4. Algorytmy klienta

4.1. Algorytm działania programu klienckiego typu połączeniowego (TCP)

1. Określ adres IP i numer portu dla serwera, z którym należy nawiązać komunikację.
2. Uzyskaj przydział gniazda (wywołanie funkcji `socket`).
3. Ustal lokalny numer portu i adres IP.
4. Uzyskaj połączenie gniazda z serwerem (wywołanie funkcji `connect`).
5. Komunikuj się z serwerem za pomocą protokołu warstwy użytkowej (wymaga to zazwyczaj wysyłania zapytań na przykład przy pomocy funkcji systemowej `write` i odbieranie odpowiedzi na przykład za pomocą funkcji `read`).
6. Zamknij połączenie (wywołanie funkcji `close`).

Ad 3. Programy klienckie używające protokołu TCP zazwyczaj nie określają adresu lokalnego punktu końcowego ani numeru portu, lecz pozostawiają oprogramowaniu TCP/IP wybór zarówno właściwego adresu IP, jak i wolnego numeru portu.

Ad 4. Funkcja `connect`:

- sprawdza poprawność odwołania do gniazda
- wypełnia w strukturze opisującej gniazdo pole adresu odległego hosta
- wybiera odpowiedni lokalny adres IP i numer portu (o ile nie zostały przypisane) i umieszcza w strukturze opisującej gniazdo
- inicjuje nawiązanie połączenia TCP; z funkcji powraca się wtedy, kiedy będzie ustanowione połączenie albo pojawi się błąd.

Funkcja `connect()` nie może być ponawiana bez uprzedniego otworzenia nowego gniazda.

Ad 5. Przebieg współpracy z serwerem określa protokół komunikacji. TCP jest protokołem strumieniowym, odbiór danych musi być wykonywany iteracyjnie – dane mogą przychodzić podzielone na porcje. Należy zapewnić odbiór całej porcji danych przeznaczonej do przetwarzania.

Ad 6. Do zamykania połączenia służy funkcja `close`. Czasem jednak klient wie, że chce zakończyć połączenie, ale nie wie czy otrzymał już wszystkie dane wysyłane przez serwer. Serwer może przesyłać dowolnie dużo danych w odpowiedzi na zapytanie klienta. Wtedy klient może wykonać operację częściowego zamknięcia połączenia po wysłaniu ostatniego zapytania – służy do tego funkcja `shutdown`. Klient przekazuje do oprogramowania warstwy niższej informację, że nie będzie już wysyłał danych, ale gniazdo nie zostaje jeszcze zwolnione. W rezultacie serwer otrzymuje sygnał końca pliku. Może zatem zamknąć to połączenie po wysłaniu ostatniej odpowiedzi.

- Przykład komunikacji z serwerem przez połączenie TCP

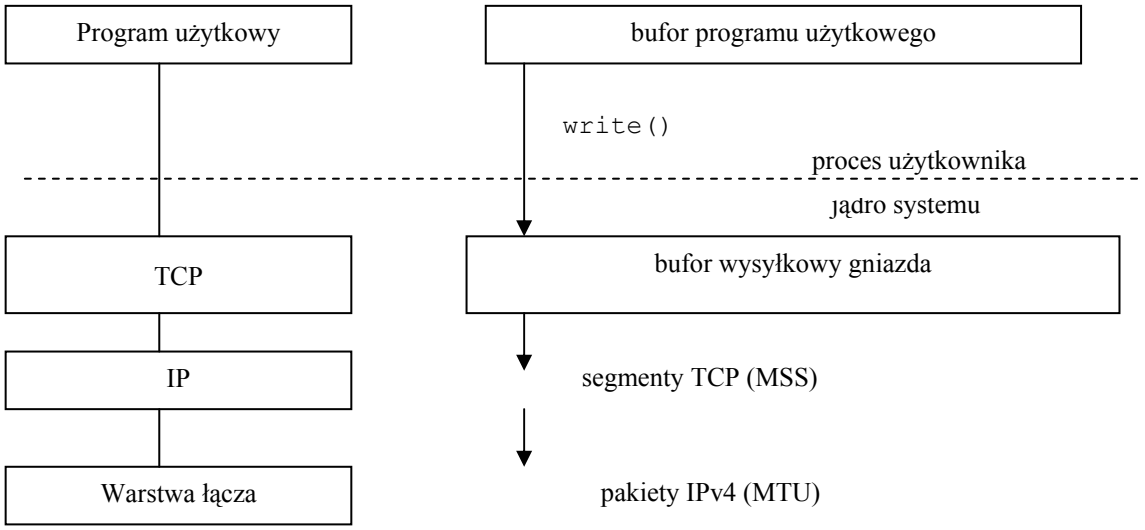
```
#define BDL 120  /* dlugosc bufora */
char pytanie="Czy jestes gotowy?";  /* tekst do przesłania */
char buf[BDL];  /* bufor na odpowiedź */
char bptr;      /* wskaźnik do bufora */
int n;          /* liczba przeczytanych bajtów */
int bufdl;      /* ilość wolnego miejsca w buforze */

bptr=buf;
bufdl=BDL;

/* wyślij zapytanie */
write(gniazdo,pytanie,strlen(pytanie));

/* czytaj 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 */
}
```

- Wysyłanie danych do gniazda TCP



- Każde gniazdo TCP ma określony bufor wysyłkowy dla danych, które ma wysyłać.
- Funkcja `write` powoduje, że jądro kopiuje dane z bufora programu użytkownika do bufora wysyłkowego gniazda. Jeśli w buforze wysyłkowym nie ma miejsca, proces się usypia. Powrót z `write` będzie wykonany dopiero po skopiowaniu ostatniego bajtu do bufora wysyłkowego.
- Oprogramowanie TCP pobiera dane z bufora wysyłkowego i przesyła je do warstwy IP porcjami o rozmiarze MSS – *maksymalny rozmiar segmentu* (ang. *Maximum Segment Size*), uzgodnionym z partnerem., dołączając nagłówek TCP. Zwykłe MSS ≤ MTU –40 .
- Oprogramowanie IP dołącza swój nagłówek i przesyła do warstwy łącza danych.

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

- Funkcja `write` służy do przesłania danych do komputera odległego. Funkcja zwraca liczbę poprawnie przesłanych bajtów lub –1 w wypadku błędu. Kod błędu jest umieszczany w `errno`. Składnia jest następująca:

  ```
  int write(int socket, char* buf, int buflen)
  ```

  Każde gniazdo TCP ma określony bufor wysyłkowy dla danych, które ma wysyłać. Funkcja `write` powoduje, że jądro kopiuje dane z bufora programu użytkownika do bufora wysyłkowego gniazda. Jeśli w buforze wysyłkowym nie ma miejsca, proces się usypia. Powrót z `write` będzie wykonany dopiero po skopiowaniu ostatniego bajtu do bufora wysyłkowego.

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

- Funkcja `read` służy do odebrania danych wejściowych z gniazda. Funkcja zwraca 0 w razie końca pliku, wartość określającą liczbę przeczytanych bajtów lub –1 w wypadku błędu. Kod błędu jest umieszczany w `errno`. Składnia jest następująca:

  ```
  int read(int socket, char* buf, int buflen)
  ```

- **`send()`** - wysyłanie danych do odległego komputera (dla gniazda połączonego)

  SKŁADNIA

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

  OPIS
  - Funkcja send służy do przesyłania danych do odległego komputera. Gniazdo musi być w stanie `connected`.

  ARGUMENTY
  - `sockfd` - deskryptor gniazda,
  - `buff` - adres bufora zawierającego dane do wysłania,
  - `nbytes` - liczba bajtów danych w buforze,
  - `flags`- opcje sterowania transmisja lub opcje diagnostyczne (lub 0)

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

- **`recv()`** - odbieranie danych z gniazda (dla gniazda połączonego)

  SKŁADNIA

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

  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,

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

Budowanie komunikatów

- Protokół TCP jest protokołem strumieniowym. Aplikacja często działa na komunikatach o określonej strukturze. Odbiorca musi wiedzieć w jaki sposób wyróżnić początek i koniec komunikatu, ewentualnie początek i koniec pól składowych komunikatu.
- Można wyróżnić przypadki:
  - znana jest długość odbieranego komunikatu
  - znany jest znak kończący dane zawarte w pojedynczym komunikacie, np. znak nowej linii
  - każdy komunikat może być zmiennej długości i nie ma specjalnego znaku kończącego dane, informacja o długości komunikatu jest przesyłana w ustalony sposób
- Przykład funkcji czytającej dla przypadku, w którym znana jest długość komunikatu, wykorzystywana jest funkcja `read()`:

```
/* readn - czytaj dokładnie n bajtów */
int readn(int s, char *bufor, size_t dl) {
  int licznik; /* ile bajtów jeszcze do przeczytania */
  int rl;      /* ile bajtów otrzymano */
  licznik = dl;
  while ( licznik > 0 )   {
    rl = recv( s, bufor, licznik, 0 ); /* lub read() */
    if ( rl < 0 )  {
      if ( errno == EINTR )  /* przerwano? */
        continue;
      return -1;        /* zwróć błąd */
    }
    if ( rl == 0 )        /* koniec? */
      return dl - licznik;
    bufor += rl;
    licznik -= rl;
  }
  return dl;
}
```

- W przypadku użycia funkcji `recv()` można wykorzystać opcję (`MSG_WAITALL`), która powoduje, że powrót z funkcji następuje dopiero po wypełnieniu całego bufora.

4.2. Algorytm działania programu klienckiego typu bezpołączeniowego (UDP)

1. Określ adres IP i numer portu dla serwera, z którym należy nawiązać komunikację.
2. Uzyskaj przydział gniazda.
3. Ustal lokalny numer portu i adres IP (zdaj się na oprogramowanie warstwy UDP i IP).
4. Podaj adres serwera, do którego mają być wysyłane komunikaty.
5. Realizuj własne zadania komunikując się z serwerem za pomocą protokołu poziomu użytkowego (wyślij zapytanie – czekaj na odpowiedź).
6. Zamknij połączenie.

Ad 4. Są dwa tryby działania programu klienckiego UDP:

- połączeniowy – do gniazda przypisywany jest adres IP i numer portu serwera (funkcja `connect`), ale nie jest inicjowana żadne połączenie z serwerem.
- bezpołączeniowy – gniazdo nie jest związane z żadnym punktem końcowym, w każdym datagramie trzeba podać adres docelowy.

Ad 5. Po wykonaniu `connect` klient może wywołać funkcję `read`, żeby odczytać datagram i `write`, żeby go wysłać. Do odebrania lub wysłania całego komunikatu wystarczy pojedyncze wywołanie takiej funkcji.
W wypadku gniazda bezpołączeniowego trzeba używać funkcji `sendto` i `recvfrom`, które pozwalają podać adres punktu końcowego.

Ad 6. Funkcja `close` zamyka gniazdo i zwalnia związane z nim zasoby systemowe. Po jej wywołaniu oprogramowanie UDP będzie odrzucać wszystkie komunikaty wysyłane do portu związanego z danym gniazdem, jednak nie zawiadomi o tym serwera. Zamknięcie gniazda może być również zrealizowane przy pomocy funkcji `shutdown`. Komunikacja jest zamykana w jednym kierunku bez powiadamianie o tym serwera.

- Protokół UDP jest protokołem zawodnym. Trzeba wbudować mechanizmy zabezpieczające (szczególnie w sieciach złożonych), w szczególności:
  - odmierzające limit czasu na transmisję
  - inicjujące w razie potrzeby ponowną transmisję
  - ustalające kolejność odbieranych datagramów
  - potwierdzające odbiór

- **`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.

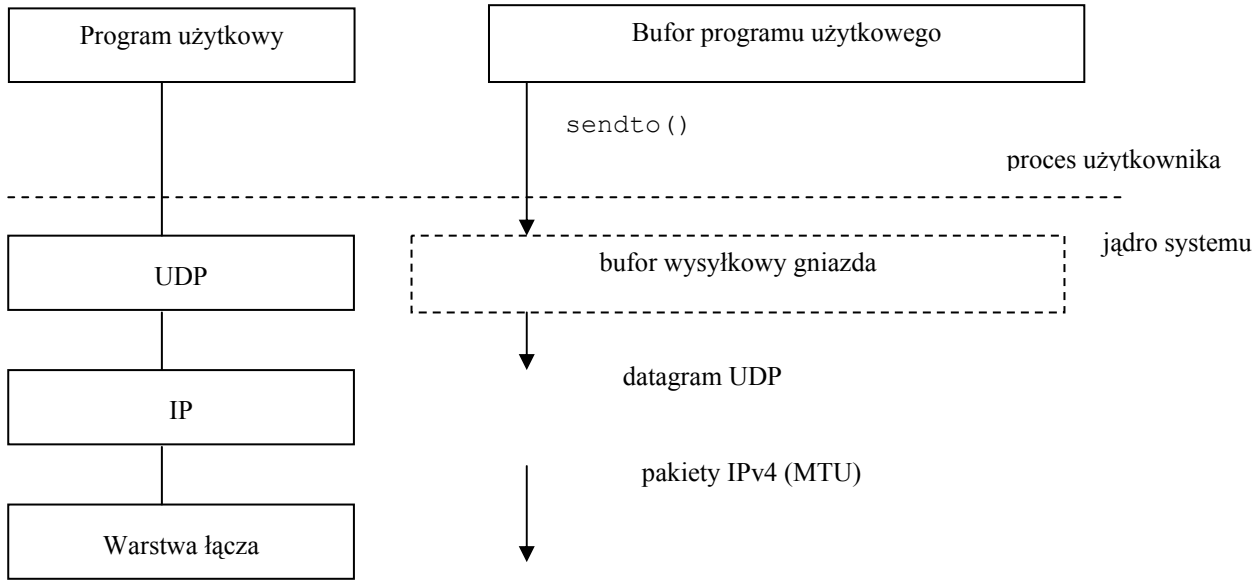
- Przykład komunikacji z serwerem przez połączenie UDP

```
#define BDL 120  /* dlugosc bufora */
char pytanie="Czy jestes gotowy?";  /* tekst
                   do przesłania */
char buf[BDL];  /* bufor na odpowiedź */
char bptr;      /* wskaźnik do bufora */
int n;          /* liczba przeczytanych bajtów */
int bufdl;      /* ilość wolnego miejsca w buforze */

bptr=buf;
bufdl=BDL;

write(s,pytanie,strlen(pytanie));
n=read(gniazdo, bptr, bufdl);
```

- Wysyłanie danych do gniazda UDP



- Każde gniazdo UDP ma określony górny rozmiar datagramu UDP, który można przesyłać do gniazda (pełni funkcję podobną do bufora wysyłkowego TCP, ale UDP nie przechowuje kopii danych i nie ma potrzeby posiadania bufora wysyłkowego).
- Warstwa UDP umieszcza swój nagłówek i przesyła do warstwy UP.
- Oprogramowanie IP dołącza swój nagłówek i przesyła do warstwy łącza danych.

Przykład 1: Klient echa - czas oczekiwania na odpowiedź w kliencie zarządzany za pomocą sygnału `SIGALRM`

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>

#define ECHOMAX  255
#define TIMEOUT  2
#define MAXPOWT  5

int powtorz=0;   // liczba powtórzeń przesyłania

void ObslugaBledu(char *komBledu);
void PrzechwycAlarm(int ignoruj); // SIGALRM

int main(int argc, char *argv[]) {
  int gniazdo;
  struct sockaddr_in SerwAdr;
  struct sockaddr_in Nadawca;
  unsigned short SerwPort;
  unsigned int rozmiarNad;
  struct sigaction obslugaSyg;
  char *serwIP;
  char *napis;
  char bufor[ECHOMAX+1];
  int napisDl;
  int odpDl;

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

  serwIP = argv[1];
  napis = argv[2];

  if ((napisDl = strlen(napis)) > ECHOMAX)
    ObslugaBledu("Tekst za dlugi");

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

  if ((gniazdo = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP)) < 0)
   ObslugaBledu("socket()");

  obslugaSyg.sa_handler = PrzechwycAlarm;
  if (sigfillset(&obslugaSyg.sa_mask) < 0)
    ObslugaBledu("sigfillset()");
  obslugaSyg.sa_flags = 0;
  if (sigaction(SIGALRM, &obslugaSyg, 0) < 0)
      ObslugaBledu("sigaction() - SIGALRM");
```

```
  memset(&SerwAdr,0,sizeof(SerwAdr));
  SerwAdr.sin_family = AF_INET;
  SerwAdr.sin_addr.s_addr=inet_addr(serwIP);
  SerwAdr.sin_port = htons(SerwPort);

  if (sendto(gniazdo,napis,napisDl,0, (struct sockaddr *) &SerwAdr,
        sizeof(SerwAdr)) != napisDl)
    ObslugaBledu("sendto() nie wyslal calego tekstu");

  rozmiarNad = sizeof(Nadawca);

/* Ustaw budzik */
  alarm(TIMEOUT);

  while ((odpDl =
         recvfrom(gniazdo, bufor, ECHOMAX, 0,
           (struct sockaddr *) &Nadawca, &rozmiarNad)) < 0)
    if (errno == EINTR) /* Upłynął czas !!! */
    {
      if (liczbaPowtorzen < MAXPOWT) {
        printf("czas uplynal, jeszcze %d proby.\n", MAXPOWT-powtorz);
        if (sendto(gniazdo, napis, napisDl, 0, (struct sockaddr *)&SerwAdr,
                          sizeof(SerwAdr)) != napisDl)
          ObslugaBledu("sendto() failed");
        /* Ustaw ponownie budzik */
        alarm(TIMEOUT);
      }
      else
        ObslugaBledu("Brak odpowiedzi");
    }
    else
      ObslugaBledu("recvfrom()");

/* Otrzymałem dane, wyłącz budzik */
  alarm(0);

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

  close(gniazdo);
  exit(0);
}

void PrzechwycAlarm(int ignoruj)
{ liczbaPowtorzen += 1; }
```

Zadanie:

Sprawdź działanie programu. Zastąp funkcję obsługi sygnału `sigaction()` funkcją `signal()`. Czy program nadal działa w ten sam sposób? Jeśli nie, wyjaśnij powód.

4.3. Przykładowa biblioteka podstawowych funkcji dla programów klienckich

- Biblioteka zaproponowana w Comer, Stevens "Sieci komputerowe", tom 3.
- Podstawowe funkcje: utworzenie gniazda i nawiązanie połączenia

```
socket = connectTCP(nazwa_komputera, usługa);
socket = connectUDP(nazwa_komputera, usługa);
```

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

```
/*
************************************************
* connectTCP - nawiazanie lacznosci z serwerem
* wskazanej uslugi TCP na wskazanym komputerze
************************************************
*/
int connectTCP(const char *host, const char *service )
/*
 * Argumenty:
 *      host    - nazwa hosta, z którym chcemy się połaczyć
 *      service - usługa związana z określonym portem
 */
{
  return connectsock( host, service, "tcp");
}
```

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

```
/*
************************************************
* connectUDP - otwarcie gniazda komunikujacego sie
* z serwerem wskazanej uslugi UDP na wskazanym
* komputerze.
************************************************
*/
int connectUDP(const char *host, const char *service )
/*
 * Argumenty:
 *      host    - nazwa hosta, z którym chcemy się połaczyć
 *      service - usługa związana z określonym portem
 */
{
  return connectsock(host, service, "udp");
}
```

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

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

#include <netinet/in.h>
#include <arpa/inet.h>

#include <netdb.h>
#include <string.h>
#include <stdlib.h>

#ifndef  INADDR_NONE
#define  INADDR_NONE  0xffffffff
#endif  /* INADDR_NONE */

extern int  errno;

int  errexit(const char *format, ...);

/*------------------------------------------------------------------------
 * connectsock - utwórz i połącz gniazdo do komunikacji TCP lub UDP
 *------------------------------------------------------------------------
 */
int
connectsock(const char *host, const char *service, const char *transport )
/*
 * Argumenty:
 *      host      - nazwa hosta, z którym chcemy się połaczyć
 *      service   - usługa związana z określonym portem
 *      transport - nazwa protokołu transportowego ("tcp" lub "udp")
 */
{
  struct hostent  *phe;  /* struktura opisująca komputer w sieci  */
  struct servent  *pse;  /* struktura opisująca usługę  */
  struct protoent *ppe;  /* struktura opisująca protokół */
  struct sockaddr_in sin;  /* struktura adresowa    */
  int  s, type;  /* deskryptor gniazda, typ gniazda  */


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

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

/* Odwzoruj adres w postaci nazwy lub zapisu kropkowo dziesiętnego
   na adres binarny IP  */
  if ( phe = gethostbyname(host) )
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
  else if ( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
    errexit("can't get \"%s\" host entry\n", host);

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

```
/* Wybierz typ gniazda w zależności od protokołu */
  if (strcmp(transport, "udp") == 0)
    type = SOCK_DGRAM;
  else
    type = SOCK_STREAM;

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

/* Połącz gniazdo */
  if (connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0)
    errexit("can't connect to %s.%s: %s\n", host, service,
      strerror(errno));
  return s;
}
```

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

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>

/*------------------------------------------------------------------------
 * errexit - print an error message and exit
 *------------------------------------------------------------------------
 */
/* VARARGS1 */
int errexit(const char *format, ...)
{
  va_list  args;

  va_start(args, format);
  vfprintf(stderr, format, args);
  va_end(args);
  exit(1);
}
```

- Przykłady klientów TCP i UDP napisanych z użyciem proponowanej biblioteki:
  Comer, Stevens "Sieci komputerowe", tom 3
  klient usługi DAYTIME, wersja TCP, str. 117-118
  klient usługi TIME, wersja UDP, str-119-123
  klient usługi ECHO, wersja TCP, str.123-125
  klient usługi ECHO, wersja UDP, str. 125-126

Należy przeczytać:

Douglas E. Comer, David L. Stevens: *Sieci komputerowe TCP/IP, tom 3*: str. 96-127

W. Richard Stevens: *Unix, programowanie usług sieciowych, tom 1: API gniazda i XTI*: str. 110-138, 248-272