    if((my_ssl_ctx = SSL_CTX_new(my_ssl_method)) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    SSL_CTX_use_certificate_file(my_ssl_ctx,"server.pem",SSL_FILETYPE_PEM);
    SSL_CTX_use_PrivateKey_file(my_ssl_ctx,"server.pem",SSL_FILETYPE_PEM);

    if(!SSL_CTX_check_private_key(my_ssl_ctx)) {
        fprintf(stderr,"Private key does not match certificate\n");
        exit(-1);
    }

    my_fd = socket(PF_INET, SOCK_STREAM, 0);
    server.sin_family = AF_INET;
    server.sin_port = htons(5353);
    server.sin_addr.s_addr = INADDR_ANY;
    bind(my_fd, (struct sockaddr *)&server, sizeof(server));
    listen(my_fd, 5);
```

```
    for(;;) {
        client_size = sizeof(client);
        bzero(&client,sizeof(client));
        client_fd = accept(my_fd, (struct sockaddr *)&client,
                    (socklen_t *)&client_size);
        if((my_ssl = SSL_new(my_ssl_ctx)) == NULL) {
            ERR_print_errors_fp(stderr);
            exit(-1);
        }

        SSL_set_fd(my_ssl,client_fd);

        if(SSL_accept(my_ssl) <= 0) {
            ERR_print_errors_fp(stderr);
            exit(-1);
        }

        printf("Connection made with [version,cipher]:
                [%s,%s]\n",SSL_get_version(my_ssl),SSL_get_cipher(my_ssl));

        for(wrote = 0; wrote < strlen(buffer); wrote += error) {
            error = SSL_write(my_ssl,buffer+wrote,strlen(buffer)-wrote);

            if(error <= 0)
                break;
        }

        SSL_shutdown(my_ssl);

        SSL_free(my_ssl);
        close(client_fd);
    }

    SSL_CTX_free(my_ssl_ctx);

    return 0;
}
```

B. Przykład klienta

Wykorzystywana jest struktura BIO

```
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
/* SSL  */
#include <openssl/ssl.h>
#include <openssl/err.h>

int main(int argc, char *argv[]) {
    SSL_METHOD *my_ssl_method;
    SSL_CTX *my_ssl_ctx;
    SSL *my_ssl;
    BIO *my_bio;
    int error = 0, read_in = 0;
    char buffer[512];

    memset(buffer,'\0',sizeof(buffer));

    OpenSSL_add_all_algorithms();
    SSL_library_init();
    SSL_load_error_strings();

    my_ssl_method = TLSv1_client_method();

    if((my_ssl_ctx = SSL_CTX_new(my_ssl_method)) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    if((my_ssl = SSL_new(my_ssl_ctx)) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    if((my_bio = BIO_new_connect("127.0.0.1:5353")) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    if(BIO_do_connect(my_bio) <=0) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    SSL_set_bio(my_ssl,my_bio,my_bio);

    if(SSL_connect(my_ssl) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    printf("Connection made with [version,cipher]:
            [%s,%s]\n",SSL_get_version(my_ssl),SSL_get_cipher(my_ssl));
```

```
    for( read_in = 0; read_in < sizeof(buffer); read_in += error ) {
        error = SSL_read(my_ssl,buffer+read_in,sizeof(buffer) - read_in);
        if(error <= 0)
            break;
    }

    SSL_shutdown(my_ssl);
    SSL_free(my_ssl);
    SSL_CTX_free(my_ssl_ctx);

    printf("%s",buffer);

    return 0;

    report_error("Report error (not quit) test\n",__FILE__,__LINE__,0);
    report_error_q("Report error (quit) test\n",__FILE__,__LINE__,0);
    return 0;
}
```

B. Przykład serwera

Wykorzystywana jest struktura BIO

```
/*
    Standard includes
 */
#include <stdio.h>
#include <stdlib.h>
#include <strings.h>
#include <string.h>
/*
    SSL includes
 */
#include <openssl/ssl.h>
#include <openssl/err.h>

int main(int argc, char *argv[]) {
    SSL_METHOD *my_ssl_method;
    SSL_CTX *my_ssl_ctx;
    SSL *my_ssl;
    BIO *server_bio,*client_bio;
    int error = 0, wrote = 0;
    char buffer[] = "Hello there! Welcome to the SSL test server.\n\n";

    OpenSSL_add_all_algorithms();
    SSL_library_init();
    SSL_load_error_strings();

    my_ssl_method = TLSv1_server_method();

    if((my_ssl_ctx = SSL_CTX_new(my_ssl_method)) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    SSL_CTX_use_certificate_file(my_ssl_ctx,"server.pem",SSL_FILETYPE_PEM);
    SSL_CTX_use_PrivateKey_file(my_ssl_ctx,"server.pem",SSL_FILETYPE_PEM);

    if(!SSL_CTX_check_private_key(my_ssl_ctx)) {
        fprintf(stderr,"Private key does not match certificate\n");
        exit(-1);
    }

    if((server_bio = BIO_new_accept("5353")) == NULL) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }

    if(BIO_do_accept(server_bio) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(-1);
    }
```

```
    for(;;) {
        if(BIO_do_accept(server_bio) <= 0) {
            ERR_print_errors_fp(stderr);
            exit(-1);
        }

        client_bio = BIO_pop(server_bio);

        if((my_ssl = SSL_new(my_ssl_ctx)) == NULL) {
            ERR_print_errors_fp(stderr);
            exit(-1);
        }

        SSL_set_bio(my_ssl,client_bio,client_bio);

        if(SSL_accept(my_ssl) <= 0) {
            ERR_print_errors_fp(stderr);
            exit(-1);
        }

        printf("Connection made with [version,cipher]:
                [%s,%s]\n",SSL_get_version(my_ssl),SSL_get_cipher(my_ssl));

        for(wrote = 0; wrote < strlen(buffer); wrote += error) {
            error = SSL_write(my_ssl,buffer+wrote,strlen(buffer)-wrote);

            if(error <= 0)
                break;
        }

        SSL_shutdown(my_ssl);
        SSL_free(my_ssl);
    }

    SSL_CTX_free(my_ssl_ctx);
    SSL_BIO_free(server_bio);

    return 0;
}
```

Biblioteka OpenSSL

- Implementacja protokołów SSL/TSL
- Procedury kryptograficzne
- Generatory liczb losowych
- Wsparcie działań na wielkich liczbach

Inicjalizacja

```
OpenSSL_add_all_algorithms();
SSL_load_error_strings();
```

Struktury wykorzystywane przez bibliotekę

- SSL_METHOD – SSLv1, SSLv2, ..., TLSv1

```
SSL_METHOD *my_ssl_method;
my_ssl_method = TLSv1_method();
```

- SSL_CTX

```
SSL_CTX *my_ssl_ctx;
my_ssl_ctx = SSL_CTX_new(my_ssl_method)
```

```
if((my_ssl_ctx = SSL_CTX_new(my_ssl_method)) == NULL) {
     ERR_print_errors_fp(stderr);
     exit(1);
}
```

- SSL

```
SSL *my_ssl;
my_ssl = SSL_new(my_ssl_ctx
```

```
if((my_ssl = SSL_new(my_ssl_ctx)) == NULL) {
     ERR_print_errors_fp(stderr);
     exit(-1);
}
```

- BIO