14. Komunikacja sieciowa

Wstęp

Możliwość komunikacji sieciowej z wykorzystaniem różnych protokołów transmisji stanowi jedno z podstawowych wymagań stawianych przed współczesnymi systemami operacyjnymi. Bezwzględnie wymaganym minimum stało się już obsługiwanie połączeń w sieci Internet z wykorzystaniem rodziny protokołów TCP/IP.

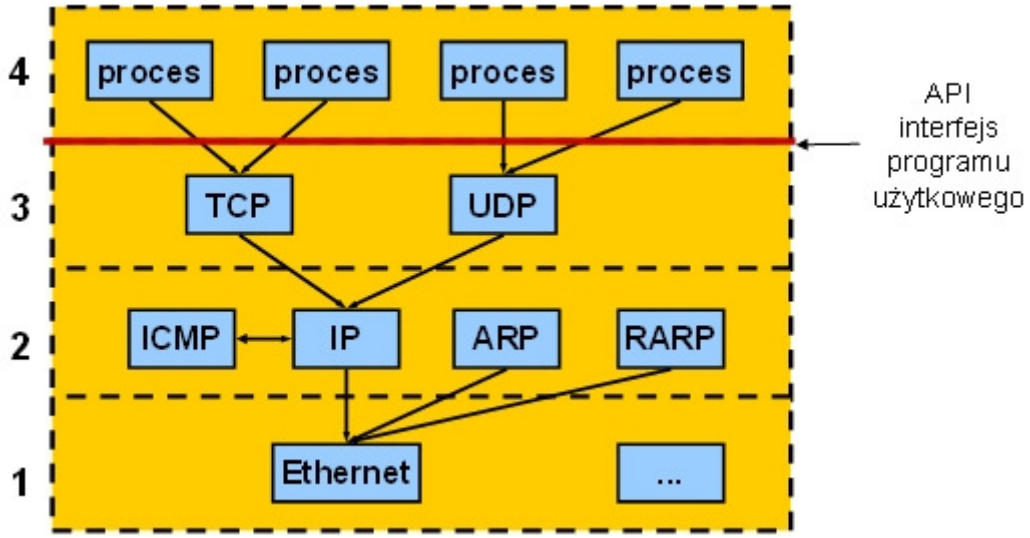
W wykładzie 14 omawiamy podstawowe aspekty modelu komunikacji sieciowej procesów. Przedstawiamy interfejs gniazd BSD, najpopularniejszy interfejs programu użytkowego API stworzony dla systemu Unix, oraz jego implementację w systemie Linux. Opisujemy również podstawowe funkcje systemowe interfejsu gniazd.

14.1. Model komunikacji sieciowej

Międzynarodowa Organizacja Normalizacyjna ISO opracowała model warstwowy połączenia systemów otwartych zwany modelem OSI, który określa normę komunikacji sieciowej komputerów. Model OSI składa się z 7 warstw i definiuje funkcje każdej warstwy, protokół komunikacji z warstwą i interfejsy pomiędzy warstwami. W większości istniejących rodzin protokołów zastosowano jednak uproszczony czterowarstwowy model komunikacji sieciowej.

Protokoły transmisji

Każda warstwa w modelu musi mieć zdefiniowany protokół transmisji danych. Powstają w ten sposób rodziny grupujące komplet protokołów dla wszystkich warstw. Najpowszechniej wykorzystywaną obecnie rodziną protokołów jest rodzina TCP/IP w sieci Internet (Rys. 14.1). Szczegółowe omówienie najpopularniejszych rodzin protokołów można znaleźć w literaturze [1]. Ważną cechą protokołów jest tryb komunikacji: **połączeniowy** lub **bezpołączeniowy**. W trybie połączeniowym procesy muszą ustanowić logiczne połączenie, aby rozpocząć komunikację. W trybie bezpołączeniowym (datagramowym) nie nawiązują połączenia, tylko wysyłają do siebie niezależne komunikaty zwane datagramami.



Rys. 14.1 Rodzina protokołów TCP/IP

Asocjacje

Połączenie w sieci ustanawiane jest między dwoma procesami działającymi na dwóch różnych komputerach.
Każde połączenie może być opisane przez pięcioelementowy zbiór parametrów zwany **asocjacją**:

1. protokół,
2. adres lokalny,
3. proces lokalny,
4. adres zdalny,
5. proces zdalny.

Wszystkie elementy asocjacji muszą zostać określone zanim procesy zaczną się komunikować.

Model komunikacji procesów klient - serwer

Komunikacja procesów w sieci komputerowej oparta jest zwykle na modelu klient - serwer. Relacja między procesami klienta i serwera jest asymetryczna, co wynika z różnych zadań tych procesów.

Serwer uruchamiany jest zazwyczaj jako pierwszy i podejmuje następujące działania:

1. otwiera kanał komunikacji,
2. informuje system operacyjny o gotowości odbierania zleceń od klientów pod ustalonym, ogólnie znanym adresem,
3. oczekuje na zlecenia klientów,
4. odbiera i przetwarza zlecenie w sposób uzależniony od rodzaju serwera, a następnie wysyła odpowiedź do klienta,
5. powraca do oczekiwania na kolejne zlecenia.

Klient postępuje w następujący sposób:

1. otwiera kanał komunikacji,
2. nawiązuje połączenie z serwerem pod ogólnie znanym adresem (tylko w protokole połączeniowym),
3. wysyła zlecenie na ogólnie znany adres serwera,
4. odbiera odpowiedź od serwera,
5. powtarza dwie powyższe czynności,
6. zamyka kanał komunikacyjny i kończy działanie.

Rozróżnia się dwa rodzaje serwerów:

1. iteracyjny - bezpośrednio obsługuje nadchodzące zlecenia klientów,
2. współbieżny - tworzy nowy proces potomny do obsługi każdego kolejnego zlecenia.

Serwery iteracyjne stosuje się najczęściej wtedy, gdy wiadomo z góry ile czasu zajmuje przetwarzanie każdego zlecenia. Jeśli czas obsługi pojedyńczego zlecenia klienta jest stosunkowo krótki, to serwer może obsługiwać je sekwencyjnie. Serwery iteracyjne wykorzystują zwykle protokoły bezpołączeniowe.

Serwer współbieżny znajduje zastosowanie w sytuacjach, gdy czasy obsługi nie są określone i mogą być długie. Proces serwera tworzy wtedy proces potomny i powierza mu obsługę bieżącego zlecenia, a sam oczekuje na kolejne połączenia.

Interfejsy programu użytkowego w systemach UNIX i Linux

Zgodnie z modelem komunikacji sieciowej, procesy użytkowników działają w najwyżej położonej warstwie - warstwie procesu. Możliwość korzystania z protokołów komunikacyjnych niższych warstw zapewnia specjalny interfejs programowy pomiędzy warstwą transportową i warstwą procesu. Nosi on nazwę interfejsu programu użytkowego (ang. Application Program Interface - API) lub interfejsu programowego warstwy transportowej.

Powstały dwa takie interfejsy dla systemu Unix:

1. interfejs gniazd BSD (ang. BSD sockets) stworzony dla systemu BSD Unix,
2. interfejs warstwy transportowej Systemu V (ang. Transfer Layer Interface - TLI) stworzony dla systemu Unix SVR3.

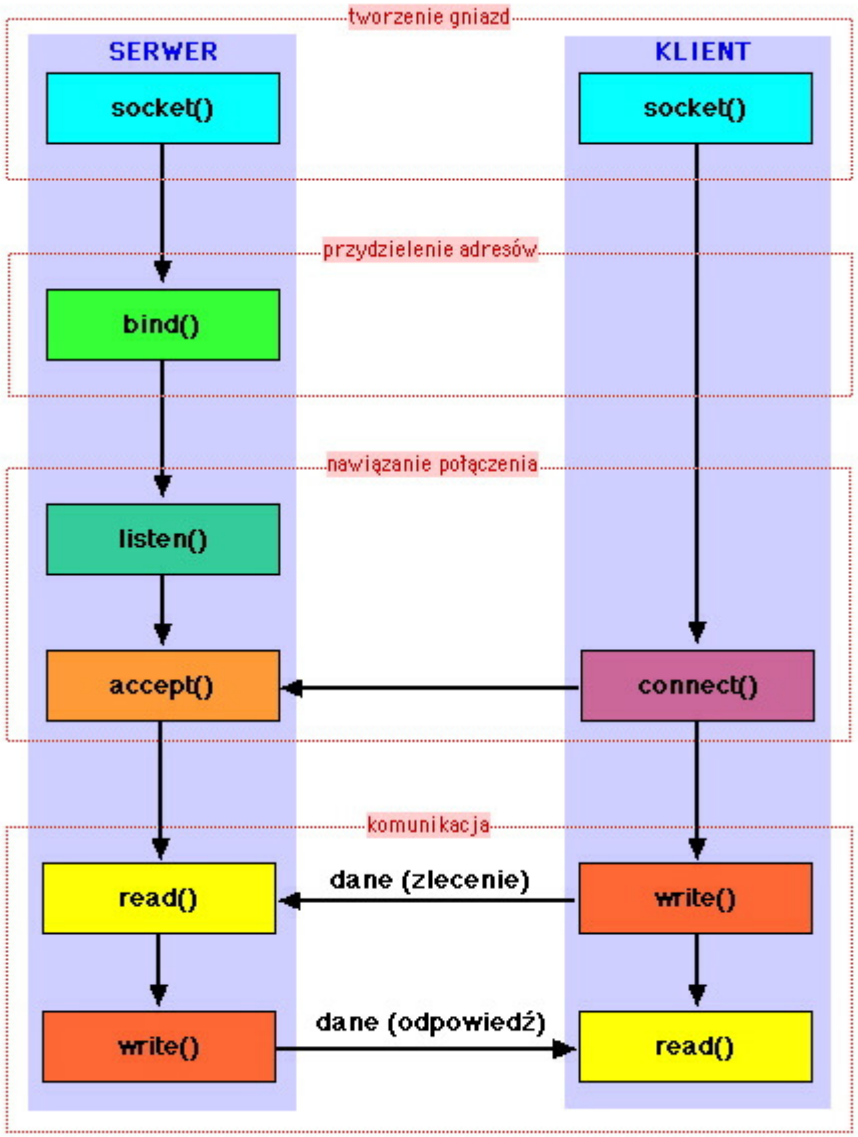
Obecnie wiele systemów uniksowych wykorzystuje obydwa interfejsy. Większą popularność zdobył jednak interfejs gniazd BSD. W systemie Linux zaimplementowano wyłącznie ten interfejs.

14.2. Gniazda BSD

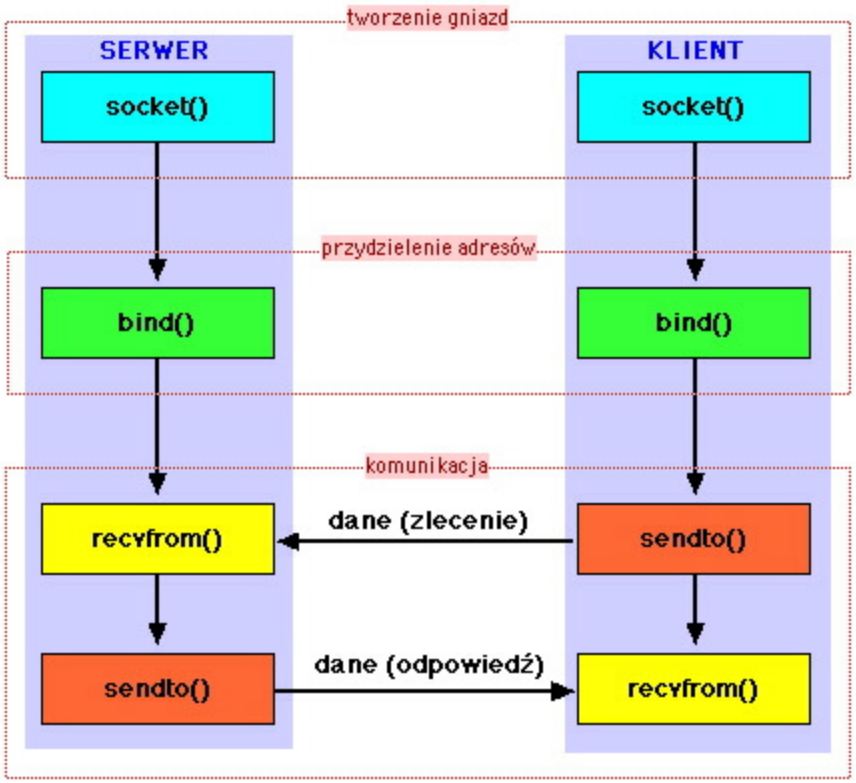
Gniazda BSD umożliwiają komunikację sieciową z wykorzystaniem różnych rodzin protokołów np. TCP/IP, IPX, AppleTalk, AX25 i wielu innych. Rodzinę protokołów komunikacyjnych określa się również jako dziedzinę komunikacji. Zdecydowanie najpopularniejsza jest obecnie komunikacja w dziedzinie Internetu, wykorzystująca protokoły TCP/IP. Gniazda dają również możliwość komunikacji międzyprocesowej w dziedzinie UNIX-a, czyli wewnątrz jednego systemu operacyjnego z wykorzystaniem jego wewnętrznych protokołów.

Scenariusze transmisji

Funkcje systemowe interfejsu gniazd pozwalają zrealizować transmisję danych w protokole połączeniowym lub bezpołączeniowym. Na rysunkach 14.2 i 14.3 przedstawiono typowe scenariusze takich transmisji.



Rys. 14.2 Typowy scenariusz transmisji połączeniowej



Rys. 14.3 Typowy scenariusz transmisji bezpołączeniowej

Asocjacje

Aby dwa procesy mogły się komunikować w sieci, muszą zostać określone wszystkie elementy asocjacji. Tablica 14.1 pokazuje określanie tych elementów przez funkcje interfejsu gniazd.

Tablica 14.1 Określanie elementów asocjacji

	Protokół	Adres lokalny, proces lokalny	Adres zdalny, proces zdalny
Serwer połączeniowy	socket()	bind()	listen(), accept()
Klient połączeniowy	socket()	connect()	
Serwer bezpołączeniowy	socket()	bind()	recvfrom()
Klient bezpołączeniowy	socket()	bind()	sendto()

Adresy gniazd

System definiuje ogólną postać adresu gniazda w następujący sposób:

```
struct sockaddr {
        u_short sa_family; - rodzina adresów: AF_xxx
        char sa_data[14];  - adres
}
```

Zestawienie stałych symbolicznych określających rodziny adresów i odpowiadające im rodziny protokołów zawiera tablica 14.2. Stałe z przedrostkami AF_ i PF_ można stosować zamiennie.

Tablica 14.2 Wybrane rodziny adresów i protokołów

Rodzina adresów	Rodzina protokołów	Opis
AF_UNIX	PF_UNIX	Protokoły wewnętrzne systemu UNIX
AF_INET	PF_INET	Protokoły Internetu TCP/IP
AF_IPX	PF_IPX	Protokoły IPX systemu Novell
AF_APPLETALK	PF_APPLETALK	Protokoły Appletalk
AF_AX25	PF_AX25	Protokoły radioamatorskie AX25

W dalszej części niniejszej lekcji będziemy rozważać tylko dwie rodziny protokołów: dziedzinę UNIX-a i dziedzinę Internetu. Każda z tych rodzin wymaga innego sposobu adresowania.

Adres w dziedzinie Internetu ma następującą postać:

```
struct sockaddr_in {
        short sin_family;          - rodzina: AF_INET
        u_short sin_port;          - 16-bitowy numer portu
        struct in_addr sin_addr;   - struktura zawierająca 32-bitowy adres internetowy
}

struct in_addr {
        u_long s_addr; - 32-bitowy adres internetowy
}
```

32-bitowy adres internetowy identyfikuje sieć i stację (komputer) w sieci, zaś 16-bitowy numer portu identyfikuje proces w stacji, komunikujący się przez gniazdo.

Adresy internetowe są zwykle przedstawiane w notacji kropkowo-dziesiętnej, podczas gdy protokoły TCP/IP posługują się liczbami 32-bitowymi. System Linux dostarcza zestaw funkcji bibliotecznych dokonujących odpowiednich konwersji adresów.

Funkcje biblioteczne **inet_aton**() i **inet_addr**() przekształcają adresy internetowe podane w notacji kropkowo-dziesiętnej na liczby 32-bitowe.

```
int inet_aton(const char *cp, struct in_addr *inp);

unsigned long int inet_addr(const char *cp);
```

gdzie:

cp - adres internetowy w notacji kropkowo-dziesiętnej,

inp - wskaźnik do struktury in_addr przeznaczonej na 32-bitowy adres internetowy.

Funkcja **inet_aton**() zapisuje przekształcony 32-bitowy adres w strukturze **in_addr**, podczas gdy funkcja **inet_addr**() zwraca tylko liczbę 32-bitową.

Funkcja **inet_ntoa**() przekształca 32-bitowy adres internetowy na ciąg znaków w notacji kropkowo-dziesiętnej.

```
char *inet_ntoa(struct in_addr in);
```

Numery portów są 16-bitowymi liczbami z zakresu od 0 do 65535. Część portów jest zastrzeżona dla procesów działających z uprawnieniami administratora. Początkowe numery przeznaczone są dla typowych usług internetowych, takich jak echo, finger, telnet czy ftp i określane jako ogólnie znane numery portów.

Tablica 10.3 Przyporządkowanie portów

Porty	Numery
porty zastrzeżone	0 - 1023
porty automatycznie przydzielane przez system	1024 - 5000
porty przydzielane dowolnie	5000 - 65535

Sposób przechowywania liczb wielobajtowych jest zależny od platformy sprzętowej. Stacje komunikujące się w sieci mogą stosować różną kolejność przechowywania bajtów liczb całkowitych. Protokoły sieciowe wymagają podawania liczb z zachowaniem sieciowej kolejności bajtów, w której starszy bajt przechowywany jest wczesniej (pod niższym adresem w pamięci). Dotyczy to zarówno adresów, jak i numerów portów. System dostarcza cztery funkcje biblioteczne do przekształcania kolejności bajtów:

```
unsigned long int htonl(unsigned long int hostlong);

unsigned short int htons(unsigned short int hostshort);

unsigned long int ntohl(unsigned long int netlong);

unsigned short int ntohs(unsigned short int netshort);
```

Funkcje **htonl**() i **htons**() przekształcają liczby całkowite, odpowiednio 4-bajtowe i 2-bajtowe, z kolejności bajtów stacji na kolejność sieciową. Funkcje **ntohl**() i **ntohs**() dokonują przekształceń odwrotnych.

Adresy w dziedzinie Unix-a mają odmienną postać:

```
struct sockaddr_un {
        u_short sun_family; - rodzina: AF_UNIX
        char sun_path[108]; - nazwa ścieżkowa pliku
}
```

Jako adres wykorzystywana jest tu nazwa ścieżkowa pliku, który jest tworzony podczas związywania adresu z gniazdem

14.3. Interfejs funkcji systemowych

Tworzenie gniazda

Funkcja **socket**() tworzy nowe gniazdo z ustalonym protokołem transmisji i zwraca jego deskryptor.

```
int socket(int domain, int type, int protocol);
```

gdzie:

domain - dziedzina komunikacji określana przez rodzinę adresów lub rodzinę protokołów,

type - typ gniazda,

protocol - protokół transmisji.

Domenę komunikacji **domain** specyfikuje się przez podanie jednej ze stałych opisanych w tablicy 10.xx i określających rodzinę adresów lub rodzinę protokołów. Stałe z przedrostkami AF_ i PF_ mogą być tu używane zamiennie.

Typ gniazda **type** określa jedna ze stałych podanych poniżej:

SOCK_STREAM - gniazdo strumieniowe wykorzystujące protokół połączeniowy (np. TCP),

SOCK_DGRAM - gniazdo datagramowe wykorzystujące protokół bezpołączeniowy (np. UDP),

SOCK_RAW - gniazdo surowe wykorzystujące bezpośrednio protokół warstwy sieciowej (np. IP),

SOCK_SEQPACKET - gniazdo pakietów uporządkowanych,

SOCK_RDM - gniazdo komunikatów niezawodnie doręczanych.

Dla dziedziny UNIX-a dostępne są tylko typy SOCK_STREAM i SOCK_DGRAM.

Trzeci argument funkcji wybiera konkretny protokół z rodziny, przeznaczony dla podanego typu gniazda. W większości przypadków w grę wchodzi tylko jeden protokół, który jądro systemu jest w stanie jednoznacznie wyznaczyć na podstawie wartości pozostałych dwóch argumentów. Dlatego argument **protocol** przyjmuje zazwyczaj wartość 0.

Nie wszystkie kombinacje rodzin protokołów i typów gniazd są dozwolone, ponieważ niektóre typy nie są zaimplementowane dla niektórych rodzin.

Dla dziedziny UNIX-a dostępna jest funkcja **socketpair**(), która tworzy dwa połączone ze sobą gniazda i zwraca ich deskryptory. Zaraz po utworzeniu jądro automatycznie przydziela adresy (nazwy) nowym gniazdom, przez co są od razu gotowe do użycia.

```
int socketpair(int d, int type, int protocol, int sv[2]);
```

gdzie:

d - dziedzina komunikacji,

type - typ gniazda,

protocol - protokół transmisji,

sv - tablica deskryptorów utworzonych gniazd.

Otrzymuje się w ten sposób dwukierunkowy kanał komunikacyjny pomiędzy procesami, czyli łącze strumieniowe. Ponieważ gniazda nie są jawnie nazywane przez proces, więc mogą z nich korzystać tylko procesy spokrewnione, które odziedziczyły deskryptory po procesie tworzącym. Często stosuje się określenie: gniazda nienazwane dziedziny UNIXa. Widać tu wyraźną analogię do łączy nienazwanych tworzonych funkcją **pipe**(). Łącza nienazwane zapewniają tylko jednokierunkową komunikację, podczas gdy łącza strumieniowe dają możliwość komunikacji w obydwu kierunkach.

Przydzielanie adresu

Każde gniazdo musi mieć przydzielony lokalny adres zanim będzie mogło być użyte do komunikacji. Operację tę określa się również jako związywanie adresu z gniazdem lub nazywanie gniazda. Ostatnie określenie ma szczególne uzasadnienie w przypadku gniazd w dziedzinie UNIX-a, dla których adresami są nazwy ścieżkowe plików. Pliki te są tworzone przez jądro podczas nazywania gniazda. Adres może być przydzielony jawnie za pomocą funkcji **bind**() lub automatycznie przez jądro systemu.

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

gdzie:

sockfd - deskryptor otwartego gniazda,

my_addr - wskaźnik do struktury zawierającej adres gniazda,

addrlen - długość adresu.

W wyniku działania funkcji gniazdo zostaje nazwane. Jak pokazują typowe scenariusze transmisji z rys. 14.1 i 14.2, jawne nazwanie gniazda muszą zrealizować procesy:

- serwera i klienta transmisji bezpołączeniowej,
- serwera transmisji połączeniowej.

Serwer w obydwu przypadkach musi ogłosić w systemie swój ogólnie znany adres,

Klient bezpołączeniowy musi się upewnić, że wysyłane przez niego datagramy zostaną opatrzone właściwym adresem zwrotnym. aby serwer wiedział dokąd wysyłać odpowiedzi.

Również klient połączeniowy może użyć funkcji **bind**(), ale nie jest to konieczne. Adres lokalnego gniazda klienta połączeniowego zostaje przydzielony przez jądro systemu przy próbie nawiązania połączenia z serwerem.

Ustanawianie połączenia

Serwer protokołu połączeniowego wywołuje funkcję **listen**(), aby zgłosić w systemie gotowość przyjmowania połączeń i ustalić jednocześnie maksymalną liczbę połączeń oczekujących na obsłużenie.

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

gdzie:

s - deskryptor gniazda,

backlog - limit długości kolejki oczekujących połączeń.

W większości systemów maksymalna dopuszczalna długość kolejki wynosi 5. Próby połączenia przekraczające tę liczbę będą odrzucane przez system.

Serwer akceptuje oczekujące połączenia za pomocą funkcji **accept**().

```
int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

gdzie:

s - deskryptor gniazda,

addr - wskaźnik do struktury adresowej przeznaczonej na odczytany adres gniazda klienta,

addrlen - długość adresu.

Funkcja pobiera pierwsze zgłoszenie z kolejki oczekujących połączeń i tworzy nowe gniazdo o tych samych własnościach, co stare gniazdo. Funkcja zwraca nowy deskryptor do utworzonego gniazdo, które jest przeznaczone do obsługi połączenia. Dzięki temu serwer może nadal przyjmować połączenia korzystając ze starego gniazda. Jeżeli kolejka połączeń jest pusta, to funkcja blokuje proces do momentu nawiązania nowego połączenia.

Funkcja systemowa **connect**() umożliwia klientom nawiązanie połączenia z serwerem.

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t
addrlen);
```

gdzie:

sockfd - deskryptor gniazda,

serv_addr - wskaźnik do struktury zawierającej adres gniazda serwera,

addrlen - długość adresu.

W protokole połączeniowym funkcja nawiązuje połączenie z gniazdem strumieniowym serwera. Proces klienta jest blokowany do momentu ustanowienia połączenia. Klient połączeniowy może zrealizować tylko jedno pomyślne połączenie.

Funkcja może być również użyta przez klienta protokołu bezpołączeniowego w celu zapamiętania adresu serwera. Połączenie między gniazdami nie zostaje ustanowione. Klient informuje tylko system operacyjny, że wszystkie datagramy wysyłane przez gniazdo mają być dostarczone pod ten adres i mogą być odbierane wyłącznie spod tego adresu. Dzięki temu proces klienta nie musi określać adresu docelowego dla każdego wysyłanego datagramu. Klient protokołu bezpołączeniowego może używać funkcji **connect**() wielokrotnie w celu zapamiętania kolejnych adresów.

Przesyłanie danych

Sposób przesyłania danych jest uzależniony od typu gniazd. Gniazda strumieniowe, wykorzystywane w transmisji połączeniowej, mogą być dostępne za pomocą typowych funkcji systemowych interfejsu plików. Po ustanowieniu połączenia, obydwa procesy uczestniczące w komunikacji znają już swoje adresy i mogą korzystać z funkcji **read**() i **write**() lub **recv**() i **send**() do odbierania i wysyłania danych przez gniazdo.

Klient protokołu bezpołączeniowego, który zapisał adres serwera funkcją **connect**(), może również używać wspomnianych funkcji do wysyłania i odbierania datagramów. W pozostałych przypadkach, przesłanie datagramu wymaga podania adresu odbiorcy, co umożliwia funkcja **sendto**().

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

int sendto(int s, const void *msg, size_t len, int flags, const struct
sockaddr *to, socklen_t tolen);
```

gdzie:

s - deskryptor gniazda,

msg - wskaźnik do bufora zawierającego wiadomość do wysłania do gniazda,

len - długość wiadomości,

flags - flagi,

to - wskaźnik do struktury zawierającej adres gniazda odbiorcy,

tolen - długość adresu.

Odbieranie datagramów wraz z adresem nadawcy umożliwia funkcja systemowa **recvfrom**().

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

int recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr
*from, socklen_t *fromlen);
```

gdzie:

s - deskryptor gniazda,

buf - wskaźnik do bufora przeznaczonego na odczytaną z gniazda wiadomość,

len - długość odczytanej wiadomości,

flags - flagi,

from - wskaźnik do struktury przeznaczonej na adres gniazda nadawcy,

fromlen - długość adresu.

Likwidacja połączenia i zamykanie gniazda

Do likwidacji połączenia i zamykania gniazda służy typowa funkcja interfejsu plików **close**(). Jeżeli protokół związany z gniazdem zapewnia niezawodne doręczanie danych (np. protokół TCP), to jądro systemu próbuje jeszcze wysłać wszystkie dane znajdujące się w kolejce.

Większe możliwości daje funkcja **shutdown**(), która pozwala zlikwidować połączenie całkowicie lub częściowo.

```
int shutdown(int s, int how);
```

gdzie:

s - deskryptor gniazda,

how - sposób likwidacji połączenia.

Argument **how** decyduje o sposobie likwidacji połączenia:

how = 0 - zabrania dalszego przyjmowania danych,

how = 1 - zabrania dalszego wysyłania danych,

how = 2 - zabrania zarówno przyjmowania jak i wysyłania danych.

Pobieranie adresów

Funkcja **getsockname**() umożliwia pobranie adresu związanego z lokalnym gniazdem.

```
int getsockname(int s, struct sockaddr *name, socklen_t *namelen);
```

gdzie:

s - deskryptor gniazda,

name - wskaźnik do struktury adresowej przeznaczonej na adres gniazda lokalnego,

namelen - długość adresu.

Funkcja **getpeername**() pobiera adres związany ze zdalnym gniazdem połączonego partnera komunikacji.

```
int getpeername(int s, struct sockaddr *name, socklen_t *namelen);
```

gdzie:

s - deskryptor gniazda,

name - wskaźnik do struktury adresowej przeznaczonej na adres gniazda zdalnego,

namelen - długość adresu.

14.4. Przykłady komunikacji klient-serwer

Poniżej prezentujemy implementacje dwóch schematów komunikacji z wykorzystaniem gniazd. Pierwszy przykład pokazuje realizację programy serwera i klienta wykorzystujące protokół strumieniowy w dziedzinie Uniksa. W przykładzie drugim prezentowany jest klient i serwer demonstrujące model komunikacji wykorzystujący protokół datagramowy w dziedzinie Internetu.

Serwer współbieżny protokołu strumieniowego działający w dziedzinie Uniksa.

Serwer tworzy gniazdo strumieniowego w dziedzinie Uniksa i przydziela mu lokalny adres. Następnie zgłasza gotowość odbierania połączeń i oczekuje na połączenia klientów. Po nawiązaniu połączenia serwer tworzy proces potomny, który przejmuje obsługę bieżącego połączenia. Proces potomny odczytuje wiadomość od klienta i wypisuje na ekranie terminala. Adres serwera jest ustalony w kodzie poprzez nazwę ścieżkową pliku (./socket1.tmp).

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>
/*adres serwera*/
#define UNIX_PATH "./socket1.tmp"
#define MAX 80

main(int argc, char *argv[])
{
  int fd, newfd, pid, serverLength, clientLength;
  struct sockaddr_un clientAddr, serverAddr;
  char buf[MAX]="\0";

  /*tworzenie gniazda*/
  if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
    perror("Blad socket");
    exit(1);
  }

  /*wypełnienie struktury adresowej serwera*/
  bzero((char*) &serverAddr, sizeof(serverAddr));
  serverAddr.sun_family = AF_UNIX;
  strcpy(serverAddr.sun_path, UNIX_PATH);
  serverLength = strlen(serverAddr.sun_path) +
sizeof(serverAddr.sun_family);

  /*przydzielenie lokalnego adresu serwera*/
  if (bind(fd, (struct sockaddr *) &serverAddr, serverLength) == -1)  {
    perror("Blad bind");
    exit(1);
  }

  /*zgloszenie gotowości odbierania połączeń*/
  listen(fd, 5);
  for (;;) {
    clientLength = sizeof(clientAddr);

    /*oczekiwanie na polaczenie klienta*/
    if ((newfd = accept(fd, (struct sockaddr *) &clientAddr,
&clientLength)) == -1) {
      perror("Blad accept");
```

```
      exit(1);
    }

    /*utworzenie procesu potomnego*/
    if ((pid = fork()) == -1) {
      perror("Blad fork");
      exit(1);
    }
    /*obsluga polaczenia przez proces potomny*/
    else if (pid == 0) {
      close(fd);
      read(newfd, buf, MAX);
      printf("\n%s %s\n", clientAddr.sun_path, buf);
      exit(0);
    }
    close(newfd);
  }
}
```

Klient protokołu strumieniowego działający w dziedzinie Unix-a.

Klient tworzy gniazdo strumieniowego w dziedzinie Unixa i przydziela mu lokalny adres. Następnie nawiązuje połączenie z serwerem i przesyła mu wiadomość. Adres serwera jest ustalony w kodzie poprzez nazwę ścieżkową pliku (./socket1.tmp).

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

/*adres serwera*/
#define UNIX_PATH "./socket1.tmp"
#define MAX 80

main(int argc, char *argv[])
{
  int fd, serverLength, n;
  struct sockaddr_un serverAddr;
  char buf[MAX]="\0";
  if (argc != 2) {
    fprintf(stderr, "Poprawne wywolanie: %s wiadomosc\n", argv[0]);
    exit(1);
  }
  /*tworzenie gniazda*/
  if ((fd = socket(AF_UNIX, SOCK_STREAM, 0)) == -1) {
    perror("Blad socket");
    exit(1);
  }
  /*wypełnienie struktury adresowej serwera*/
  bzero((char*) &serverAddr, sizeof(serverAddr));
  serverAddr.sun_family = AF_UNIX;
  strcpy(serverAddr.sun_path, UNIX_PATH);
  serverLength = strlen(serverAddr.sun_path) +
sizeof(serverAddr.sun_family);

  /*nawiązanie połączenia z serwerem*/
  if (connect(fd, (struct sockaddr *) &serverAddr, serverLength) == -1)
```

```
  {
    perror("Blad connect");
    exit(1);
  }
  /*kopiowanie i wysyłanie danych*/
  strcpy(buf, argv[1]);
  n = strlen(buf);
  write(fd, buf, n);
  close(fd);
  exit(0);
}
```

Serwer iteracyjny protokołu datagramowego działający w dziedzinie Internetu.

Serwer tworzy gniazdo datagramowe w dziedzinie Internetu i przydziela mu lokalny adres. Następnie oczekuje na datagramy od klientów. Z każdego datagramu odczytuje wiadomość i adres nadawcy oraz wypisuje na ekranie terminala.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
/*ustalenie portu usługi*/
#define SERVER_PORT 5500
#define MAX 80

main(int argc, char *argv[])
{
  int fd, newfd, pid, serverLength, clientLength;
  struct sockaddr_in clientAddr, serverAddr, cAddr;
  char buf[MAX]="\0", *addr;

  /*tworzenie gniazda*/
  if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror("Blad socket");
    exit(1);
  }

  /*wypełnienie struktury adresowej serwera*/
  bzero((char*) &serverAddr, sizeof(serverAddr));
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  serverAddr.sin_port = htons(SERVER_PORT);
  serverLength = sizeof(serverAddr);

  /*przydzielenie lokalnego adresu serwera*/
  if (bind(fd, (struct sockaddr *) &serverAddr, serverLength) == -1) {
    perror("Blad bind");
    exit(1);
  }
  for (;;) {
    clientLength = sizeof(clientAddr);
    /*oczekiwanie na nadejście datagramu*/

    if (recvfrom(fd, buf, MAX, 0, (struct sockaddr *) &clientAddr,
&clientLength) == -1) {
      perror("Blad recvfrom");
```

```
      exit(1);
    }
    /*wypisanie adresu klienta i wiadomości*/
    addr = inet_ntoa(clientAddr.sin_addr);
    printf("\n%s\t%s\n", addr, buf);
  }
}
```

Klient protokołu datagramowego działający w dziedzinie Internetu.

Klient tworzy gniazdo datagramowe w dziedzinie Internetu i przydziela mu lokalny adres. Następnie wpisuje do struktury adresowej adres serwera, podany w linii wywołania, oraz numer portu serwera. W obydwu przypadkach dokonuje zamiany kolejności bajtów na sieciową. Klient wysyła podaną wiadomość w postaci datagramu na adres serwera.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
/*ustalenie portu usługi*/
#define SERVER_PORT 5500
#define MAX 80

main(int argc, char *argv[])
{
  int fd, serverLength, n;
  struct sockaddr_in serverAddr, clientAddr;
  char buf[MAX]="\0";

  if (argc != 3) {
    fprintf(stderr, "Poprawne wywolanie: %s adres wiadomosc\n", argv[0]);
    exit(1);
  }

  /*tworzenie gniazda*/
  if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
    perror("Blad socket");
    exit(1);
  }
  /*wypełnienie struktury adresowej klienta*/
  bzero((char*) &clientAddr, sizeof(clientAddr));
  clientAddr.sin_family = AF_INET;
  clientAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  clientAddr.sin_port = htons(0);

  /*przydzielenie lokalnego adresu klienta*/
  if (bind(fd, (struct sockaddr *) &clientAddr, sizeof(clientAddr)) ==-1)
  {
    perror("Blad bind");
    exit(1);
  }
  /*przepisanie wiadomości do bufora*/
  strcpy(buf, argv[2]);
  n = strlen(buf);
  /*wypełnienie struktury adresowej serwera*/
  bzero((char*) &serverAddr, sizeof(serverAddr));
```

```
  serverAddr.sin_family = AF_INET;
  serverAddr.sin_addr.s_addr = inet_addr(argv[1]);
  serverAddr.sin_port = htons(SERVER_PORT);

  /*wysłanie datagramu*/
  sendto(fd, buf, MAX, 0, (struct sockaddr *) &serverAddr,
sizeof(serverAddr));

  /*zamknięcie gniazda*/
  close(fd);
  exit(0);
}
```

Bibliografia

[1] Glass G., Ables K.: Linux dla programistów i użytkowników, Wydawnictwo Helion 2007 (rozdziały: 12.6, 13,6)

[2] Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000 (rozdział 16)

[3] Rochkind M.J.: Programowanie w systemie UNIX dla zaawansowanych, WNT 2007 (rozdział 8)

Słownik

Termin	Objaśnienie
asocjacja	pięcioelementowy zbiór parametrów opisujący połączenie w sieci ustanawiane między dwoma procesami działającymi na dwóch różnych stacjach
bezpołączeniowy tryb komunikacji	tryb, w którym procesy nie nawiązują połączenia, tylko wysyłają do siebie niezależne komunikaty zwane datagramami; inna nazwa: tryb datagramowy
gniazdo	mechanizm umożliwiający komunikację sieciową procesów z wykorzystaniem różnych rodzin protokołów sieciowych lub komunikację międzyprocesową w dziedzinie Uniksa, czyli wewnątrz jednego systemu operacyjnego z wykorzystaniem jego wewnętrznych protokołów
interfejs programu użytkowego API	specjalny interfejs programowy pomiędzy warstwą transportową i warstwą procesu
połączeniowy tryb komunikacji	tryb, w którym procesy muszą ustanowić logiczne połączenie, aby rozpocząć komunikację
serwer iteracyjny	serwer, który bezpośrednio obsługuje nadchodzące zlecenia klientów
serwer współbieżny	serwer, który tworzy nowy proces potomny do obsługi każdego kolejnego zlecenia, a sam oczekuje na kolejne zlecenia klientów

Zadania do wykładu 14

Zadanie 1

Napisz zestaw programów dla serwera i klienta usługi echo. Zadaniem serwera jest odsyłanie do klienta wszystkich nadesłanych przez niego danych. Składnia wywołania programów powinna być następująca:

program_serwera numer_portu

program_klienta adres_komputera | nazwa_komputera numer_portu

Zrealizuj jedną z dwóch wersji: dla protokołu TCP (serwer współbieżny) lub UDP (serwer iteracyjny).

Zadanie 2

Napisz zestaw programów dla serwera i dwóch klientów. Serwer iteracyjny przekazuje wiadomości (w postaci datagramów) pomiędzy dwoma klientami. Proces klienta odbiera dane od serwera i wypisuje je na **stdout** oraz wczytuje nowe wiadomości z **stdin** i wysyła je do serwera. Jeden z klientów operuje w dziedzinie Internetu a drugi w dziedzinie Uniksa. Kolejność i częstotliwość nadawania przez klientów nie jest określona. Rozwiązanie powinno angażować jak najmniej zasobów systemowych i odznaczać się krótkim czasem reakcji.