



# Programowanie Sieciowe 1

dr inż. Tomasz Jaworski  
[tjaworski@iis.p.lodz.pl](mailto:tjaworski@iis.p.lodz.pl)  
<http://tjaworski.iis.p.lodz.pl/>

# Standardowe protokoły transportowe

- TCP (Transmission Control Protocol)
- UDP (User Datagram Protocol)

# TCP – Transmission Control Protocol

- Protokół zorientowany na **połączenia**,
- Potwierdzenia odbioru danych przesyłane do nadawcy
  - W przypadku błędu przesyłu następuje **retransmisja** lub zerwanie połączenia
- Kontrola poprawności przesyłania danych,
- Odebrane dane przed przekazaniem do warstwy wyższej układane są w odpowiedniej kolejności
- **Większy narzut transmisji,**
- **Wysoki stopień niezawodności,**
- Komunikacja na dużych odległościach

# UDP – User Datagram Protocol

- Brak mechanizmu połączeń,
  - Serwer może przyjąć datagramy od kilku różnych hostów na tym samym porcie,
- Brak kontroli zgubienia pakietu,
  - Poprawność transmisji określana jest na podstawie jedynie **sumy kontrolnej** datagramu,
  - Brak potwierdzenia otrzymania danych,
- Datagramy mogą zostać odebrane w różnej kolejności,
- Mniejszy narzut na transmisję,
- Pomimo swoich wad dobrze sprawdzają się w sieci lokalnej (gdzie niezawodność połączenia fizycznego jest wysoka)

# Wybrane zagadnienia z programowania sieciowego (wszechobecne)

- Adres IP - identyfikacja hosta,
- Numer portu, - identyfikacja aplikacji,
- Para gniazdowa, - identyfikacja połączenia,
- Konwersja danych,
- Co to jest uchwyt?

# Adres IP (dla IPv4)

- Liczba całkowita, 32-bitowa bez znaku; w języku C unsigned long, zakres: 0x00000000 – 0xFFFFFFFF (0 – 4,294,967,295),
- Umożliwia identyfikację komputera w sieci Internet
  - Przypadkiem szczególnym jest **współdzielenie łącza** (NAT/Maskarada)
- Dla użytkownika przewidziana jest postać „**A.B.C.D**”, gdzie litery to liczby 8-bitowe bez znaku.

Przykład:

0x7F000001 = 7F.00.00.01 = **127.0.0.1**

0xC0A81401 = C0.A8.14.01 = **192.168.20.1**

A jeśli programy **A** i **B** na hoście **M** będą chciały skomunikować się z odpowiednio programami **A** i **B** na hoście **N**?

# Numer portu

- Liczba całkowita, 16-bitowa bez znaku; w języku C `unsigned short`, zakres: `0x0000 – 0xFFFF` (0 – 65535),
- Ten sam numer (wartość) portu może być wykorzystana dla protokołu TCP i UDP,
- Umożliwia identyfikację połączenia dla danego numeru IP,
- Dany port może być przypisany tylko do jednej aplikacji,
- Para uporządkowana (`numer_ip:port`) **jednoznacznie** określa hosta oraz aplikację z którą to połączenie jest nawiązane,
- Przykłady portów:
  - **53** – DNS,
  - **80** – Serwer HTTP, 8080 – najczęściej serwer proxy,
  - **21, 21** – Serwer FTP,
  - **25** – SMTP (poczta wychodząca)
  - **110** – POP3 (poczta przychodząca),
  - **22** – SSH

# Para gniazdowa

- **Cztery elementy** definiujące dwa punkty końcowe połączenia:
  - Adres IP lokalny,
  - Port lokalny,
  - Adres IP zdalny,
  - Port zdalny
- Czwórka umożliwia **jednoznacznie** identyfikuje dane połączenie TCP w sieci.
- W przypadku protokołu UDP czwórka jednoznacznie określa źródło i cel datagramu znajdującego się w sieci.



# Konwersja danych

- Liczba 3735928559 (0xDEADBEEF) zapisana dane w formacie:

**Little-endian** (najmniej znaczący bajt jako pierwszy):

.EF .BE .AD .DE .

- Procesory z rodziny x86 (Intel)

**Big-endian** (najbardziej znaczący bajt jako pierwszy):

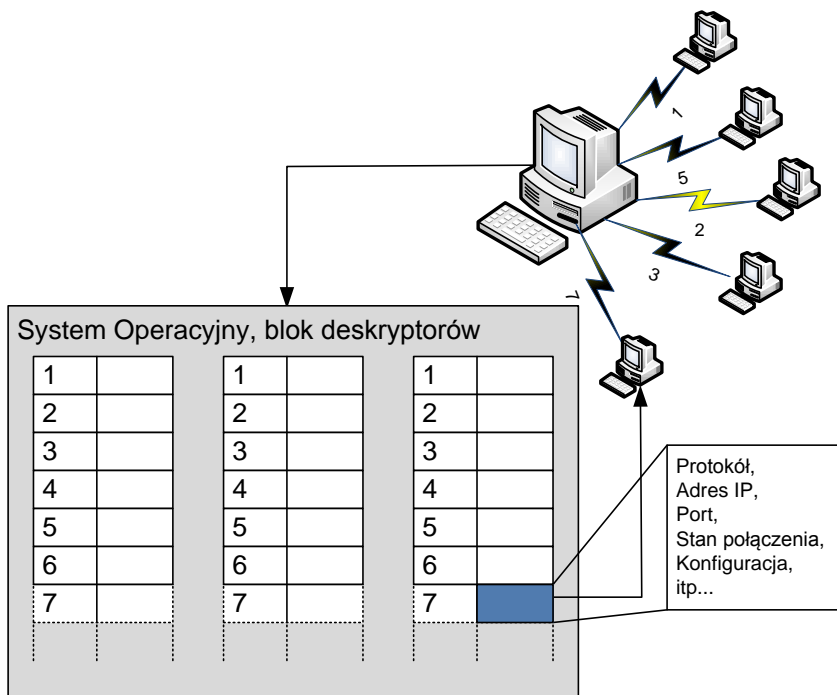
.DE .AD .BE .EF .

- Procesory 68000 Motorola

Format sieciowy (*Network Byte Order*): **big endian**

# Uchwyt (deskryptor)

- Uchwyt jest **formą** wskaźnika do struktury systemowej opisującej dany zasób,
- W przeciwieństwie do wskaźnika, uchwyt umożliwia **weryfikację** istnienia informacji na którą wskazuje

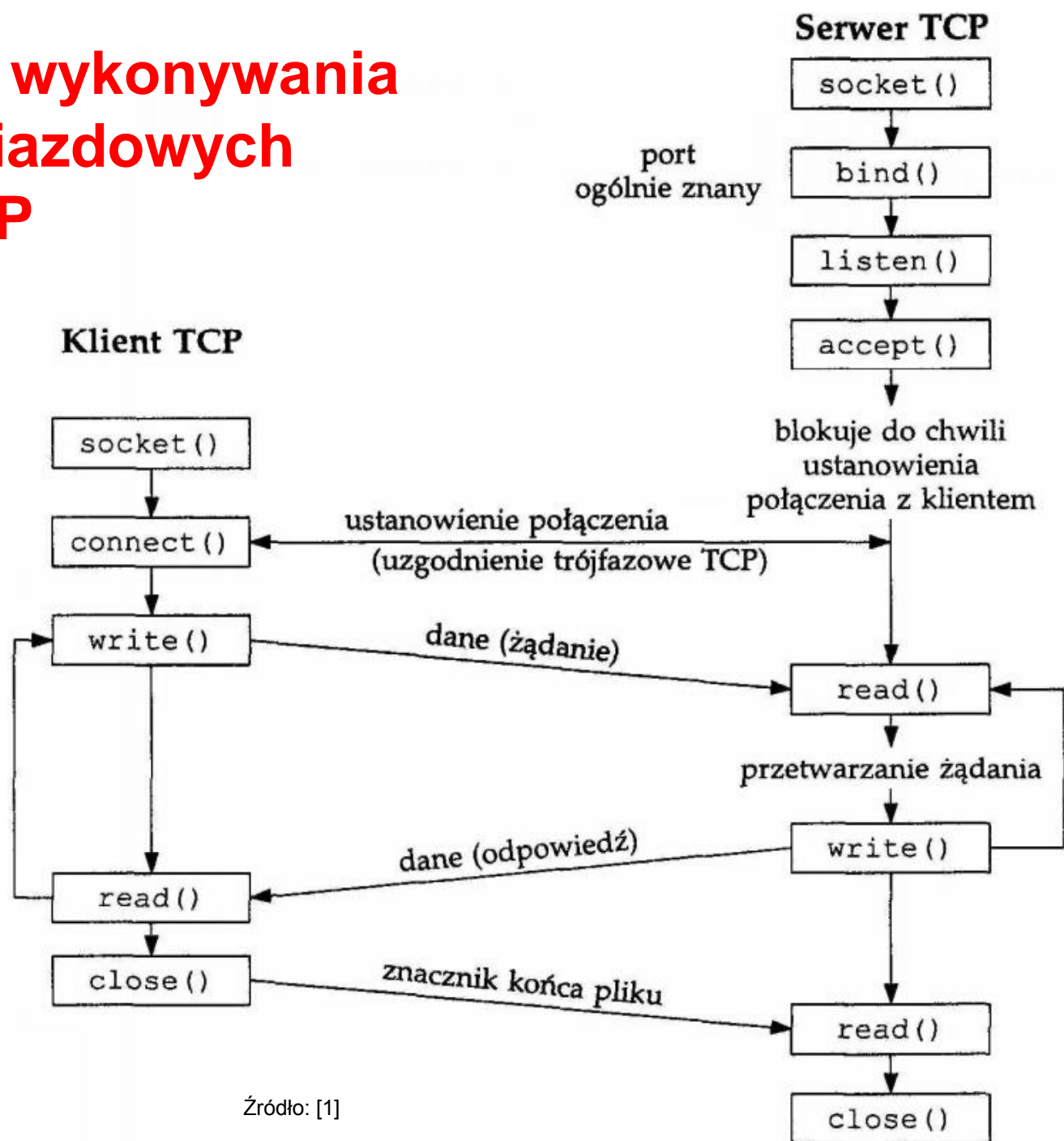


# Operacje blokujące i nieblokujące

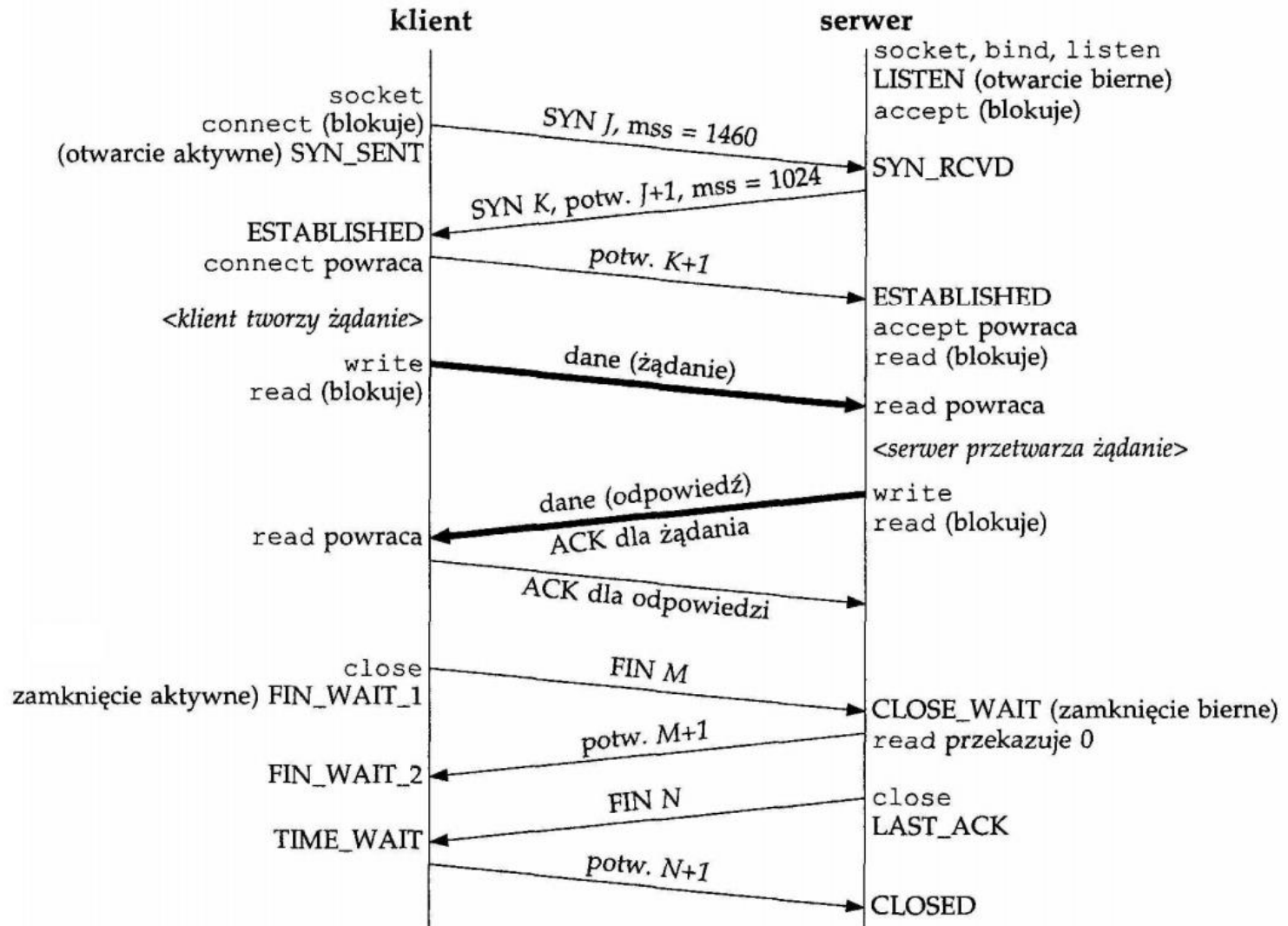
**Operacje blokujące (blocking)**: operacje gniazdowe wykonywane są synchronicznie. Funkcje kończą się po całkowitym wykonaniu operacji lub w przypadku błędu.

**Operacje nieblokujące (nonblocking)**: operacje gniazdowe wykonywane są asynchronicznie. Funkcje kończą się zawsze, t/j. po zleceniu podsystemowi gniazd danej operacji (np. `connect`, `send`, `recv`) lub w przypadku błędu.

# Kolejność wykonywania funkcji gniazdowych klienta TCP



# Wymiana pakietów przez połączenie TCP



# Klient: Tworzenie gniazda [1]

socket ► connect ► send ► recv ► close

```
int socket(int family, int type, int proto);
```

Funkcja tworzy nowe gniazdo w systemie i konfiguruje je do zadanego protokołu.

- *family* – rodzina protokołów:
  - **AF\_INET** – protokół IPv4,
  - **AF\_INET6** – protokół IPv6
- *type* – rodzaj gniazda:
  - **SOCK\_STREAM** – gniazdo strumieniowe,
  - **SOCK\_DGRAM** – gniazdo datagramowe,
  - **SOCK\_RAW** – gniazdo surowe (raw),
- *proto* – protokół (dla *type*=**SOCK\_RAW**):
  - **0** – Domyślny protokół (**SOCK\_STREAM**=**TCP**, **SOCK\_DGRAM**=**UDP**),
- **Wynik:** **uchwyt gniazda**, lub:
  - **INVALID\_SOCKET**, kod błędu z *WSAGetLastError* (Windows),
  - **-1**, kod błędu z *errno* (Unix)

# Klient: Tworzenie gniazda [1]

socket ► connect ► send ► recv ► close

Domyślnie utworzone gniazdo jest **blokujące**. Oznacza to, że każda operacja, która wymaga wymiany danych z drugą stroną połączenia, będzie oczekiwała tak długo, aż dojdzie do komunikacji lub zmieni się stan gniazda (chyba, że dane już są w buforze gniazdowym).

W przypadku gniazd **nieblokujących**, operacje kończą się natychmiast. Jeśli dane były w buforze lub zostały już wysłane, operacja kończy się pomyślnie. Jeśli nie, operacja kończy się „*błędem*” (zwraca **SOCKET\_ERROR**) a wartość błędu dla gniazda zawiera kod **WSAEWOULDBLOCK/EWOULDBLOCK**. **Błąd ten oznacza, że w przypadku gniazda blokującego, operacja wstrzymałaby wykonywanie programu.**

# Klient: Tworzenie gniazda [2]

socket ► connect ► send ► recv ► close

**W przypadku systemów Windows konieczne przed skorzystaniem z funkcji gniazdowych aplikacja musi zainicjować system Winsock.**

```
WSADATA wsaData;  
int nCode;  
char errdesc[100];  
  
if ((nCode = WSStartup(MAKEWORD(1, 1), &wsaData)) != 0)  
{  
    sprintf(errdesc,  
            "Bład podczas inicjalizacji biblioteki WinSock."  
            "Bład %d", nCode);  
    exit(1);  
}  
printf("WinSock: %s [%s]Wn", wsaData.szDescription,  
        wsaData.szSystemStatus);  
printf("MaxSockets: %dWn", wsaData.iMaxSockets);
```



## Klient: Tworzenie gniazda [3]

socket ► connect ► send ► recv ► close

### Przykłady:

```
SOCKET sock_fd = socket(AF_INET, SOCK_STREAM,  
                        IPPROTO_TCP);
```

```
SOCKET sock_fd = socket(AF_INET, SOCK_STREAM, 0);
```

```
int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
```

```
SOCKET sock_fd = socket(AF_INET, SOCK_DGRAM, 0);
```

# Klient: Nawiązywanie połączenia [1]

socket ► connect ► send ► recv ► close

```
int connect(int sock, sockaddr *rmt, int rmtlen);
```

Funkcja nawiązuje połączenie ze zdalnym hostem, określonym w *rmt*. Wykonuje tzw. **otwarcie aktywne**.

- *sock* – uchwyt gniazda (zwrócony przez `socket`)
- *rmt* – wskaźnik na strukturę `sockaddr` przechowującą adres zdalnego hosta oraz informację o protokole,
- *rmtlen* – długość, w bajtach, struktury `sockaddr` dla danego protokołu.
- **Wynik:** 0, lub:
  - **SOCKET\_ERROR**, kod błędu z `WSAGetLastError` (Windows),
  - -1, kod błędu z `errno` (Unix)
- **Blocking:** Funkcja `connect` próbuje się połączyć ze zdalnym hostem przez określony czas. W przypadku porażki zwraca `SOCKET_ERROR`
- **Nonblocking:** Wtedy wynik = `SOCKET_ERROR` a kod błędu = `WSAEWOULDBLOCK/EWOULDBLOCK`.

# Klient: Nawiązywanie połączenia [2]

socket ► connect ► send ► recv ► close

```
typedef struct sockaddr {
    u_short sa_family;
    CHAR sa_data[14];
} SOCKADDR;

typedef struct sockaddr_in {
    short sin_family;
    unsigned short sin_port;
    IN_ADDR sin_addr;
    CHAR sin_zero[8];
} SOCKADDR_IN, *PSOCKADDR_IN;
```

```
typedef struct in_addr {
    union {
        struct {
            UCHAR s_b1, s_b2, s_b3,
            s_b4;
        } S_un_b;
        struct {
            USHORT s_w1,s_w2;
        } S_un_w;
        ULONG S_addr;
    } S_un;
} IN_ADDR, *PIN_ADDR, *LPIN_ADDR;
```

```
sockaddr_in service;
service.sin_family = AF_INET;
service.sin_addr.s_addr =
    inet_addr("127.0.0.1" );
service.sin_port = htons(3370);
```

```
#define s_addr S_un.S_addr
#define s_host S_un.S_un_b.s_b2
#define s_net S_un.S_un_b.s_b1
#define s_imp S_un.S_un_w.s_w2
#define s_impno S_un.S_un_b.s_b4
#define s_lh S_un.S_un_b.s_b3
```

# Klient: Nawiązywanie połączenia [3]

socket ► connect ► send ► recv ► close

```
sockaddr_in service;  
service.sin_family = AF_INET;  
service.sin_port = htons(3370);  
service.sin_addr.s_addr = ??????;
```

Adres IP w postaci tekstowej:

```
.s_addr = inet_addr("127.0.0.1" );
```

A co zrobić z postacią tekstową, domeną?

**google.pl = 74.125.77.147 ?**

#define AF_UNSPEC	0	// unspecified
#define AF_UNIX	1	// local to host (pipes, portals)
<b>#define AF_INET</b>	<b>2</b>	<b>// internetwork: UDP, TCP, etc.</b>
#define AF_IMPLINK	3	// arpanet imp addresses
#define AF_PUP	4	// pup protocols: e.g. BSP
#define AF_CHAOS	5	// mit CHAOS protocols
#define AF_NS	6	// XEROX NS protocols
#define AF_IPX	AF_NS	// IPX protocols: IPX, SPX, etc.
#define AF_ISO	7	// ISO protocols
#define AF_OSI	AF_ISO	// OSI is ISO
#define AF_ECMA	8	// european computer manufacturers
#define AF_DATAKIT	9	// datakit protocols
#define AF_CCITT	10	// CCITT protocols, X.25 etc
#define AF_SNA	11	// IBM SNA
#define AF_DECnet	12	// DECnet
#define AF_DLI	13	// Direct data link interface
#define AF_LAT	14	// LAT
#define AF_HYLINK	15	// NSC Hyperchannel
#define AF_APPLETALK	16	// AppleTalk
#define AF_NETBIOS	17	// NetBios-style addresses
#define AF_VOICEVIEW	18	// VoiceView
#define AF_FIREFOX	19	// Protocols from Firefox
#define AF_UNKNOWN1	20	// Somebody is using this!
#define AF_BAN	21	// Banyan
#define AF_ATM	22	// Native ATM Services
<b>#define AF_INET6</b>	<b>23</b>	<b>// Internetwork Version 6</b>
#define AF_CLUSTER	24	// Microsoft Wolfpack
#define AF_12844	25	// IEEE 1284.4 WG AF
#define AF_IRDA	26	// IrDA
#define AF_NETDES	28	// Network Designers OSI & gateway

# Klient: Nawiązywanie połączenia [4]

socket ► connect ► send ► recv ► close

Aby otrzymać adres IP odpowiadający znanej nazwie hosta, należy skorzystać z serwera DNS poprzez funkcję **gethostbyname**:

```
hostent* h = gethostbyname("google.pl" );  
if (h == NULL)  
{  
    printf(„blad” ); exit(1);  
}  
service.sin_addr =  
    *(struct in_addr*)h->h_addr_list[0];
```

# Klient: Zapisywanie danych do gniazda

socket ► connect ► **send** ► recv ► close

```
int send(int sock,  
         const char* buf, int len, int flags);
```



Funkcja zapisuje dane do bufora nadawczego gniazda

- *sock* – uchwyt gniazda (zwrócony przez **socket** lub **accept**),
- *buf* – wskaźnik do bufora zawierającego dane do wysłania,
- *buflen* – ilość bajtów do wysłania,
- *flags* – flagi, domyślnie **0**,
- **Wynik:** ilość wysłanych bajtów (**blocking**) lub ilość wysłanych bajtów  $\leq$  *buflen* (**nonblocking**) lub:
  - **SOCKET\_ERROR**, kod błędu z *WSAGetLastError* (Windows),
  - **-1**, kod błędu z *errno* (Unix)

# Klient: Wczytywanie danych z gniazda

socket ► connect ► send ► **recv** ► close

```
int recv(int sock,  
char *buf, int buflen, int flags);
```



Funkcja odczytuje dane z bufora odbiorczego gniazda

- *sock* – uchwyt gniazda (zwrócony przez **socket** lub **accept**),
- *buf* – wskaźnik do bufora docelowego,
- *buflen* – ilość bajtów do odczytania,
- *flags* – flagi, domyślnie **0**,
- **Wynik:**  $1 \leq$  ilość wysłanych bajtów  $\leq$  *buflen* (**blocking**), błąd **WSAEWOULDBLOCK** (**nonblocking**) lub:
  - **0** – gdy połączenie zostało zamknięte zdalnie lub lokalnie,
  - **SOCKET\_ERROR**, kod błędu z *WSAGetLastError* (Windows),
  - **-1**, kod błędu z *errno* (Unix)

# Klient: Zamykanie połączenia

socket ► connect ► send ► recv ► close

```
int closesocket(SOCKET sock);           // windows
int close(int sock);                     // unix
```

- Funkcja zamyka gniazdo a wraz z nim połączenie. Wszystkie aktywne operacje związane z gniazdem są anulowane. Jeśli w buforze wyjściowym znajdują się dane, system spróbuje je dostarczyć do hosta po czym rozpocznie procedurę zamykania połączenia.
- W przypadku zamkniętego połączenia zasoby związane z gniazdem są zwalniane.
- *sock* – uchwyt gniazda (zwrócony przez `socket` lub `accept`),
- **Wynik:** **0** gdy gniazdo zostało zamknięte, lub:
  - **SOCKET\_ERROR**, kod błędu z *WSAGetLastError* (Windows),
  - **-1**, kod błędu z *errno* (Unix)



```

int main(int argc, char* argv[])
{
    WSADATA data;
    int result;

    result = WSAStartup(MAKEDWORD(2, 0), &data);
    assert(result == 0);

    SOCKET sock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    assert(sock != INVALID_SOCKET);

    sockaddr_in service;
    service.sin_family = AF_INET;
    service.sin_port = htons(3301);
    service.sin_addr.s_addr = inet_addr("127.0.0.1");
    result = connect(sock, (sockaddr*)&service,
                      sizeof(sockaddr_in));
    assert(result != SOCKET_ERROR);

    char str[100];
    for(int i = 0; i < 3; i++) {
        if (!read_line(sock, str))
            break;
        printf("%d: %s", i, str);
    }
    closesocket(sock);
}

```

### Protokół serwera

```

Data 11/10/2010WrWn
Godzina 17:53:41WrWn
Jestes klientem #1WrWn

```

```

bool read_line(SOCKET sock, char* line)
{
    while(true)
    {
        int result = recv(sock, line, 1, 0);
        if (result == 0 || result == SOCKET_ERROR)
            return false;
        if (*line++ == '\n')
            break;
    }
    *line = '\0';
    return true;
}

```

### Protokół serwera

```

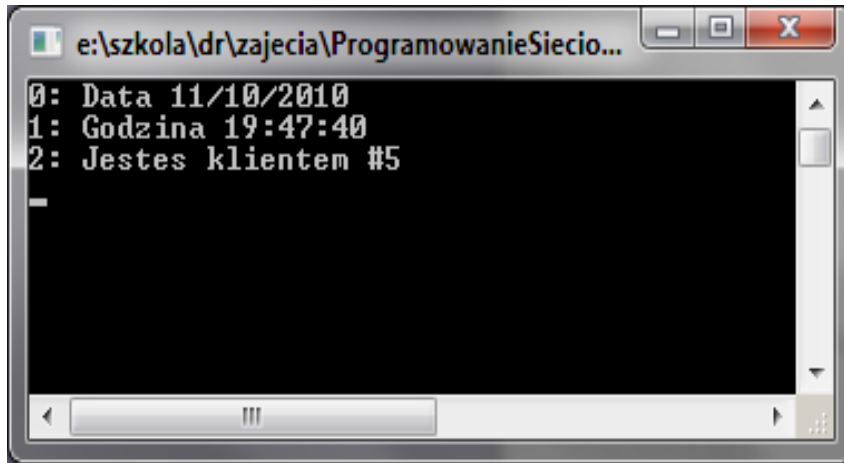
Data 11/10/2010\r\n
Godzina 17:53:41\r\n
Jestes klientem #1\r\n

```

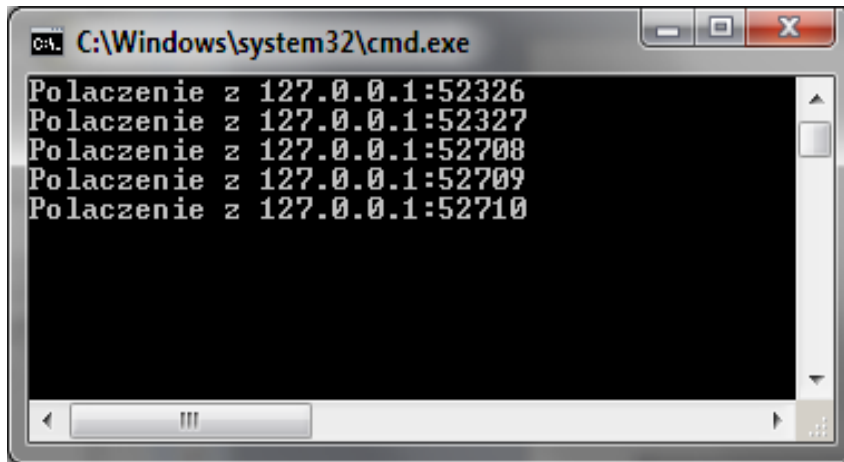
```
static void Main()
{
    Socket s = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Unspecified);
    s.Connect(new IPEndPoint(IPAddress.Parse("127.0.0.1"),
        3301));

    byte[] buffer = new byte[1024];
    int result = s.Receive(buffer);
    String time = Encoding.ASCII.GetString(buffer, 0,
        result);
    Console.WriteLine(time);
}
```

# Klient: Testowanie



```
e:\szkola\dr\zajecia\ProgramowanieSiecio...
0: Data 11/10/2010
1: Godzina 19:47:40
2: Jestes klientem #5
-
```



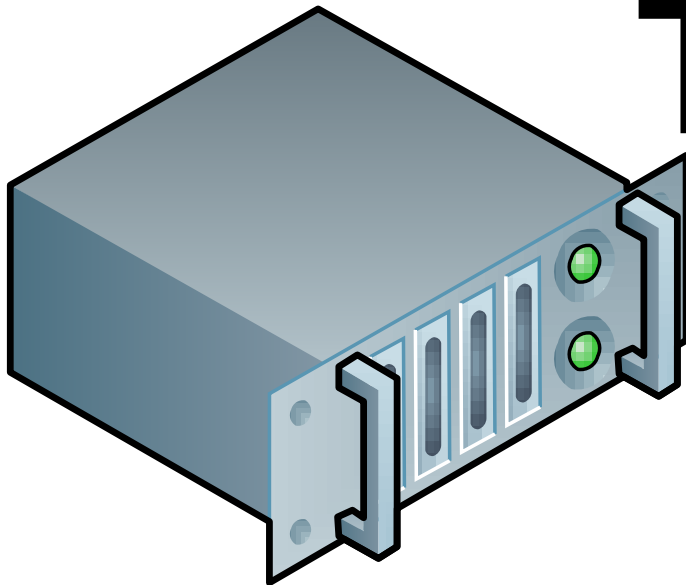
```
C:\Windows\system32\cmd.exe
Polaczenie z 127.0.0.1:52326
Polaczenie z 127.0.0.1:52327
Polaczenie z 127.0.0.1:52708
Polaczenie z 127.0.0.1:52709
Polaczenie z 127.0.0.1:52710
```

1. Uruchomić **serwer testowy** (opisany w dalszej części wykładu).
2. Program *telnet* w celu przetestowania serwera.
3. Uruchomić **klienta testowego**.

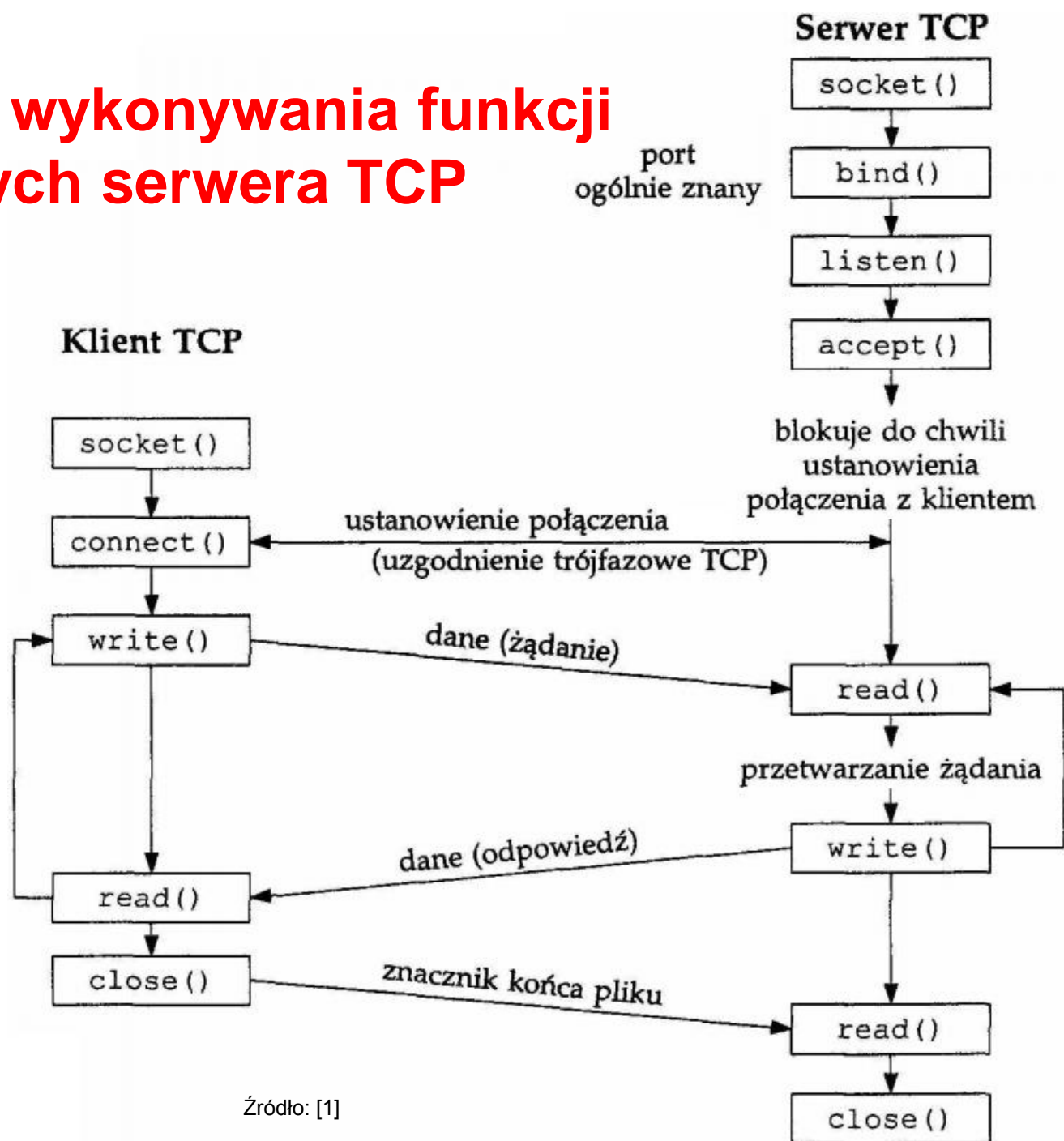
## Protokół serwera

```
Data 11/10/2010WrWn
Godzina 17:53:41WrWn
Jestes klientem #1WrWn
```

# Serwer TCP



# Kolejność wykonywania funkcji gniazdowych serwera TCP



# Serwer: Tworzenie gniazda

socket ► bind ► listen ► accept ► recv ► send ► close

Serwer wykorzystuje funkcję **socket** w taki sam sposób,  
jak klient – do tworzenia gniazda.

To samo tyczy się funkcji **read**, **write** oraz **close**.

# Serwer: Konfiguracja gniazda [1]

socket ► bind ► listen ► accept ► recv ► send ► close

```
int bind(int sock,  
        sockaddr *loc, int loclen);
```

Funkcja dowiązuje gniazdo do lokalnego adresu, określonego w *loc*. Wykonuje tzw. **otwarcie bierne**.

- *sock* – uchwyt gniazda (zwrócony przez `socket`)
- *loc* – wskaźnik na strukturę `sockaddr` przechowującą informację o protokole oraz lokalny adres na jaki będą łączyć się klienci.
- *loclen* – długość, w bajtach, struktury `sockaddr` dla danego protokołu.
- **Wynik:** 0, lub:
  - **SOCKET\_ERROR**, kod błędu z `WSAGetLastError` (Windows),
  - **-1**, kod błędu z `errno` (Unix)

Najczęstszy powód niepowodzenia funkcji `bind` to niezamknięcie gniazda nasłuchowego. Zdarza się to przy nagłym zamknięciu aplikacji serwera.



# Serwer: Konfiguracja gniazda [2]

socket ► bind ► listen ► accept ► recv ► send ► close

```
typedef struct sockaddr {  
    u_short sa_family;  
    CHAR sa_data[14];  
} SOCKADDR;
```

```
typedef struct sockaddr_in {  
    short sin_family;  
    unsigned short sin_port;  
    IN_ADDR sin_addr; ←  
    CHAR sin_zero[8];  
} SOCKADDR_IN, *PSOCKADDR_IN;
```

```
sockaddr_in service;  
service.sin_family = AF_INET;  
service.sin_addr.s_addr = INADDR_ANY;  
service.sin_port = htons(3370);
```

```
typedef struct in_addr {  
    union {  
        struct {  
            UCHAR s_b1, s_b2, s_b3,  
                s_b4;  
        } S_un_b;  
        struct {  
            USHORT s_w1, s_w2;  
        } S_un_w;  
        ULONG S_addr;  
    } S_un;  
} IN_ADDR, *PIN_ADDR, *LPIN_ADDR;
```

```
#define s_addr S_un.S_addr  
#define s_host S_un.S_un_b.s_b2  
#define s_net S_un.S_un_b.s_b1  
#define s_imp S_un.S_un_w.s_w2  
#define s_impno S_un.S_un_b.s_b4  
#define s_lh S_un.S_un_b.s_b3
```

# Serwer: Konfiguracja gniazda [3]

socket ► bind ► listen ► accept ► recv ► send ► close

```
sockaddr_in service;  
service.sin_family = AF_INET;  
service.sin_addr.s_addr = INADDR_ANY;  
service.sin_port = htons(3370);
```

```
#define INADDR_ANY      (ULONG)0x00000000  
#define INADDR_LOOPBACK 0x7f000001  
#define INADDR_BROADCAST (ULONG)0xffffffff
```

## Adres IP w postaci tekstowej:

```
.s_addr = inet_addr("127.0.0.1" )
```

Jeżeli serwer posiada trzy interfejsy:  
192.168.1.1; 10.1.0.21; 212.191.78.134  
to może nasłuchiwać na wszystkich  
[INADDR\_ANY] lub tylko na wybranym  
[inet\_addr("10.1.0.21" )]

```
#define AF_UNSPEC      0      // unspecified  
#define AF_UNIX       1      // local to host (pipes, portals)  
#define AF_INET       2      // internetwork: UDP, TCP, etc.  
#define AF_IMPLINK    3      // arpanet imp addresses  
#define AF_PUP        4      // pup protocols: e.g. BSP  
#define AF_CHAOS      5      // mit CHAOS protocols  
#define AF_NS         6      // XEROX NS protocols  
#define AF_IPX        AF_NS  // IPX protocols: IPX, SPX, etc.  
#define AF_ISO        7      // ISO protocols  
#define AF_OSI        AF_ISO // OSI is ISO  
#define AF_ECMA       8      // european computer manufacturers  
#define AF_DATAKIT    9      // datakit protocols  
#define AF_CCITT      10     // CCITT protocols, X.25 etc  
#define AF_SNA        11     // IBM SNA  
#define AF_DECnet     12     // DECnet  
#define AF_DLI        13     // Direct data link interface  
#define AF_LAT        14     // LAT  
#define AF_HYLINK     15     // NSC Hyperchannel  
#define AF_APPLETALK  16     // AppleTalk  
#define AF_NETBIOS    17     // NetBios-style addresses  
#define AF_VOICEVIEW  18     // VoiceView  
#define AF_FIREFOX    19     // Protocols from Firefox  
#define AF_UNKNOWN1   20     // Somebody is using this!  
#define AF_BAN        21     // Banyan  
#define AF_ATM        22     // Native ATM Services  
#define AF_INET6      23     // Internet Network Version 6  
#define AF_CLUSTER    24     // Microsoft Wolfpack  
#define AF_12844      25     // IEEE 1284.4 WG AF  
#define AF_IRDA       26     // IrDA  
#define AF_NETDES     28     // Network Designers OSI & gateway
```

# Serwer: Nasłuchiwanie

socket ► bind ► **listen** ► accept ► recv ► send ► close

```
int listen(int sock, int backlog);
```

Funkcja uruchamia tryb nasłuchu dla zadanego gniazda.

- *sock* – uchwyt gniazda (zwrócony przez **socket**)
- *backlog* – ilość połączeń oczekujących na odebranie funkcją **accept**,
- **Wynik: 0**, lub:
  - **SOCKET\_ERROR**, kod błędu z *WSAGetLastError* (Windows),
  - **-1**, kod błędu z *errno* (Unix)

# Serwer: Przyjmowanie połączenia

socket ► bind ► listen ► **accept** ► recv ► send ► close

```
int accept(int sock,  
           sockaddr *loc, int *locLen);
```

Funkcja uruchamia tryb nasłuchu dla zadanego gniazda.  
Wykonuje tzw. **otwarcie bierne**.

- *sock* – uchwyt gniazda (zwrócony przez **socket**)
- *loc* – wskaźnik na strukturę `sockaddr` przechowującą informację o protokole oraz zdalny adres z jakiego łączy się dany klient
- *locLen* – długość, w bajtach, struktury `sockaddr` dla danego protokołu,
- **Wynik:** uchwyt gniazda połączenia przychodzącego + *loc*, lub:
  - **INVALID\_SOCKET**, kod błędu z `WSAGetLastError` (Windows),
  - **-1**, kod błędu z `errno` (Unix)

```

int main(int argc, char* argv[])
{
    WSADATA data;
    int result, counter = 0;
    sockaddr_in service, remote;

    result = WSAStartup(MAKEWORD(2, 0), &data);
    assert(result == 0);

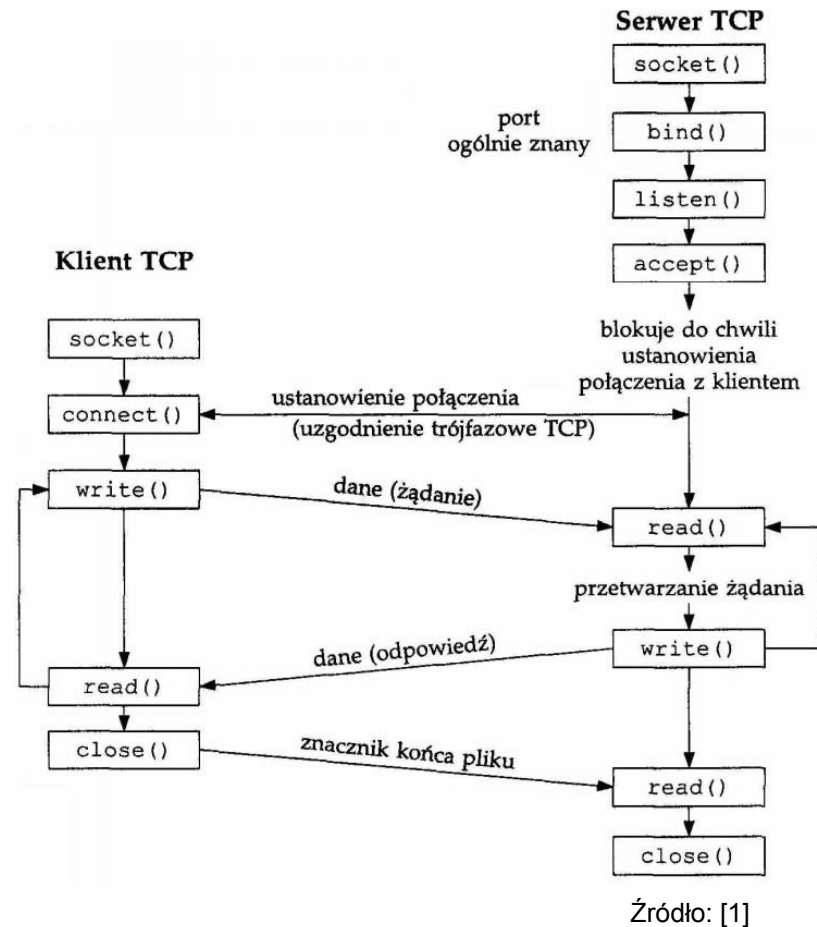
    SOCKET listen_socket = socket(AF_INET,
                                  SOCK_STREAM, IPPROTO_TCP);
    assert(listen_socket != INVALID_SOCKET);

    service.sin_family = AF_INET;
    service.sin_port = htons(3301);
    service.sin_addr.s_addr = INADDR_ANY;
    result = bind(listen_socket, (sockaddr*)&service,
                  sizeof(sockaddr_in));
    assert(result != SOCKET_ERROR);

    result = listen(listen_socket, 5);
    assert(result != SOCKET_ERROR);

    closesocket(listen_socket);
    return 0;
}

```



## Tutaj główna pętla programu serwera

```

closesocket(listen_socket);
return 0;
}

```

## Protokół serwera

```

Data 11/10/2010WrWn
Godzina 17:53:41WrWn
Jestes klientem #1WrWn

```

```

while(true)
{
    int size = sizeof(sockaddr_in);
    SOCKET client = accept(listen_socket,
        (sockaddr*)&remote, &size);
    printf("Polaczenie z %s:%d\n",
        inet_ntoa(remote.sin_addr),
        ntohs(remote.sin_port));
    assert(client != INVALID_SOCKET);
    char str[100];
    time_t curr_time;
    time(&curr_time);
    tm *t = gmtime(&curr_time);

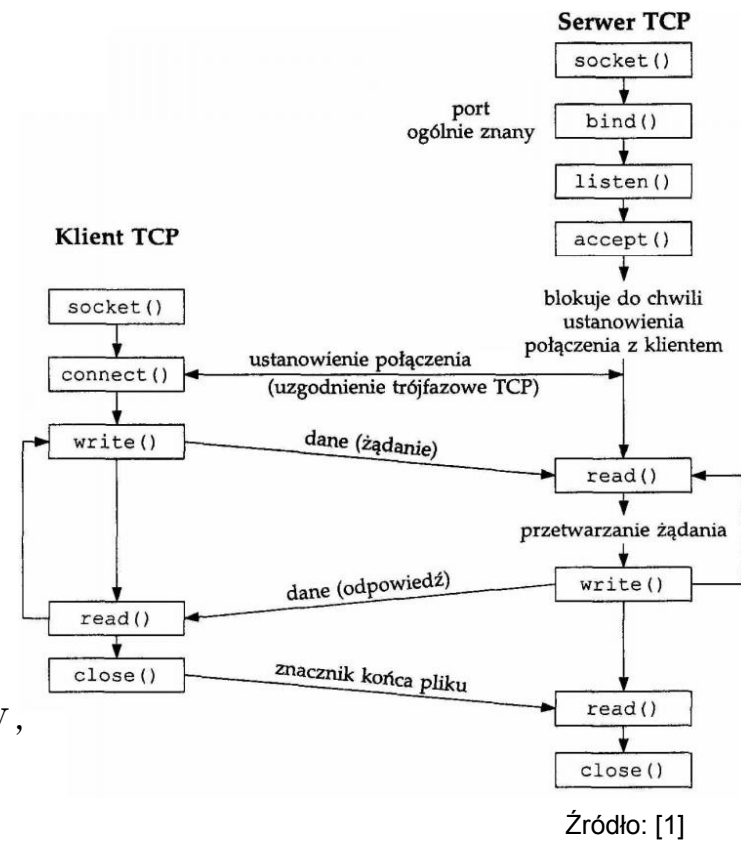
    sprintf(str, "Data %02d/%02d/%04dWrWn", t->tm_mday,
        t->tm_mon + 1, t->tm_year + 1900);
    send(client, str, strlen(str), 0);

    sprintf(str, "Godzina %02d:%02d:%02dWrWn", t->tm_hour,
        t->tm_min, t->tm_sec);
    send(client, str, strlen(str), 0);

    counter++;
    sprintf(str, "Jestes klientem #%dWrWn",
        counter);
    send(client, str, strlen(str), 0);

    closesocket(client);
}

```



```

Protokół serwera
Data 11/10/2010WrWn
Godzina 17:53:41WrWn
Jestes klientem #1WrWn

```



```
static void Main()
{
    Socket s = new Socket(AddressFamily.InterNetwork,
        SocketType.Stream, ProtocolType.Unspecified);
    s.Bind(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 3301));
    s.Listen(5);

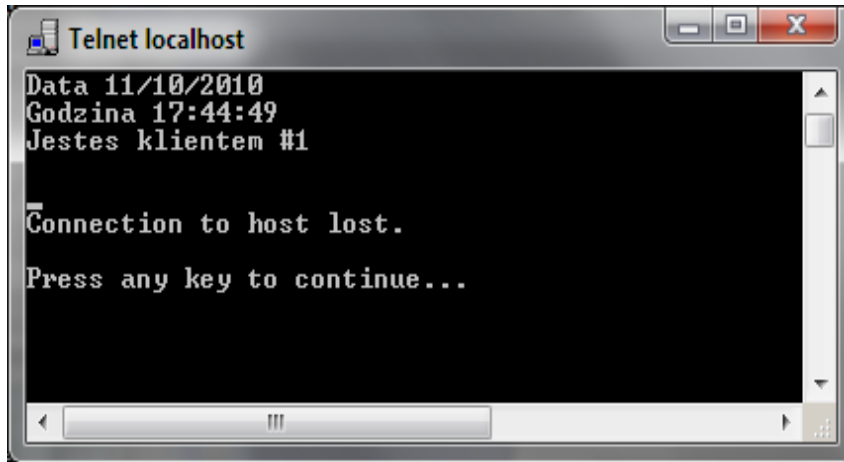
    int counter = 0;
    while (true)
    {
        Socket cli = s.Accept();
        Console.WriteLine("Polaczenie z {0}",
            cli.RemoteEndPoint.ToString());

        DateTime now = DateTime.Now;
        StringBuilder sb = new StringBuilder();
        sb.AppendLine(string.Format("Data: {0:00}/{1:00}/{2:0000}",
            now.Day, now.Month, now.Year));
        sb.AppendLine(string.Format("Czas: {0:00}:{1:00}:{2:00}",
            now.Hour, now.Minute, now.Second));
        sb.AppendLine(string.Format("Jestes klientem #{0}",
            counter));

        byte[] bufor = Encoding.ASCII.GetBytes(sb.ToString());
        cli.Send(bufor);
        cli.Close();
    }
}
```



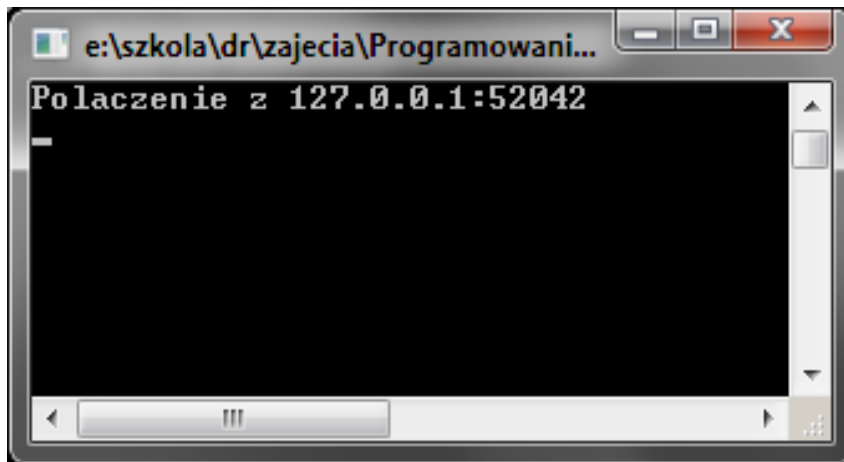
# Serwer: Testowanie



```
Telnet localhost
Data 11/10/2010
Godzina 17:44:49
Jestes klientem #1

Connection to host lost.
Press any key to continue...
```

1. Start-> Uruchom -> "telnet localhost 3301"
2. Start->Uruchom -> telnet -> "open localhost 3301"



```
e:\szkola\dr\zajecia\Programowani...
Polaczenie z 127.0.0.1:52042
```



# Kody błędów operacji gniazdowych

Funkcje gniazdowe sygnalizują wystąpienie sytuacji awaryjnej poprzez wartość zwracaną:

- `INVALID_SOCKET` (np. funkcja `accept`, `socket`)
- `SOCKET_ERROR` (np. funkcja `recv`, `send`)

Jest to jednak informacja o **zaistnieniu sytuacji awaryjnej**. Szczegóły awarii dostępne są w zależności od systemu...

▪

# Kody błędów operacji gniazdowych funkcji ogólnych i operujących na gniazdach

W przypadku klienta lub serwera, obsługującego tylko jedno połączenie (lub jedno połączenie na wątek) można skorzystać z:

- Windows: **int WSAGetLastError(void);**
- Linux: **int errno;** (z `errno.h`)

Dla funkcji gniazdowych, ale nieoperujących na gniazdach (np. `accept`), powyższe funkcje to jedne źródło informacji o błędzie.

# Kody błędów operacji gniazdowych

## ZWIAZANE Z KONKRETNYM GNIAZDEM

Struktura każdego gniazda zawiera pole `so_error`.  
W sytuacji awaryjnej jest ono ustawiane odpowiednim kodem błędu.

Jeśli tak, to:

- operacje **blokujące** zostają przerywane i zwracają `SOCKET_ERROR`;
- operacje **nieblokujące** nie rozpoczynają operacji asynchronicznej, kończą się natychmiast i zwracają `SOCKET_ERROR`

Pobranie kodu błędu łączy się z jego usunięciem ze zmiennej `so_error`.

# Kody błędów operacji gniazdowych

## ZWIAZANE Z KONKRETNYM GNIAZDEM

Aby odczytać błąd, ze zmiennej *so\_error* struktury gniazda, skojarzony z konkretnym gniazdem, należy odczytać **wartość opcji SO\_ERROR** z warstwy gniazda (**SOL\_SOCKET**):

```
static int getSocketError(SOCKET socket)
{
    int error_code = 0;
    socklen_t size = sizeof(socklen_t);
    getsockopt(socket, SOL_SOCKET, SO_ERROR,
                (char*)&error_code, &size);
    return error_code;
}
```

# Klient/Serwer:

## Operacje blokujące i nieblokujące [1]

Do czego można wykorzystać operacje **blokujące**?

- Krótki i przejrzysty kod,
- Idealne dla oddzielnego wątku,

A operacje **nieblokujące**?

- Serwery dla wielu połączeń jednocześnie,
- Realizacja timeout-ów,

# Klient/Serwer:

## Operacje blokujące i nieblokujące [1]

**Operacje blokujące (blocking)**: operacje gniazdowe wykonywane są synchronicznie. Funkcje kończą się po całkowitym wykonaniu operacji lub w przypadku błędu.

**Operacje nieblokujące (nonblocking)**: operacje gniazdowe wykonywane są asynchronicznie. Funkcje kończą się zawsze, t/j. po zleceniu podsystemowi gniazd danej operacji (np. `connect`, `send`, `recv`) lub w przypadku błędu.

# Operacje blokujące i nieblokujące dla `recv`

## Gniazdo blokujące

- `recv` czeka, aż w buforze odbiorczym gniazda będzie minimum bajtów do pobrania; minimum podawane jest w wywołaniu funkcji `recv` – parametr *buflen*;
- Funkcja zwraca ilość danych pobranych z bufora odbiorczego (*buflen*);
- W przypadku błędu gniazda, funkcja zwraca `SOCKET_ERROR` (-1);

## Gniazdo nieblokujące

- `recv` wczytuje tyle danych, ile zostało odebrano (nie mniej niż 1) i zwraca ilość wczytanych bajtów.
- W przypadku braku danych w buforze, `read` zwraca `SOCKET_ERROR` a kod błędu = **WSAEWOULDBLOCK/EWOULDBLOCK**.
- W przypadku zamknięcia połączenia lokalnie lub zdalnie, `recv` zwraca 0.
- Lub `SOCKET_ERROR` w przypadku innego błędu.

# Operacje blokujące i nieblokujące dla `send`

## Gniazdo blokujące

- `send` czeka, aż w buforze nadawczym będzie wystarczająco dużo miejsca na wysłanie *buflen* bajtów;
- Po czym (po zapisie do bufora nadawczego) zwraca informacje o wysłaniu *buflen* bajtów;
- lub `SOCKET_ERROR` w przypadku błędu (np. gdy nie ma wolnego

## Gniazdo nieblokujące

- `send` zapisuje do bufora nadawczego tyle, ile może (nie mniej niż 1) i zwraca ilość zapisanych bajtów.
- W przypadku braku miejsca w buforze, `send` zwraca `SOCKET_ERROR` a kod błędu = `WSAEWOULDBLOCK/EWOULDBLOCK`.



# Operacje blokujące i nieblokujące dla `connect`

## Gniazdo blokujące

- Funkcja zwraca 0, jeśli operacja się powiodła
- Lub `SOCKET_ERROR` jeśli połączenie nie zostało nawiązane.

## Gniazdo nieblokujące

- Funkcja rozpocznie proces nawiązywania połączenia i natychmiast powróci do programu;
- Sprawdzenie, czy połączenie zostało nawiązane możliwe jest dzięki funkcjom monitorowania operacji *asynchronicznych*: `select` (Win/Lin), `poll/epoll` (Lin), `WSAAsyncSelect`, `WSAEventSelect` (Win)

# Operacje blokujące i nieblokujące dla `accept`

## Gniazdo blokujące

- Funkcja oczekuje tak długo, aż zostanie nawiązane połączenie;
- i zwraca uchwyt gniazda (SOCKET)

## Gniazdo nieblokujące

- Jeśli w buforze odbiorczym gniazda nasłuchowego są oczekujące połączenia, to jedno (kolejka FIFO) zostaje sfinalizowane i funkcja zwraca uchwyt gniazda;
- Jeśli nie ma połączeń oczekujących, funkcja zwraca `SOCKET_ERROR/WSAEWOULDBLOCK`;
- lub `SOCKET_ERROR` w przypadku innego błędu.

# Operacje blokujące i nieblokujące

**Operacje blokujące (blocking)**: operacje gniazdowe wykonywane są synchronicznie. Funkcje kończą się po całkowitym wykonaniu operacji lub w przypadku błędu.

**Operacje nieblokujące (nonblocking)**: operacje gniazdowe wykonywane są asynchronicznie. Funkcje kończą się zawsze, t/j. po zleceniu podsystemowi gniazd danej operacji (np. `connect`, `send`, `recv`) lub w przypadku błędu.

# Klient/Serwer:

## Operacje blokujące i nieblokujące [3]

Po utworzeniu gniazdo jest domyślnie ustawione jako blokujące.

Kiedy przełączać gniazdo ze stanu blokowania do nieblokowania?

**Serwer/gniazdo nasłuchu:** po funkcji `listen`,

**Serwer/gniazdo klienta:** po funkcji `accept`,

**Klient:** po funkcji `connect`.

# Klient/Serwer:

## Operacje blokujące i nieblokujące [2]

```
bool SetBlocking(SOCKET socket, bool can_block)
{
#ifdef WIN32
    u_long flag = !can_block;
    return ioctlsocket(socket, FIONBIO, &flag) != SOCKET_ERROR;
#else // __linux__
    if (can_block)
    {
        int flags = fcntl(socket, F_GETFL, 0);
        return fcntl(socket, F_SETFL, flags & ~O_NONBLOCK) != -1;
    } else
        return fcntl(socket, F_SETFL, O_NONBLOCK) != -1;
#endif
}
```

# Synchronizacja operacji na gniazdach

Na czas wykonywania operacji gniazdowych **blokujących**, wątek/proces wykonujący jest wstrzymywany aż do jej **zakończenia** lub **wystąpienia błędu**.

Alternatywą są gniazda **nieblokujące** z **operacjami asynchronicznymi**.

Każda z takich operacji kończy się zmianą stanu gniazda.

**Można to wykorzystać poprzez funkcje monitorujące stan wielu gniazd jednocześnie.**

# Rodzina funkcji monitorujących operacje asynchroniczne

Windows:

- **select**
- **WSAPoll** (minimum Vista)
- **WSAAsyncSelect** (pomost między zdarzeniami gniazd a komunikatami GUI)
- **WSAEventSelect** (połączenie zdarzeń gniazd z obiektami synchronizującymi WSAEVENT) -> Oczekiwanie w WSAWaitForMultipleEvents

Linux:

- **select**
- **poll**
- **epoll**
- **/dev/poll**
- **/dev/epoll**

# SELECT

- Funkcja **select** należy do rodziny funkcji monitorujących zbiór deskryptorów WE/WY (tutaj – gniazd)
- Funkcja niezwykle przydatna do synchronizacji wielu operacji wykonywanych w tle (asynchronicznych)
- Funkcja wraca do programu użytkownika, gdy:
  - Minie **czas** przeznaczony na oczekiwania zdarzenia
  - **Zdarzenie**: jedno lub więcej gniazd jest **gotowych do odczytu** (dane nadeszły, są w buforze odbiorczym)
  - **Zdarzenie**: jedno lub więcej gniazd jest **gotowych do zapisu** (pojawiło się miejsce w buforze nadawczym)
  - **Zdarzenie**: jedno lub więcej gniazd weszło w **stan wyjątku**/awaryjny (ang. *exceptional condition pending*).



# SELECT - prototyp

```
int select(
    int nfds,                                // we
    fd_set* readfds,                        // we/wy
    fd_set* writefds,                      // we/wy
    fd_set* exceptfds,                    // we/wy
    const struct timeval* timeout);      // we
```

Parametry:

- ***nfds*** – ignorowana, parametr tylko dla zachowania kompatybilności ze standardem Berkeley.
- ***readfds*** – zbiór deskryptorów sprawdzanych pod kątem możliwości odczytu (obecności danych w buforze odbiorczym)
- ***writefds*** – j/w. tylko sprawdzanych pod kątem możliwości zapisu (czy jest wolne miejsce w buforze nadawczym?)
- ***exceptfds*** – deskryptory sprawdzane pod kątem pola błędów **so\_error**.
- ***timeout*** – struktura opisująca czas, jaki funkcja select ma oczekiwać na dowolne zdarzenie

# SELECT – wartość zwracana

```
int select(
    int nfds,                                // we
    fd_set* readfds,                        // we/wy
    fd_set* writefds,                       // we/wy
    fd_set* exceptfds,                     // we/wy
    const struct timeval* timeout);       // we
```

Wartość zwracana:

- **SOCKET\_ERROR (-1)** jeśli wystąpił błąd podczas oczekiwania na jakieś zdarzenie związane z monitorowanymi gniazdami;
- **0** jeśli minął czas oczekiwania określony w parametrze **timeout**;
- **≥ 1** jeśli pojawiło się zdarzenie dla któregoś z gniazd; wartość zwracana określa ilość gniazd w których nastąpiła zmiana stanu

# SELECT – parametr *nfds*

```
int select(
    int nfds,                                // we
    fd_set* readfds,                        // we/wy
    fd_set* writefds,                       // we/wy
    fd_set* exceptfds,                     // we/wy
    const struct timeval* timeout);       // we
```

## Parametry:

- ***nfds*** – ignorowany, parametr tylko dla zachowania kompatybilności ze standardem Berkeley. Jest to spowodowane inną konstrukcją struktury **fd\_set** w Windows Sockets. Domyślnie 0.
- W przypadku innych systemów (standard BSD) jest to zakres wartości uchwytów gniazd występujących w zbiorach **fd\_set**.
  - Jeśli **select** oczekuje na zdarzenia dla trzech uchwytów: {100, 234, 23}, to ***nfds* = 234 + 1**.

# SELECT – parametr *readfds*

```
int select(  
    int nfds,                                // we  
    fd_set* readfds,                         // we/wy  
    fd_set* writefds,                       // we/wy  
    fd_set* exceptfds,                      // we/wy  
    const struct timeval* timeout);         // we
```

Parametry:

- ***readfds*** – zbiór deskryptorów sprawdzanych pod kątem możliwości odczytu.
- Po zakończeniu ***select***, zbiór ***readfds*** będzie zawierał listę deskryptorów gniazd, które mają dane w buforach odbiorczych i z których można odczytywać

Przykład:

- a) ***select*** ma monitorować trzy gniazda, których buforzy odbiorcze są na starcie puste. ***select*** zakończy się **po czasie** w zmiennej ***timeout*** lub kiedy w buforze odbiorczym jednego z gniazd **pojawią się odebrane dane**.
- b) Jeśli już na starcie, w buforze odbiorczym choć jednego z gniazd, będą dane – funkcja ***select*** zakończy się **natychmiast**.

# SELECT – parametr *writefds*

```
int select(  
    int nfds,                                // we  
    fd_set* readfds,                         // we/wy  
    fd_set* writefds,                        // we/wy  
    fd_set* exceptfds,                       // we/wy  
    const struct timeval* timeout);         // we
```

Parametry:

- ***writefds*** – zbiór deskryptorów sprawdzanych pod kątem możliwości zapisu.
- Po zakończeniu **select**, zbiór ***writefds*** będzie zawierał listę deskryptorów gniazd, które mają wolne miejsce w buforach odbiorczych i można do nich zapisywać

Przykład:

- Założmy, że wielkość bufora nadawczego to 16kB.
- Funkcją **send** chcemy wysłać blok 128kB danych;
- **send** zapisze do bufora nadawczego tyle danych, ile jest w nim wolnego miejsca i wartość tę zwróci jako wynik funkcji. W tym czasie system kontynuuje proces wysyłania danych.
- Jeśli gniazdo to jest monitorowane funkcja **select** i jego deskryptor znajduje się w zbiorze ***writefds***, to **select** zareaguje na zwolnienie się miejsca w buforze nadawczym poprzez dodanie deskryptora do wyjściowej wersji zbioru ***writefds*** i zakończenie działania.

# SELECT – parametr *exceptfds*

```
int select(  
    int nfds,                                // we  
    fd_set* readfds,                         // we/wy  
    fd_set* writefds,                       // we/wy  
    fd_set* exceptfds,                      // we/wy  
    const struct timeval* timeout);         // we
```

Parametry:

- **exceptfds** – zawiera zbiór deskryptorów sprawdzanych pod kątem pola błędów **so\_error**.
- Po zakończeniu **select**, zbiór **exceptfds** będzie zawierał listę deskryptorów gniazd, które znajdują się w stanie błędu (pole **so\_error**)
- Deskryptory zwrócone w wyjściowej wersji **exceptfds** zawierają kod błędu, który można odczytać poprzez wywołanie **getsockopt SO\_ERROR**.

Przykład:

- Funkcja **select** monitoruje gniazdo klienta z uruchomioną operacją **connect** (**writefd** i **exceptfd**).
- Jeśli połączenie zostanie nawiązane, deskryptor trafi do zbioru **writefds** (gotowy do zapisu). Jeśli połączenie zostanie odrzucone, deskryptor trafi do zbioru **exceptfds**.

# SELECT – parametr *timeout*

```
int select(  
    int nfds,                                // we  
    fd_set* readfds,                        // we/wy  
    fd_set* writefds,                      // we/wy  
    fd_set* exceptfds,                    // we/wy  
    const struct timeval* timeout);        // we
```

## Parametry:

- ***timeout*** – struktura opisująca czas, jaki funkcja **select** będzie oczekiwała na dowolne zdarzenie związane ze zbiorami deskryptorów **read**, **write** i **except**.
- Jeśli czas zostanie przekroczony, funkcja **select** zwróci wartość **0**.
- System Windows Sockets **nie modyfikuje** struktury **timeout**, można ją zatem użyć przy ponownym wywołaniu funkcji **select**.
- Niektóre dystrybucje systemu Linux zapisują w **timeout** czas *pozostały* do zakończenia oczekiwania.
- Zatem dobrym zwyczajem jest inicjowanie struktury **timeout** przed każdym wywołaniem funkcji **select**.

## SELECT – parametr *timeout*

### Struktura `timeval`:

```
typedef struct timeval
{
    long tv_sec;
    long tv_usec;
} timeval;
```

### Pola:

- `tv_sec` – ilość czasu oczekiwania, w sekundach;
- `tv_usec` – ilość czasu oczekiwania, w mikrosekundach;

```
struct timeval czas;
czas.tv_sec = 3 * 60; // trzy minuty
czas.tv_usec = 0;
```



# SELECT – operacje na zbiorach deskryptorów

- Zbiory deskryptorów (*readfds*, *writefds*, *exceptfds*) opisywane są przez typ *fd\_set*.
- Ze względu na różnice między systemami operacyjnymi, zaleca się stosowanie odpowiednich funkcji operujących na tych zbiorach, a przystosowanych do konkretnego systemu operacyjnego: **FD\_CLR**, **FD\_ISSET**, **FD\_SET**, **FD\_ZERO**.
- Wersje tych funkcji mogą być bardzo nieoptymalne (np. w Windows). Warto znać ich zasadę działania i platformę. Dzięki temu można wkomponować we własny program obsługę typu *fd\_set* bez odwoływania się do nieoptymalnych odpowiedników **FD\_xxx**.

# SELECT – operacje na zbiorach deskryptorów

void **FD\_ZERO**(fd\_set \*set);

- **Zeruje zbiór** set, kasując z niego wszystkie deskryptory.

void **FD\_CLR**(int fd, fd\_set \*set);

- **Usuwa** deskryptor fd ze zbioru set.

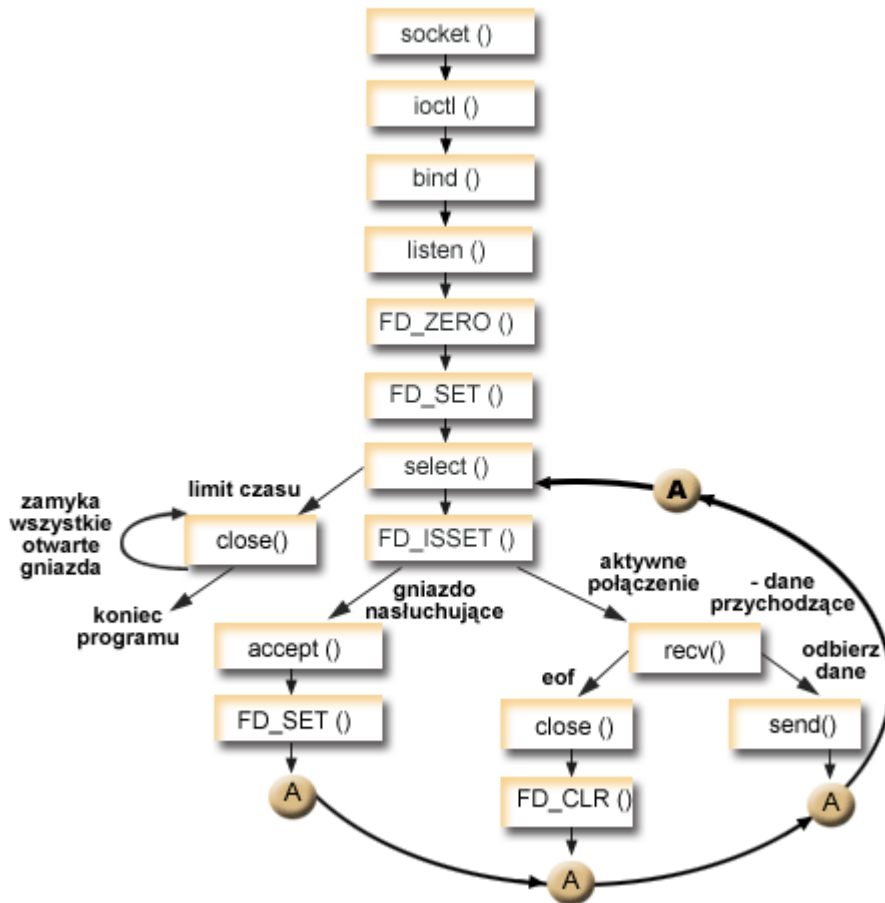
void **FD\_SET**(int fd, fd\_set \*set);

- **Dodaje** deskryptor fd do zbioru set.

int **FD\_ISSET**(int fd, fd\_set \*set);

- **Sprawdza**, czy deskryptor fd jest obecny w zbiorze set.

# SELECT – przykład 1



Źródło: <http://publib.boulder.ibm.com>

- Limit czasu dla **select** = 30sek;
- Serwer czeka max 30 sekund na dane przychodzące.
- Gniazda, które nie otrzymały danych w tym czasie są zamykane;

# SELECT – przykład 2

1. Zainicjuj Windows Sockets
2. Zainicjuj gniazdo serwera

while(true)

{

1. Przygotuj zbiór deskryptorów **readfds** dla funkcji **select**
2. Wywołaj funkcję **select** i ewaluuj wartość zwracaną
3. W zbiorze wynikowym **readfds** jest deskryptor gniazda nasłuchowego serwera (sserver); sprawdzić, czy nie oczekują na nim nadchodzące połączenia
4. Dla wszystkich deskryptorów w wynikowym zbiorze **readfds**:
  1. Jeśli brak danych przychodzących, to przejdź do następnego deskryptora
  2. Wykonaj funkcję **recv** i sprawdź wartość zwracaną (czy nie ma błędu/zamkniętego połączenia)
  3. Odeślij odebrane dane do nadawcy

}

# SELECT – przykład 2

## inicjowanie gniazda serwera

```
char buffer[1024];  
SOCKET sserver;  
fd_set clean_readfds, readfds;  
int maxfd;  
  
//sserver = ???  
  
FD_ZERO(&clean_readfds);  
FD_SET(sserver, &clean_readfds);  
maxfd = sserver;
```

# SELECT – przykład 2

## przygotowania zbioru deskryptorów

```
memcpy(&readfds, &clean_readfds, sizeof(readfds));
```

```
timeval tv;
```

```
tv.tv_sec = 30; // 30 sekund
```

```
tv.tv_usec = 0;
```

# SELECT – przykład 2

## Wywołanie select i ewaluacja wyniku

```
int result = select(maxfd + 1, &readfds,  
    NULL, NULL, &tv);  
  
if (result == 0)  
    continue;  
  
if (result < 0) {  
    printf("Blad select(): %d\n",  
        WSAGetLastError());  
    exit(1);  
}
```

# SELECT – przykład 2

## sprawdzenie i odebranie połączeń przychodzących

```
if (FD_ISSET(sserver, &readfds)) {
    sockaddr_in addr;
    int len = sizeof(addr);
    SOCKET scli = accept(sserver, (sockaddr*)&addr, &len);
    if (scli < 0) {
        printf("Blad ACCEPT: %f\n", WSAGetLastError());
    }
    else {
        FD_SET(scli, &clean_readfds);
        maxfd = (scli > maxfd) ? scli : maxfd;
    }
    FD_CLR(sserver, &readfds);
}
```



# SELECT – przykład 2

## sprawdzenie, czy są dane przychodzące

```
for (int isocket = 0; isocket < maxfd + 1;  
isocket++) {
```

.....

Kod wykonywany dla wszystkich deskryptorów w zbiorze *readfds*:

```
if (!FD_ISSET(isocket, &readfds))  
    continue;
```

## SELECT – przykład 2

### odebranie danych z bufora odbiorczego i ocena wyniku funkcji recv

```
int recv_count = recv(isocket, buffer,
                      sizeof(buffer) - 1, 0);
if (recv_count == 0) {
    closesocket(isocket);
    FD_CLR(isocket, &clean_readfds);
    continue;
}
if (recv_count == SOCKET_ERROR) {
    printf("Blad RECV: %d\n", WSAGetLastError());
    FD_CLR(isocket, &clean_readfds);
    closesocket(isocket);
    continue;
}
```

# SELECT – przykład 2

## odesłanie odebranych danych

```
buffer[recv_count] = 0;
printf("Odsyłanie: %s\n", buffer);
int sent = 0;
while(recv_count - sent > 0) {
    int bytes_sent = send(isocket, buffer + sent,
                          recv_count - sent, 0);
    if (bytes_sent == SOCKET_ERROR) {
        closesocket(isocket);
        FD_CLR(isocket, &clean_readfds);
    }
    sent += bytes_sent;
}
```

**Dziękuję za uwagę!**