Упражнение 7

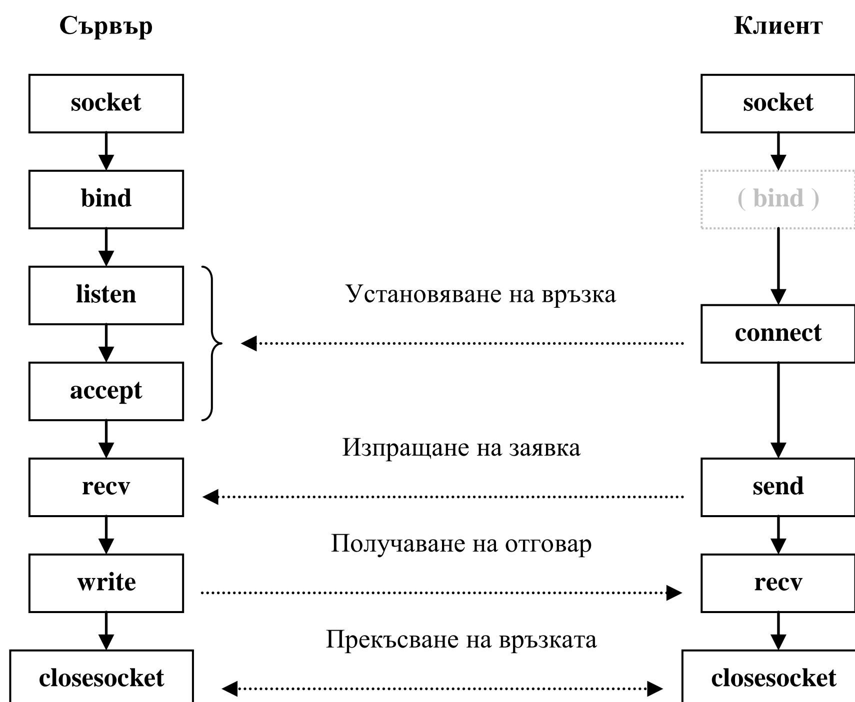
Мрежово програмиране на език С - програми „Клиент” и „Сървър”

Модел Клиент – Сървър

Моделът Клиент – Сървър е най- често използуваната схема за създаване на разпределени приложения. Според нея клиентските приложения имат инициативата изпращайки заявки до приложението сървър и очаквайки неговите отговори.

Това предполага някаква асиметрия в установяването на връзката между клиент и сървър. В процеса на установяване на връзка приложението сървър обикновено очаквава на някой от добре известните адреси (например WWW – порт 80, FTP порт 21, POP порт 110, SMTP порт 23 и др.) заявка по инициативата на клиента.

Взаимодействието протича при следната последователност:



Интрефейс потребител – TCP

- интрфейсът потребител - TCP е зависим от характерните особености на отделните операционни системи;
- конкретните реализации на TCP могат да имат различен потребителски интерфейс;
- всички те обаче предостяват минимален набор от общи услуги, с което гарантират съвместимост в рамките на различни компютърни архитектури;

- този минимален набор се базира на потребителският програмен сокет интерфейс (**Socket API**);
- Socket API е разработен за първи път като разширение към входно-изходната система на операционната система Berkeley BSD UNIX и се превръща “де факто” в стандарт;

Функции за мрежово програмиране от Socket API

1. Получаване на името на локалната система:

```
int gethostname(char * name, int namelen);
```

name	указател към буфер, в който ще бъде записано името на локалната система.
namelen	дължина на буфера.

Функцията записва символен низ с името на локалната система в буфера `name`. Това име може да бъде зададено впоследствие като параметър на функцията **`gethostbyname()`**.

При успешен край функцията връща 0, а при грешка `SOCKET_ERROR`.

2. Намиране на името на система по нейния IP адрес:

```
struct hostent * gethostbyaddr(const char * addr, int
        len, int type);
```

addr	указател към адреса в мрежов формат.
len	дължина на адреса (4 за `PF_INET` адресите).
type	тип на адреса, за Internet трябва да е `PF_INET`.

Функцията връща указател към структура от тип `hostent`. При наличие на грешка връща `NULL` указател.

Ако IP адресът е въведен от клавиатурата като символен низ в десетичен точкуван формат, първо трябва да се използва функцията **`inet_addr()`** за преобразуването му в 4 байтово число в мрежов формат и едва след това **`gethostbyaddr()`** за получаване на `hostent` структурата.

```
struct hostent {
     char *    h_name;
     char * *  h_aliases;
     short          h_addrtype;
     short          h_length;
     char * *  h_addr_list;
};
#define h_addr  h_addr_list[0]
```

където:

`h_name` име на системета (PC).

`h_aliases` масив от указатели към алтернативни имена, завършващ с `NULL`-елемент.
`h_addrtype` тип на адресите, за Internet винаги е `PF_INET`.
`h_length` дължина на адресите, за `PF_INET` винаги е `4`.
`h_addr_list` масив от указатели към всички IP адреси на системата, завършващ с `NULL`-елемент.

3. Намиране на IP адреса на система по нейното име:

`struct hostent * gethostbyname(const char *` *`name`*`);`

`name` указател към името на системата.

Функцията връща указател към описаната по- горе структура от тип `hostent`. При наличие на грешка връща `NULL` указател.

4. Преобразуване на IP адрес от десетичен точкуван в мрежов формат:

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

`cp` символен низ, представящ IP адрес в десетичен точкуван формат.

Функцията връща цифров IP адрес в мрежов формат. При наличие на грешка връща стойността `INADDR_NONE`.

5. Преобразуване на IP адрес от мрежов в десетичен точкуван формат:

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

`in` структура от тип `in_addr`, съдържаща IP адрес в мрежов формат.

```
struct in_addr {
    union {
         struct { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
         struct { u_short s_w1,s_w2; } S_un_w;
         u_long S_addr;
    } S_un;
}

#define s_addr  S_un.S_addr
typedef struct in_addr IN_ADDR;
```

Функцията връща указател към ASCII символен низ, представящ IP адрес в десетичен точкуван формат. При наличие на грешка връща стойността `NULL`.

6. Намиране на протокол по име:

```
struct protoent * getprotobyname(const char * name);
```

name указател към името на протокола.

Функцията връща указател към структура от тип protent. При наличие на грешка връща NULL указател.

```
struct protoent {
     char *    p_name;
     char * *  p_aliases;
     short          p_proto;
};
```

където:

p_name официално име на протокола.

p_aliases масив от алтернативни имена, завършващ с NULL-елемент.

p_proto номер на протокола в локален формат.

7. Намиране на протокол по номер:

```
struct protoent * getprotobynumber(int number);
```

number номер на протокола в локален формат.

Функцията връща указател към описаната по- горе структура от тип protent. При наличие на грешка връща NULL указател.

8. Намиране на услуга по име:

```
struct servent * getservbyname(const char * name, const
        char * proto );
```

name указател към името на услугата.

proto указател към името на използувания протокол за достъп до услугата. Може да бъде и NULL.

Функцията връща указател към структура от тип servent. При наличие на грешка връща NULL указател.

```
struct servent {
     char *    s_name;
     char * *  s_aliases;
     short          s_port;
     char *    s_proto;
};
```

където:

`s_name`	официално име на услугата.
`s_aliases`	масив от указатели към алтернативни имена, завършващ с `NULL`-елемент.
`s_port`	номер на порта определен за услугата - мрежов формат.
`s_proto`	име на използувания протокол за достъп до услугата.

8. Намиране на услуга по номер на порт:

```
struct servent * getservbyport(int port, const char *proto);
```

port	номер на порта определен за услугата - мрежов формат.
proto	указател към името на използувания протокол за достъп до услугата. Може да бъде и `NULL`.

Функцията връща указател към описаната по- горе структура от тип `servent`. При наличие на грешка връща `NULL` указател.

9. Превръщане на число от локален в мрежов формат:

```
unsigned short htons(unsigned short hostshort);
unsigned long htonl(unsigned long hostlong);
```

hostshort	16-битово число в локален формат.
hostlong	32- битово число в локален формат.

Превръщат цяло число от локален в мрежов формат.

10. Превръщане на число от мрежов в локален формат:

```
unsigned short ntohs(unsigned short netshort);
unsigned long ntohs(unsigned long netlong);
```

netshort	16-битово число в мрежов формат.
netlong	32- битово число в мрежов формат.

Превръщат цяло число от мрежов в локален формат.

11. Създаване на дескриптор на сокет:

```
SOCKET socket(int af, int type, int protocol);
```

af	спецификатор на адресен формат. За Internet се задава адресен формат `PF_INET`.

type тип на сокета. За TCP се задава тип `SOCK_STREAM`, а за UDP - `SOCK_DGRAM`.
protocol номер на комуникационнен протокол. За TCP се задава протокол с номер `IPPROTO_TCP`, а за UDP - `IPPROTO_UDP` . Ако се зададе номер `0` се избира протокола по подразбиране за дадения тип сокет.

Функцията връща дескриптор на специфицирания сокет. При наличие на грешка връща стойност `INVALID_SOCKET`.

12. Свързване на сокет с локално име:

```
int bind(SOCKET s, const struct sockaddr * name,
     int namelen);
```

s дескриптор на сокет.
name структура, съдържаща елементите на локалното име (адресен формат, номер на порт, адрес на система и др.).
namelen дължина на структурата *name*.

Функцията установява съответствие между дескриптор на сокет и локално име, като го привързва към IP адрес на система и номер на порт. При успешен край връща стойност `0`. При наличие на грешка връща стойност `SOCKET_ERROR`.

```
struct sockaddr_in {
     short   sin_family;
     u_short sin_port;
     struct  in_addr sin_addr;
     char    sin_zero[8];
};

struct in_addr {
     union {
         struct {u_char s_b1,s_b2,s_b3,s_b4;} S_un_b;
         struct {u_short s_w1,s_w2;} S_un_w;
         u_long S_addr;
     } S_un;
}

#define s_addr  S_un.S_addr
```

13. Установяване на връзка с отдалечен сокет:

```
int connect(SOCKET s, const struct sockaddr * name, int
     namelen);
```

s локален дескриптор на сокет.

name структура съдържаща името на отдалечения сокет, с който трябва да се установи връзка.
namelen дължина на структурата *name*.

Функцията установява връзка между двойка сокети- локален и отдалечен. Ако локалния сокет е неименован, функцията го привързва към локалния IP адрес на системата и към случайно избран номер на порт, по- голям от 1023. Името на отдалечения сокет се задава в структурата *name*.

При успешено установена връзка функцията връща стойност 0. При наличие на грешка връща стойност SOCKET_ERROR.

14. Очакване на заявки за установяване на връзка:

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

s дескриптор на именован, но несвързан сокет.
backlog максимален брой на заявки очакващи установяване на връзка, които може да бъдат приети.

Функцията поставя именован сокет в режим на очакване на заявки за установяване на връзка по външна инициатива. Използува се основно от страна на процеси от тип сървър. Пристигащите заявки за установяване на връзка се приемат и се редят в опашка с дължина *backlog*.

При успешен край функцията връща стойност 0. При наличие на грешка връща стойност SOCKET_ERROR.

15. Приемане на заявка за установяване на връзка:

```
SOCKET accept (SOCKET s, struct sockaddr * name, int * namelen);
```

s дескриптор на сокет, очакващ установяване на връзка чрез функцията **listen**().
name структура в която при успех се попълва името на отдалечения сокет, с който е установена връзката.
namelen указател към променлива, съдържаща дължината на структурата *name*.

Функцията избира първата чакаща заявка за установяване на връзка от опашката към сокета *s*, създава нов именован сокет със същите параметри, установява връзка между него и чакащата заявка и връща дескриптор към новия сокет. Името на отдалечения сокет - инициатор на заявката, се попълва в структурата *name*.

При успешен край функцията връща дескриптора на новосъздадения сокет. При наличие на грешка връща стойност INVALID_SOCKET.

15. Приемане на данни:

```
int recv(SOCKET s, char * buf, int len, int flags);
```

s	дескриптор на свързан сокет.
buf	указател към приемния буфер за данните.
len	дължина на буфера.
flags	определя различни режими на изпълнение. Типична стойност е 0.

Функцията прочита в буфера получените към зададения сокет данни. При успешен край връща реалния брой прочетени байтове. При наличие на грешка връща стойност SOCKET_ERROR. Ако междувременно връзката е прекратена по инициатива на отсрещната страна, функцията връща 0.

15. Предаване на данни:

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

s	дескриптор на свързан сокет.
buf	указател към буфер, съдържащ данните за предаване.
len	дължина на буфера.
flags	определя различни режими на изпълнение. Типична стойност е 0.

Функцията изпраща данни за предаване към зададения сокет. При успешен край връща реалния брой предадени за изпращане байтове. При наличие на грешка връща стойност SOCKET_ERROR. Успешното завършване на функцията не означава, че данните са получени коректно в отсрещната страна.

18. Затваряне на сокет:

```
int closesocket(SOCKET s);
```

s	дескриптор на сокет.

Функцията затваря сокета и освобождава дескриптора му. При успешен край връща стойност 0. При наличие на грешка връща стойност SOCKET_ERROR.

19. Определяне на статуса на множество сокети:

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

nfds	този аргумент не се използува. Оставен е само за съвместимост.
readfds	незадължителен указател към множество от сокети, които ще бъдат проверявани за готовност за приемане.

Ако такива сокети липсват, като параметър се задава NULL.

writefds незадължителен указател към множество от сокети, които ще бъдат проверявани за готовност за предаване. Ако такива сокети липсват, като параметър се задава NULL.

exceptfds незадължителен указател към множество от сокети, които ще бъдат проверявани за възникнали грешки. Ако такива сокети липсват, като параметър се задава NULL.

timeout максимално време, което **select()** ще изчаква ако няма констатиран резултат.

Функцията определя състоянието на един или повече сокети. За всеки сокет може да бъде изисквана статус информация относно готовност за четене, запис или за възникнали грешки.

Множеството сокети, за които се изисква една и съща статус информация, се задава в fd_set структура. Функцията обновява тази структура, така че в нея остават само сокетите, чийто статус отговаря на проверяваното условие. **Select()** връща и техния общ брой.

Съществуват множество макроси, улесняващи работата с fd_set структурите. Те са дефинирани в **winsock.h**.

Макросите са:

FD_CLR(*s*, ***** *set***)** изтрива дескриптора *s* от структурата *set*.

FD_SET(*s*, ***** *set***)** добавя дескриптора *s* към структурата *set*.

FD_ZERO(* *set***)** изтрива всички дескриптори от структурата *set*.

FD_ISSET(*s*, ***** *set***)** връща ненулева стойност ако *s* е член на структурата *set*, нула ако това не е така.

Максималният брой сокети във всяка стуктура по подразбиране е установен на 64 и може да се променя от потребителя.

Параметърът *timeout* определя колко време **select()** да изчака преди да завърши с проверката. Ако за *timeout* е зададен null указател, **select()** ще блокира докъто поне един от зададените за проверка дескриптори не изпълни проверяваното условие. Ако timeval е инициализирана с {0,0}, **select()** връща изискваната информация веднага.

```
struct timeval {
     long    tv_sec;          /* seconds */
     long    tv_usec;         /* and microseconds
*/
};
```

При успешен край функцията връща общия брой дескриптори изпълнили критерия за проверка. При наличие на грешка връща стойност SOCKET_ERROR.

Задачи за изпълнение

1. Да се реализира програма „Клиент” при следните условия:
   - програмата осъщестрвява TCP връзка с произволен сървър в Интернет, прочита и разпечатва приветстваващото му съобщение и прекратява връзката;
   - символният адрес на сървъра и номерът на порта за връзка се задават от клавиатурата, след стартиране на програмата;
   - при успешно установяване на връзка програмата разпечатва IP адреса на сървъра.

Решение:

```
                              /* ---------       CLIENT   ------------------*/
#include <stdio.h>
#include <winsock.h>

int main() {
   WSADATA wsaData;
   WORD wVersionRequested;
   Int err, i;
   wVersionRequested = MAKEWORD(1, 1);
   err = WSAStartup(wVersionRequested, &wsaData);
   if (err != 0) {
      printf("Couldn't find an useable winsock.dll.\n");
      return (1);
   }
                              /* Get server name and port number */
   int serv_port;
   char buf[100];
   printf("Enter server name: ");
   scanf("%s", buf);
   printf("Enter server port number: ");
   scanf("%d", &serv_port);

                              /* Find server by name */

   struct hostent * server;
   server = gethostbyname(buf);
   if(server == NULL){
        printf("\nInvalid server name!\n");
        WSACleanup();
        return 1;
   }
                              /* Create a SOCKET */

   SOCKET s1;
   s1 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if(s1 == INVALID_SOCKET){
        printf("\nInvalid socket creation!\n");
        WSACleanup();
        return 1;
   }
                              /* Set up the remote SOCKET of the connection */

   struct sockaddr_in  serv_addr;
   memset((char *) &serv_addr, 0, sizeof serv_addr);
```

```
    serv_addr.sin_family          = AF_INET;
    serv_addr.sin_addr.s_addr   = ((long *)server->h_addr)[0];
    serv_addr.sin_port            = htons(serv_port);

                          /* Establish connection */

    if(connect(s1, &serv_addr, sizeof serv_addr) < 0){
        printf("\nCan't establish connection!\n");
        WSACleanup();
        return 1;
     }
     else
        printf("Connected to server: %s\n", inet_ntoa(serv_addr.sin_addr));

                          /* Read the server's hello message */

     err = recv(s1, buf, sizeof buf, 0);
     if(err > 0){
         buf[err] = '\0';
         printf("\nHello message: %s\n", buf);
     }
     else
         printf("\nNo hello message!\n");

                          /* Close connection */

     closesocket(s1);
     WSACleanup();
     return(0);
  }
```

2. Да се реализира програма „Сървър” при следните условия:
   - сървърът очаква установяване на връзка на порт 3000;
   - при успешно установяване на връзка програмата разпечатва IP адреса на съответния клиент.
   - сървърът изпраща приветстващо текстово съобщение и разкъсва връзката.

Решение:

```
            /* ---------          SERVER  ------------------*/

        #include <stdio.h>
        #include <winsock.h>

        int main() {
                WSADATA        wsaData;
                WORD           wVersionRequested;
                int            err;

                wVersionRequested = MAKEWORD(1, 1);
                err = WSAStartup(wVersionRequested, &wsaData);
                if (err != 0) {
                        printf("Couldn't find an useable winsock.dll.\n");
                        return (1);
                }
```

```
/* ---------          Create Socket  ------------------*/

SOCKET          s1, new_sock;
struct sockaddr_in  serv_addr, cli_addr;
int cli_len = sizeof cli_addr;
char hello_msg[] = "This is my hello message!";

s1 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(s1 == INVALID_SOCKET){
        printf("Invalid socket!\n");
return 1;
}

/* ---------          Bind Socket  ------------------*/

memset((char*)&serv_addr, 0, sizeof serv_addr);
serv_addr.sin_family                = AF_INET;
serv_addr.sin_port                  = htons(3000);
serv_addr.sin_addr.s_addr           = htonl(INADDR_ANY);
if(bind(s1, &serv_addr, sizeof serv_addr) < 0) {
        printf("Bind Error!\n");
        return 1;
}

while(TRUE){
        printf("Server is waiting for connection ...\n");

        listen(s1, 4);

/* ---------          Establish Connection  ------------------*/

        new_sock = accept(s1, &cli_addr, &cli_len);
        if(new_sock == INVALID_SOCKET){
                printf("Accept Error!\n");
                return 1;
                break;
        }

        printf("Connection with client: %s\n", inet_ntoa(cli_addr.sin_addr));

/* Send server's hello message */

        send(new_sock, hello_msg, sizeof hello_msg, 0);

/* Close connection */


        closesocket(new_sock);
}

closesocket(s1);

WSACleanup();
return(0);
}
```