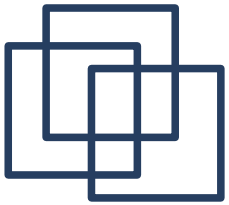
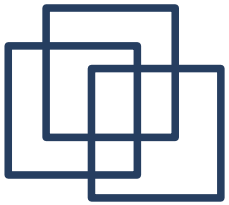


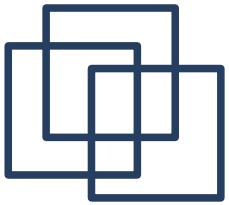
Podstawowe typy serwerów

1. Algorytm serwera.
2. Cztery podstawowe typy serwerów.
- iteracyjne, współbieżne,
- połączeniowe, bezpołączeniowe.
3. Problem zakleszczenia serwera.



Algorytm serwera

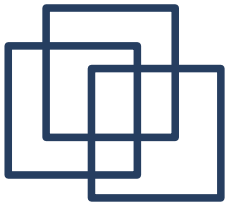
1. Utworzenie gniazda powiązanego z odpowiednim portem, pod którym będą przyjmowane zgłoszenia.
2. Przyjęcie zgłoszenia od klienta.
3. Obsługa zgłoszenia.
   - przetwarzanie danych,
   - sformatowanie i wysłanie odpowiedzi.
4. Powrót do punktu 2.



Serwery współbieżne i iteracyjne

Serwer iteracyjny obsługuje zgłoszenia sekwencyjnie. Jest łatwiejszy do zaprojektowania i konserwacji, jednak średni czas obsługi klienta może być długi ze względu na oczekiwanie przed rozpoczęciem obsługi zgłoszenia.

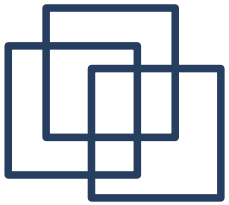
Serwer współbieżny może obsługiwać kilka zgłoszeń równocześnie. Zwykle zapewnia krótszy czas obsługi ale jest trudniejszy do zaprojektowania niż serwer iteracyjny. Termin “serwer współbieżny” jest używany niezależnie od tego, czy implementacja jest oparta na współbieżnie działających procesach czy też nie.



Serwery połączeniowe i bezpołączeniowe

Nawiązywanie logicznego połączenia zależy od protokołu warstwy transportowej, z którego korzysta klient, aby uzyskać dostęp do serwera. Serwer używający protokołu TCP jest serwerem **połączeniowym**, natomiast serwer używający protokołu UDP jest serwerem **bezpołączeniowym**.

Przy projektowaniu serwera trzeba pamiętać o tym, że protokół wykorzystywany przez warstwę aplikacji (zastosowań) może dodatkowo narzucać różne ograniczenia dla protokołu warstwy transportowej.



Serwery połączeniowe

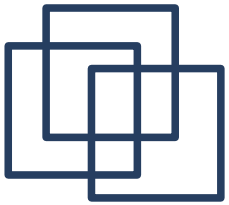
Serwer przyjmuje od klienta zgłoszenie połączenia i po jego nawiązaniu używa tego połączenia do komunikacji z klientem, który je zainicjował. Po zakończeniu interakcji połączenie jest zamykane.

Podstawowe zalety:

- niezawodność transmisji danych zapewniana przez protokół transportowy (TCP),
- łatwość zaprojektowania i implementacji.

Podstawowe wady:

- oddzielne gniazdo dla każdego połączenia (zasoby systemowe),
- wrażliwość na awarie programów klienckich.



Serwery bezpołączeniowe

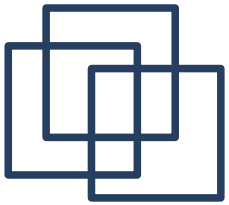
Komunikacja pomiędzy klientem i serwerem odbywa się bez nawiązywania połączenia (UDP).

Podstawowe zalety:

- mniejsze ryzyko wyczerpania zasobów serwera,
- wydajna transmisja danych,
- możliwość pracy w trybie rozgłoszeniowym.

Podstawowe wady:

- konieczność zapewnienia odpowiedniego poziomu niezawodności transmisji wymaganej przez aplikacje – wbudowanie zabezpieczeń w protokół warstwy aplikacji,
- problemy przy przenoszeniu aplikacji z sieci lokalnej do sieci rozległej.

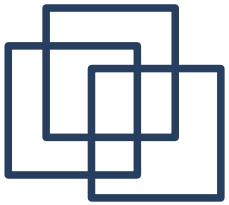


Serwery bezstanowe i wielostanowe

Serwery rejestrujące informację o stanie interakcji z klientami nazywamy serwerami **wielostanowymi**, natomiast te które nie przechowują takiej informacji, **bezstanowymi**.

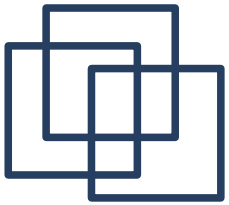
Zagadnienie bezstanowości serwerów musi być rozważane m. in. w związku z problemem zapewnienia niezawodności transmisji (UDP), jak również w zależności od używanego protokołu aplikacyjnego:

- serwery bezstanowe: ECHO, TIME,
- serwery wielostanowe: POP, IMAP, SMTP.



Optymalizacja serwerów bezstanowych

Optymalizacja serwera bezstanowego wymaga wielkiej ostrożności, ponieważ przechowywanie nawet niewielkiej ilości informacji o stanie interakcji może doprowadzić do wyczerpania zasobów w razie częstych awarii i wznowień programów klienckich lub w razie błędów transmisji, polegających na powielaniu komunikatów albo dostarczaniu ich z opóźnieniem.



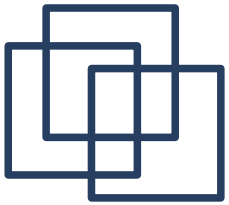
Podstawowe typy serwerów

Można wyróżnić cztery podstawowe typy serwerów:

- iteracyjne bezpołączeniowe,
- iteracyjne połączeniowe,
- współbieżne bezpołączeniowe,
- współbieżne połączeniowe.

Ocena działania serwera:

- czas przetwarzania zgłoszenia,
- obserwowany czas odpowiedzi.

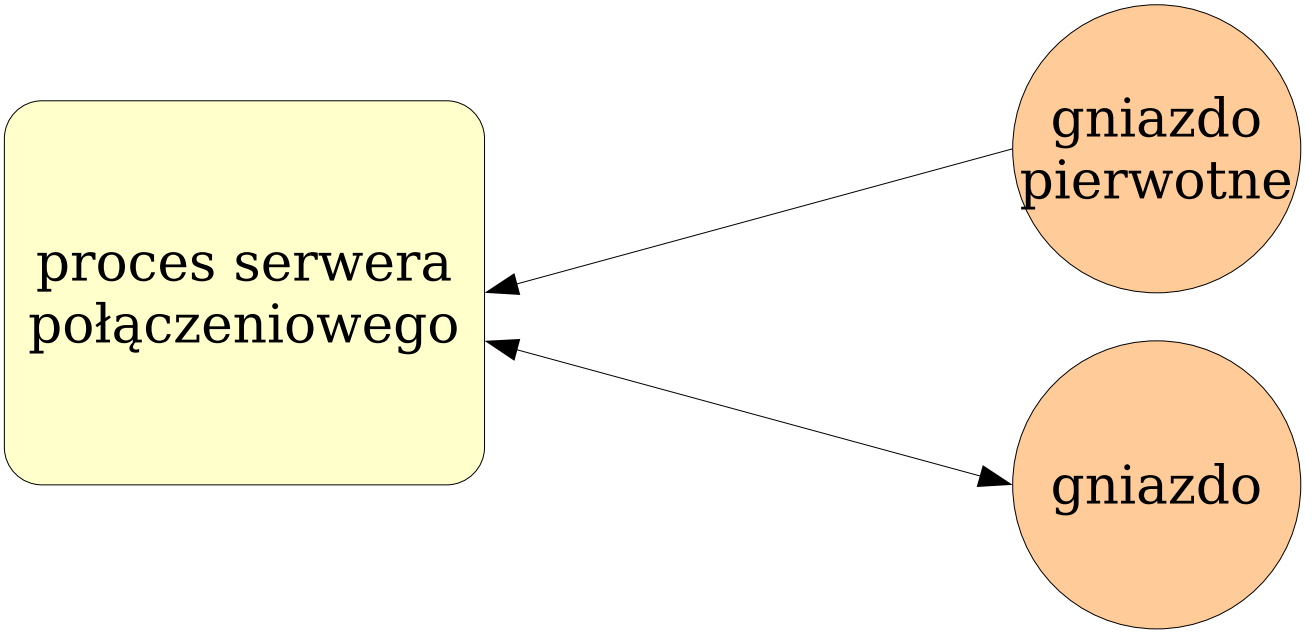


Problem zakleszczenia serwera

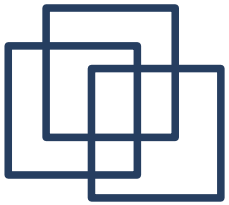
Niepoprawnie działający klient może spowodować zakleszczenie serwera jednoprocesowego, jeśli serwer używa funkcji systemowych, których wykorzystanie może zablokować proces serwera w trakcie wysyłania lub odbierania danych od klienta (np. funkcje **`read()`** lub **`write()`**). Podatność na zakleszczenie jest poważną wadą serwera, ponieważ oznacza, że określone zachowanie jednego klienta może uniemożliwić serwerowi obsługę innych klientów.

Iteracyjny serwer połączeniowy

Schemat struktury iteracyjnego serwera połączeniowego.



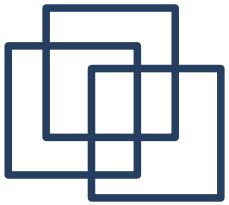
Gniazdo pierwotne jest związane z powszechnie znanym portem odpowiadającym realizowanej przez serwer usłudze. Za jego pośrednictwem proces serwera oczekuje na połączenia klientów. Po nawiązaniu połączenia tworzone jest osobne gniazdo przeznaczone do komunikacji z klientem.



Iteracyjny serwer połączeniowy

Zwykle najprostszy w implementacji jest algorytm iteracyjnego serwera połączeniowego.

1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnionej przez serwer.
2. Ustaw bierny tryb pracy gniazda.
3. Przyjmij zgłoszenie połączenia nadesłane na adres gniazda. Uzyskaj nowe gniazdo do obsługi tego połączenia.
4. Odpowiadaj na komunikaty klienta zgodnie z obsługiwanym protokołem.
5. Po zakończeniu obsługi zamknij połączenie i wróć do kroku 3.



Powiązanie gniazda z usługą

Do powiązania gniazda z określoną usługą służy funkcja **bind()** zadeklarowana w pliku **sys/socket.h**:

```
int bind(int s, const struct sockaddr *address, int len);
```

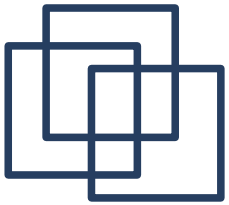
gdzie:

s – deskryptor gniazda zwrócony przez funkcję **socket()**,

address – wskaźnik do struktury zawierającej adres internetowy klienta. W przypadku programu serwera zwykle **address** wskazuje na tzw. **adres wieloznaczny** zdefiniowany stałą **INADDR_ANY**,

len – rozmiar struktury wskazywanej przez **address**.

Wartość zwracana: **0** w przypadku powodzenia, **-1** w przypadku błędu – ustawiana zmienna **errno**.



Tryb bierny gniazda

Do ustawienia gniazda w tryb bierny służy funkcja **`listen()`**

zadeklarowana w pliku **`sys/socket.h`**:

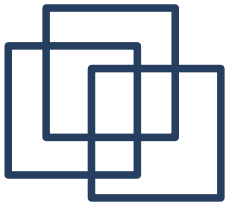
```
int listen(int s, int qlen);
```

gdzie:

`s` – deskryptor gniazda związanego z usługą,

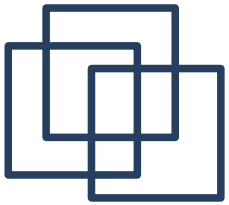
`qlen` – długość wewnętrznej kolejki zgłoszeń związanej z gniazdem – maksymalnie **`SOMAXCONN`**,

Wartość zwracana: **`0`** w przypadku powodzenia, **`-1`** w przypadku błędu – ustawiana zmienna **`errno`**.



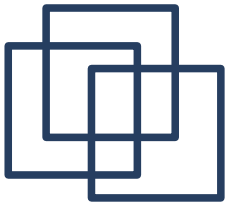
Utworzenie biernego gniazda

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

#include <netinet/in.h>
#include <netdb.h>

#include <stdio.h>
#include <errno.h>

int passivesock(char *service, char *protocol, int qlen){
  struct servent *pse;
  struct protoent *ppe;

  struct sockaddr_in sin;
  int type, s;
```



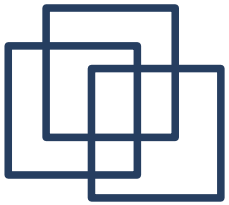
Utworzenie biernego gniazda

```
bzero((char*) &sin, sizeof(sin));
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
if (pse = getservbyname(service, protocol)){
  sin.sin_port = pse->s_port;
}else if ((sin.sin_port = htons((u_short)atoi(service)))==0){
  fprintf(stderr, "getservbyname: %s\n", strerror(errno));
  return -1;
}
if ((ppe = getprotobyname(protocol))==NULL){
  fprintf(stderr, "getprotobyname: %s\n", strerror(errno));
  return -1;
}
if (strcmp(protocol, "udp")==0){
  type = SOCK_DGRAM;
}else{
  type = SOCK_STREAM;
}
```



Utworzenie biernego gniazda

```
  if ((s = socket(PF_INET, type, ppe->p_proto))<0){
    fprintf(stderr, "socket: %s\n", strerror(errno));
    return -1;
  }
  if(bind(s, (struct sockaddr *)&sin, sizeof(sin))<0){
    fprintf(stderr, "bind: %s\n", strerror(errno));
    return -1;
  }
  if (type==SOCK_STREAM && listen(s, qlen)<0){
    fprintf(stderr, "listen: %s\n", strerror(errno));
    return -1;
  }
  return s;
}
```



Przyjmowanie połączeń

Do przyjmowania zgłoszenia połączenia służy funkcja **accept()** zadeklarowana w pliku **sys/socket.h**:

```
int accept(int s, struct sockaddr *address, int *len);
```

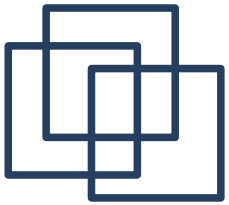
gdzie:

s – deskryptor gniazda dla którego wywołano funkcje **bind()** i **listen()**,

address – wskaźnik do struktury, w którą zostaną wpisane dane o adresie klienta.

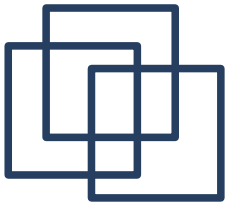
len – wskaźnik do zmiennej określającej rozmiar struktury ***address**.

Wartość zwracana: deskryptor gniazda utworzonego w celu obsługi połączenia lub **-1** w przypadku błędu – ustawiana zmienna **errno**.



Obsługa połączenia

Po zaakceptowaniu połączenia transmisja danych odbywa się z wykorzystaniem funkcji **`read()`** i **`write()`**. Po zakończeniu wymiany danych deskryptor przyznanego przez **`accept()`** gniazda powinien zostać zwolniony za pomocą funkcji **`close()`**.

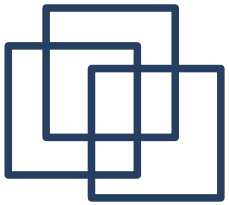


Iteracyjny serwer połączeniowy

Serwer usługi ECHO:

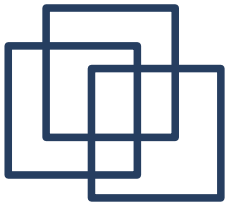
Stałe LINELEN i TESTPORT powinny być zdefiniowane w programie głównym.

```
int svr_echo_tcp(){

    int s1, s2, alen;
    struct sockaddr_in sin;

    alen = sizeof(sin);
    if((s1=passivesock(TESTPORT, "tcp", 10))<0){
       fprintf(stderr, "passivesock: %s\n", strerror(errno));
       return -1;
    }
    printf("tcp echo serwer uruchomiony\n");
```



Iteracyjny serwer połączeniowy

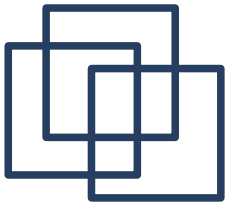
```
   while(1){
        if((s2=accept(s1, (struct sockaddr *)&sin, &alen))<0){
           fprintf(stderr, "accept: %s\n", strerror(errno));
           return -1;
        }

        echod(s2);

        if (close(s2)<0){
           fprintf(stderr, "close: %s\n", strerror(errno));
           return -1;
        }
     }
}
```



Iteracyjny serwer połączeniowy

Funkcja **echod()** obsługuje komunikację z klientem.

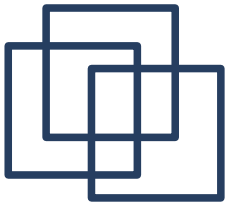
```
int echod(int s){
     char buf[LINELEN+1]; int n;

     while(n=read(s, buf, LINELEN)){
        if (n<0){
           fprintf(stderr, "read: %s\n", strerror(errno));
           return -1;
        }
        if (write(s, buf, n)<0){
           fprintf(stderr, "write: %s\n", strerror(errno));
           return -1;
        }
        buf[n] = '\0'; printf("%s", buf);
     }
     return 1;
}
```



Zakleszczenia

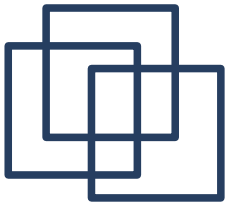
Zaprezentowany serwer iteracyjny jest wrażliwy na na niepoprawne działanie klientów. Program serwera zawiera instrukcje, które mogą prowadzić do zakleszczenia:

- **`read()`** - wstrzymuje działanie programu do czasu odebrania danych, lub osiągnięcia ,,końca pliku”. W przypadku gdy klient nie wyśle danych ani nie zamknie połączenia serwer zatrzyma się. Podobny scenariusz może mieć miejsce w instrukcji **`close()`**.
- **`write()`** - wstrzymuje działanie programy do czasu przekazania wysyłanych danych do protokołu warstwy transportowej (TCP). Jeśli serwer będzie generował nowe komunikaty a klient nie będzie ich odbierał, to po pewnym czasie bufory TCP po obu stronach zostaną wypełnione. Kolejna instrukcja **`write()`** zatrzyma serwer.



Iteracyjny serwer połączeniowy

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

public class EchoTCPServer {
    public static void main(String args[]){
        Socket s;
        OutputStream os;
        InputStream is;
        byte[] buffer = new byte[100];
        int i;
```

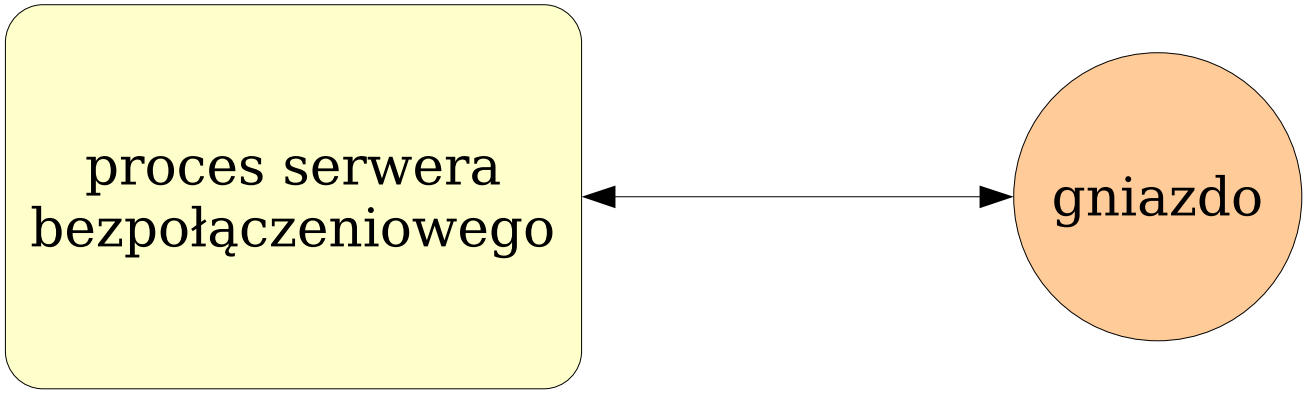


Iteracyjny serwer połączeniowy

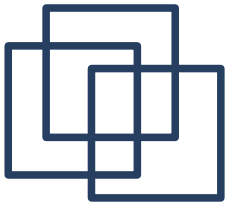
```
        try {
            ServerSocket ss = new ServerSocket(TESTPORT);
            while(true){
                s = ss.accept();
                is = s.getInputStream();
                os = s.getOutputStream();
                while (true){
                    if ((i = is.read(buffer))<0){
                        break;
                    }
                    os.write(buffer, 0, i);
                }
                s.close();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Itewracyjny serwer bezpołączeniowy

Schemat struktury iteracyjnego serwera bezpołączeniowego.



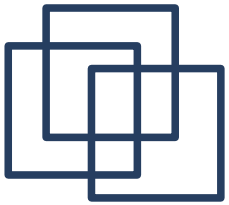
Jeden proces serwera komunikuje się kolejno z wieloma klientami przez jedno gniazdo. Gniazdo to jest związane z powszechnie znanym portem odpowiadającym realizowanej przez serwer usłudze.



Iteracyjny serwer bezpołączeniowy

Iteracyjne serwery bezpołączeniowe są używane do zadań, w których czas przetwarzania danych jest krótki. Transport bezpołączeniowy pozwala też na efektywne przesyłanie niewielkiej ilości danych.

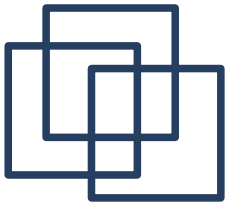
1. Utwórz gniazdo i zwiąż je z powszechnie znanym adresem odpowiadającym usłudze udostępnionej przez serwer.
2. Odpowiadaj na kolejne komunikaty otrzymywane od klientów zgodnie z obsługiwanym protokołem.



Adresowanie odpowiedzi

W przypadku serwerów bezpołączeniowych do komunikacji nie można używać funkcji **`read()`** i **`write()`**, ponieważ ograniczają one możliwość wymiany datagramów przez gniazdo do komunikacji z jednym komputerem i jednym portem na tym komputerze. Co więcej bez nawiązania połączenia funkcją **`connect()`** nie ma zdefiniowanego żadnego adresu docelowego do transmisji przez utworzone wcześniej gniazdo.

Problem można rozwiązać używając funkcji **`recvfrom()`** i **`sendto()`**.



Odbieranie danych

Do obierania danych od klientów w serwerach bezpołączeniowych używa się funkcji **recvfrom()** zdeklarowaną w pliku **sys/socket.h**.

```
int recvfrom(int s, char *buffer, int blen, int flags, struct sockaddr *address, int *alen);
```

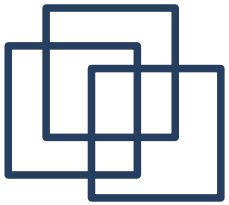
s – deskryptor gniazda,

buffer – wskaźnik do bufora o długości **blen**, w którym zostaną umieszczone odebrane dane,

flags – parametr kontrolujący odbiór wiadomości. Może składać się z **MSG_PEEK**, **MSG_OOB**, **MSG_WAITALL**. Zwykle używa się wartości **0**,

address – wskaźnik do struktury o rozmiarze ***alen**, w którą zostaną wpisane dane o adresie klienta.

Wartość zwracana: długość wiadomości lub **-1** w przypadku błędu (**errno**).



Wysyłanie danych

Do wysyłania danych do klientów w serwerach bezpołączeniowych używa się funkcji **sendto()** zdeklarowaną w pliku **sys/socket.h**.

```
int sendto(int s, char *buffer, int blen, int flags, struct
  sockaddr *address, int alen);
```

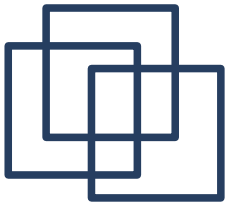
s – deskryptor gniazda,

buffer – wskaźnik do bufora z danymi do wysłania, **blen** - rozmiar komunikatu,

flags – parametr kontrolujący wysyłanie danych. Może składać się z **MSG_OOB**, **MSG_DONTROUTE**. Zwykle używa się wartości **0**,

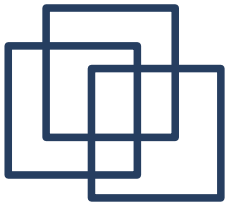
address – wskaźnik do struktury o rozmiarze **alen**, określającą adresata wiadomości.

Wartość zwracana: liczba wysłanych bajtów lub **-1** w przypadku błędu (**errno**).



Iteracyjny serwer bezpołączeniowy

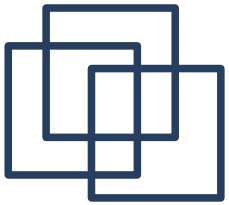
```
int svr_echo_udp(){

    int s, alen;
    struct sockaddr_in sin;
    char buf[LINELEN+1];

    alen = sizeof(sin);
    if ((s=passivesock(TESTPORT, "udp", 0))<0){
       fprintf(stderr, "passivesock: %s\n",
          strerror(errno));
       return -1;
    }

    printf("udp echo serwer uruchomiony\n");
```



Iteracyjny serwer bezpołączeniowy

```
while(1){
        if (recvfrom(s, buf, sizeof(buf), 0,
              (struct sockaddr *)&sin, &alen)<0){
           fprintf(stderr, "recvfrom: %s\n", strerror(errno));
           return -1;
        }
        if (sendto(s, buf, strlen(buf)+1, 0,
              (struct sockaddr *)&sin, sizeof(sin))<0){
           fprintf(stderr, "sendto: %s\n", strerror(errno));
           return -1;
        }
        printf("odeslalem: %s\n", buf);
     }
}
```

Serwer działa w nieskończonej pętli realizując kolejne żądania klientów. Do kontaktów z wszystkimi klientami jest używane jedno gniazdo.



Podsumowanie

Serwery iteracyjne są zwykle prostsze do implementacji jednak nie zapewniają szybkiej reakcji serwera w przypadku dużego obciążenia. W takim przypadku często lepiej zaprojektować i zaimplementować serwer współbieżny.

Serwery bezpołączeniowe są dosyć odporne na różne awarie sieci. W przypadku serwerów połączeniowych dodatkowo mogą wystąpić problemy związane z zakleszczeniami.

Ważne instrukcje:

funkcje **`bind()`**, **`listen()`**, **`accept()`**, **`recvfrom()`**, **`sendto()`**.