Sockets

Pedroso

4 de março de 2009

1 Introdução

A Interface que um aplicativo utiliza quanto interage com um módulo implementado por outro software é chamado de API (Application Programing Interfaced). Todas as funcionalidades dos protocolos TCP, IP ou Ethernet são implementadas dentro do núcleo do sistema operacional. A interface adotada para que os aplicativos de usuários acessem as funcionalidades do TCP ou do UDP é a API Sockets, que foi originada no sistema operacional UNIX, e que hoje é adotada como padrão para o acesso a funcionalidades da pilha de protocolos TCP/IP.

O sistema utiliza o paradigma de trabalho do sistema Unix para operações de E/S; ou seja, todas as operações de E/S exigem a sequência open read — write close. Do ponto de vista do aplicativo, estará sendo realizado um acesso que pode ser comparado à operação de escrita e leitura de um arquivo.

2 Descritores

Quando o aplicativo solicita que seja criado um socket, o sistema operacional irá criá-lo e o aplicativo poderá realizar manipulações no socket aberto através de um descritor. Na abertura do socket (operação open) será necessário informar sistema operacional sobre o endereço e porta de destino, e em caso de sucesso o sistema retorna o descritor. Nas demais operações de leitura e escrita, apenas o descritor será suficiente.

3 Procedimentos da API Socket

int socket(int familia_protocolo, int tipo, int protocolo); O procedimento socket cria um socket e retorna um descritor. O descritor é um número que permite que se realize o acesso ao recurso reservado dentro do sistema operacional. O argumento família_protocolo especifica qual o tipo de protocolo. Atualmente no Unix são suportados os seguintes protocolos:

PF_UNIX, PF_LOCAL	Local communication	unix(7)
PF_INET	IPv4 Internet protocols	ip(7)
PF_INET6	IPv6 Internet protocols	
PF_IPX	IPX - Novell protocols	
PF_NETLINK	Kernel user interface device	netlink(7)
PF_X25	ITU-T X.25 / ISO-8208 protocol	x25(7)
PF_AX25	Amateur radio AX.25 protocol	
PF_ATMPVC	Access to raw ATM PVCs	
PF_APPLETALK	Appletalk	ddp(7)
PF_PACKET	Low level packet interface	packet(7)

O argumento tipo especifica o tipo de comunicação. As opções mais importantes são: SOCK_STREAM corresponde a um protocolo com conexão (estará especificando o uso do TCP); SOCK_DGRAM que corresponde a uma transferência sem conexão (estará especificando o uso do UDP); SOCK_RAW o aplicativo será responsável por montar todos os pacotes; o sistema operacional irá apenas transmiti-los.

O argumento protocolo especifica o protocolo de transporte a ser utilizado (TCP ou UDP, por exemplo). Para protocolos que implementam protocolos com e sem conexão no mesmo protocolo de transporte, deve ser utilizado em conjunção com o parâmetro tipo. No caso do TCP e do UDP, a informação é a mesma contida no campo tipo porque o TCP implementa apenas um modo com conexão e o UDP apenas um modo sem conexão.

int close(int fd); Informa ao sistema para encerrar o uso do socket. O parâmetro é o descritor

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen); O procedimento bind liga o sockfd ao endereço local my_addr, que possui um tamanho de addrlen Tradicionalmente isso é chamado de ”ligar um nome ao socket”, que é representado por uma família de endereços. É necessário determinar um endereço local ao socket antes de realizar uma conexão e transmitir ou receber dados. O valor de retorno será: zero em caso de sucesso; -1 em caso de erro (e a errno contém o código do erro).

int listen(int sockfd, int backlog); Para aceitar uma conexão, um socket criado com a função socket() deve aceitar conexões. Normalmente, cada conexão inicia uma thread ou processo para atendê-la. O procedimento listen é utilizado para limitar a fila para conexões que não foram atendidas pelo processo do usuário. O procedimento listen aplica-se apenas ao modo de transmissão com conexão (ex., quando for utilizado o TCP). O parâmetro backlog define o tamanho máximo da fila de conexões pendentes. Se uma nova requisição chega e a fila está cheia, o cliente irá receber um erro com a indicação ECONNREFUSED (ou, se o protocolo suporta retransmissão, a requisição pode ser ignorada e novas tentativas de conexão se sucederão automaticamente). O valor de retorno será: zero em caso de sucesso; -1 em caso de erro (e a errno contém o código do erro).

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen); A função accept é utilizada em sockets do tipo com conexão. Quando chamada, a função irá aguardar uma conexão. Um exemplo ilustrativo é dado abaixo:

```
while(true) {
```

```
novoSocket=accept(s, ...);
novaThread(novoSocket, ...);
}
```

No caso típico apresentado acima, o procedimento accept espera uma nova conexão. Quando um cliente estabelece uma conexão, o procedimento retorna identificador da conexão aberta (um novo descritor de socket). O programa acima indica que será lançada uma thread para atender a nova conexão e o laço de repetição garante que o servidor irá esperar uma próxima conexão. O descritor do socket original não será afetado por esta chamada ou por operações realizadas sobre o descritor novoSocket.

A chamada do procedimento accept deve ser precedida pela chamada do procedimento bind e listen.

O argumento addr é um ponteiro para a estrutura sockaddr. Esta estrutura é preenchida com o endereço do cliente. O formato exato do endereço passado no parâmetro é determinado pela família de protocolo. O parâmetro addrlen indica inicialmente o parâmetro da estrutura apontada por addr e no retorno da função irá conter o valor verdadeiro em bytes do endereço retornado.

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen); Este procedimento é utilizado pelo cliente para estabelecer uma conexão com um servidor quando for utilizado o modo com conexão. O parâmetro serv_addr é um ponteiro para uma estrutura contendo o endereço do servidor que se deseja estabelecer a conexão. O parâmetro addrlen indica o tamanho desta estrutura.

ssize_t send(int s, const void *msg, size_t len, int flags);ssize_t sendto(int s, const void *msg, size_t len, int fla
As funções send, sendto e sendmsg são utilizadas para transmitir uma mensagem para outro socket. A função send pode ser utilizada somente no modo com conexão e as funções sendto e sendmsg podem ser utilizadas no modo com e sem conexão.

O endereço do destino é dado pelo parâmetro to e o campo tolen especifica o tamanho do endereço. O parâmetro len especifica o tamanho da mensagem, enquanto o campo msg é um ponteiro para a mensagem a ser transmitida propriamente dita

O valor de retorno será a quantidade de bytes transmitidos em caso de sucesso ou um valor menor que zero indicando o código do erro.

ssize_t recv(int s, void *buf, size_t len, int flags);ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct soc
São funções utilizadas para receber dados. A função recv pode ser utilizada apenas em casos com conexão e as funções recvfrom e recvmsg são utilizas para receber mensagens em casos com ou sem conexão. Sua utilização e valores de retorno são semelhantes as funções send descritas anteriormente.

4 Seqüência de chamada de funções

Para o servidor:

1. Com um sistema TCP SOCKET $\longrightarrow$ BIND $\longrightarrow$ LISTEN $\longrightarrow$ ACCEPT $\longrightarrow$ SEND | RECV

2.Com um sistema UDP SOCKET ⟶ BIND ⟶ SEND | RECV
Para o cliente:
1.Com um sistema TCP SOCKET ⟶ CONECT ⟶ SEND | RECV
2.Com um sistema UDP SOCKET ⟶ SEND | RECV

5 Exemplos

5.1 Servidor TCP

```
/* server.c - code for example server program that uses TCP */
#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#endif

#include <stdio.h>
#include <string.h>

#define PROTOPORT       5193            /* default protocol port number */
#define QLEN            6               /* size of request queue        */

int     visits      =   0,              /* counts client connections    */
i = 0;
/*------------------------------------------------------------------------
* Program:   server
*
* Purpose:   allocate a socket and then repeatedly execute the following:
*              (1) wait for the next connection from a client
*              (2) send a short message to the client
*              (3) close the connection
*              (4) go back to step (1)
*
* Syntax:    server [ port ]
*
*               port  - protocol port number to use
*
* Note:      The port argument is optional.  If no port is specified,
*            the server uses the default given by PROTOPORT.
```

```
*
*------------------------------------------------------------------------
*/
main(argc, argv)
int     argc;
char    *argv[];
{
struct  protoent *ptrp;  /* pointer to a protocol table entry   */
struct  sockaddr_in sad; /* structure to hold server's address  */
struct  sockaddr_in cad; /* structure to hold client's address  */
int     sd, sd2;         /* socket descriptors                  */
int     port;            /* protocol port number                */
int     alen;            /* length of address                   */
char    buf[1000];       /* buffer for string the server sends  */

#ifdef WIN32
WSADATA wsaData;
WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;         /* set family to Internet     */
sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address   */

/* Check command-line argument for protocol port and extract    */
/* port number if one is specified.  Otherwise, use the default */
/* port value given by constant PROTOPORT                       */

if (argc > 1) {                 /* if argument specified        */
port = atoi(argv[1]);   /* convert argument to binary   */
} else {
port = PROTOPORT;       /* use default port number      */
}
if (port > 0)                   /* test for illegal value       */
sad.sin_port = htons((u_short)port);
else {                          /* print error message and exit */
fprintf(stderr,"bad port number %s\n",argv[1]);
exit(1);
}

/* Map TCP transport protocol name to protocol number */

if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
fprintf(stderr, "cannot map \"tcp\" to protocol number");
exit(1);
}
```

```
/* Create a socket */

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
fprintf(stderr, "socket creation failed\n");
exit(1);
}

/* Bind a local address to the socket */

if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
fprintf(stderr,"bind failed\n");
exit(1);
}

/* Specify size of request queue */

if (listen(sd, QLEN) < 0) {
fprintf(stderr,"listen failed\n");
exit(1);
}

/* Main server loop - accept and handle requests */

while (1) {
alen = sizeof(cad);
if ( (sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0) {
fprintf(stderr, "accept failed\n");
exit(1);
}
visits++;
i=0;
while (i++<10) {
sprintf(buf,"%d- This server has been contacted %d time%s\n", i, visits,visits==1?".":"s.");
send(sd2,buf,strlen(buf),0);
Sleep(1000);
}
sprintf(buf,"By...");
send(sd2,buf,strlen(buf),0);
closesocket(sd2);
}
}
```

5.2 Cliente TCP

```
* client.c - code for example client program that uses TCP */

#ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#endif

#include <stdio.h>
#include <string.h>

#define PROTOPORT       5193            /* default protocol port number */

extern  int             errno;
char    localhost[] =   "localhost";    /* default host name

/*------------------------------------------------------------------------
* Program:   client
*
* Purpose:   allocate a socket, connect to a server, and print all output
*
* Syntax:    client [ host [port] ]
*
*               host  - name of a computer on which server is executing
*               port  - protocol port number server is using
*
* Note:      Both arguments are optional.  If no host name is specified,
*            the client uses "localhost"; if no protocol port is
*            specified, the client uses the default given by PROTOPORT.
*
*------------------------------------------------------------------------
*/

main(argc, argv)
int     argc;
char    *argv[];
{
```

```
struct  hostent  *ptrh;  /* pointer to a host table entry       */
struct  protoent *ptrp;  /* pointer to a protocol table entry   */
struct  sockaddr_in sad; /* structure to hold an IP address     */
int     sd;              /* socket descriptor                   */
int     port;            /* protocol port number                */
char    *host;           /* pointer to host name                */
int     n;               /* number of characters read           */
char    buf[1000];       /* buffer for data from the server     */
#ifdef WIN32
WSADATA wsaData;
WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;         /* set family to Internet     */

/* Check command-line argument for protocol port and extract    */
/* port number if one is specified.  Otherwise, use the default */
/* port value given by constant PROTOPORT                       */

if (argc > 2) {                 /* if protocol port specified   */
port = atoi(argv[2]);   /* convert to binary            */
} else {
port = PROTOPORT;       /* use default port number      */
}
if (port > 0)                   /* test for legal value         */
sad.sin_port = htons((u_short)port);
else {                          /* print error message and exit */
fprintf(stderr,"bad port number %s\n",argv[2]);
exit(1);
}

/* Check host argument and assign host name. */

if (argc > 1) {
host = argv[1];         /* if host argument specified   */
} else {
host = localhost;
}

/* Convert host name to equivalent IP address and copy to sad. */

ptrh = gethostbyname(host);
if ( ((char *)ptrh) == NULL ) {
fprintf(stderr,"invalid host: %s\n", host);
exit(1);
}
```

```
memcpy(&sad.sin_addr, ptrh->h_addr, ptrh->h_length);

/* Map TCP transport protocol name to protocol number. */

if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
fprintf(stderr, "cannot map \"tcp\" to protocol number");
exit(1);
}

/* Create a socket. */

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
fprintf(stderr, "socket creation failed\n");
exit(1);
}

/* Connect the socket to the specified server. */

if (connect(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
fprintf(stderr,"connect failed\n");
exit(1);
}

/* Repeatedly read data from socket and write to user's screen. */
memset(buf,0, 1000);
n = recv(sd, buf, sizeof(buf), 0);
while (n > 0) {
printf("%s",buf);
memset(buf,0, 1000);
n = recv(sd, buf, sizeof(buf), 0);
}

/* Close the socket. */

closesocket(sd);

/* Terminate the client program gracefully. */

exit(0);
}
```

5.3 Servidor UDP

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h> /* close() */
#include <string.h> /* memset() */

#define LOCAL_SERVER_PORT 1500
#define MAX_MSG 100

int main(int argc, char *argv[]) {

  int sd, rc, n, cliLen;
  struct sockaddr_in cliAddr, servAddr;
  char msg[MAX_MSG];

  /* socket creation */
  sd=socket(AF_INET, SOCK_DGRAM, 0);
  if(sd<0) {
    printf("%s: cannot open socket \n",argv[0]);
    exit(1);
  }

  /* bind local server port */
  servAddr.sin_family = AF_INET;
  servAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servAddr.sin_port = htons(LOCAL_SERVER_PORT);
  rc = bind (sd, (struct sockaddr *) &servAddr,sizeof(servAddr));
  if(rc<0) {
    printf("%s: cannot bind port number %d \n",
   argv[0], LOCAL_SERVER_PORT);
    exit(1);
  }

  printf("%s: waiting for data on port UDP %u\n",
   argv[0],LOCAL_SERVER_PORT);

  /* server infinite loop */
  while(1) {

    /* init buffer */
    memset(msg,0x0,MAX_MSG);
```

```
    /* receive message */
    cliLen = sizeof(cliAddr);
    n = recvfrom(sd, msg, MAX_MSG, 0,
 (struct sockaddr *) &cliAddr, &cliLen);

    if(n<0) {
      printf("%s: cannot receive data \n",argv[0]);
      continue;
    }

    /* print received message */
    printf("%s: from %s:UDP%u : %s \n",
   argv[0],inet_ntoa(cliAddr.sin_addr),
   ntohs(cliAddr.sin_port),msg);

  }/* end of server infinite loop */

return 0;

}
```

5.4 Cliente UDP

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h> /* memset() */
#include <sys/time.h> /* select() */

#define REMOTE_SERVER_PORT 1500
#define MAX_MSG 100


int main(int argc, char *argv[]) {

  int sd, rc, i;
  struct sockaddr_in cliAddr, remoteServAddr;
  struct hostent *h;

  /* check command line args */
  if(argc<3) {
    printf("usage : %s <server> <data1> ... <dataN> \n", argv[0]);
    exit(1);
  }

  /* get server IP address (no check if input is IP address or DNS name */
  h = gethostbyname(argv[1]);
  if(h==NULL) {
    printf("%s: unknown host '%s' \n", argv[0], argv[1]);
    exit(1);
  }

  printf("%s: sending data to '%s' (IP : %s) \n", argv[0], h->h_name,
 inet_ntoa(*(struct in_addr *)h->h_addr_list[0]));

  remoteServAddr.sin_family = h->h_addrtype;
  memcpy((char *) &remoteServAddr.sin_addr.s_addr,
 h->h_addr_list[0], h->h_length);
  remoteServAddr.sin_port = htons(REMOTE_SERVER_PORT);

  /* socket creation */
  sd = socket(AF_INET,SOCK_DGRAM,0);
  if(sd<0) {
```

```
    printf("%s: cannot open socket \n",argv[0]);
    exit(1);
  }

  /* bind any port */
  cliAddr.sin_family = AF_INET;
  cliAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  cliAddr.sin_port = htons(0);

  rc = bind(sd, (struct sockaddr *) &cliAddr, sizeof(cliAddr));
  if(rc<0) {
    printf("%s: cannot bind port\n", argv[0]);
    exit(1);
  }


  /* send data */
  for(i=2;i<argc;i++) {
    rc = sendto(sd, argv[i], strlen(argv[i])+1, 0,
(struct sockaddr *) &remoteServAddr,
sizeof(remoteServAddr));

    if(rc<0) {
      printf("%s: cannot send data %d \n",argv[0],i-1);
      close(sd);
      exit(1);
    }

  }

  return 1;

}
```

5.5 Servidor Multi-thread para MS Windows

```
# ifndef unix
#define WIN32
#include <windows.h>
#include <winsock.h>
#include <process.h>
#else
#define closesocket close
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#endif

#include <stdio.h>
#include <string.h>

#define PROTOPORT       5193            /* default protocol port number */
#define QLEN            6               /* size of request queue        */

int     visits      =   0;              /* counts client connections    */


void AtendeConexao( int *sd2 )
{
int sd=*sd2;
char str[1000];
int i=0;

visits++;
    sprintf(str,"This server has been contacted %d time%s\n", visits,visits==1?".":"s.");
send(sd,str,strlen(str),0);

do {
memset (str, 0, 1000);
sprintf(str,"\nMensagem %d ",i++);
send(sd,str,strlen(str),0);
Sleep(2000);
}
while (i<100);
    closesocket(sd);
_endthread();
}

main(argc, argv)
```

```
int     argc;
char    *argv[];
{
struct  protoent *ptrp;  /* pointer to a protocol table entry   */
struct  sockaddr_in sad; /* structure to hold server's address  */
struct  sockaddr_in cad; /* structure to hold client's address  */
int     sd, sd2;         /* socket descriptors                  */
int     port;            /* protocol port number                */
int     alen;            /* length of address                   */

#ifdef WIN32
WSADATA wsaData;
WSAStartup(0x0101, &wsaData);
#endif
memset((char *)&sad,0,sizeof(sad)); /* clear sockaddr structure */
sad.sin_family = AF_INET;         /* set family to Internet     */
sad.sin_addr.s_addr = INADDR_ANY; /* set the local IP address   */

/* Check command-line argument for protocol port and extract    */
/* port number if one is specified.  Otherwise, use the default */
/* port value given by constant PROTOPORT                       */

if (argc > 1) {                 /* if argument specified        */
port = atoi(argv[1]);   /* convert argument to binary   */
} else {
port = PROTOPORT;       /* use default port number      */
}
if (port > 0)                   /* test for illegal value       */
sad.sin_port = htons((u_short)port);
else {                          /* print error message and exit */
fprintf(stderr,"bad port number %s\n",argv[1]);
exit(1);
}

/* Map TCP transport protocol name to protocol number */

if ( ((int)(ptrp = getprotobyname("tcp"))) == 0) {
fprintf(stderr, "cannot map \"tcp\" to protocol number");
exit(1);
}

/* Create a socket */

sd = socket(PF_INET, SOCK_STREAM, ptrp->p_proto);
if (sd < 0) {
fprintf(stderr, "socket creation failed\n");
```

```
exit(1);
}

/* Bind a local address to the socket */

if (bind(sd, (struct sockaddr *)&sad, sizeof(sad)) < 0) {
fprintf(stderr,"bind failed\n");
exit(1);
}

/* Specify size of request queue */

if (listen(sd, QLEN) < 0) {
fprintf(stderr,"listen failed\n");
exit(1);
}

/* Main server loop - accept and handle requests */

while (1) {
alen = sizeof(cad);
if ( (sd2=accept(sd, (struct sockaddr *)&cad, &alen)) < 0) {
fprintf(stderr, "accept failed\n");
exit(1);
}
printf ("\nServidor atendendo conexão %d", visits);
_beginthread( AtendeConexao, 0, (void *) &sd2 );
}
}
```

6 Exercícios

1. Escreva um programa servidor multi-thread que transmite tudo que for recebido em uma determinada conexão para todas as outras conexões ativas. Teste o servidor com o programa telnet.
2. Escreva um programa cliente para o servidor acima.
3. Escreva um servidor de comandos. O cliente irá transmitir comandos a serem executados no servidor; os comandos serão strings finalizadas pelo caracter ENTER.
4. Escreva um programa servidor e cliente para realizar um CHAT, ou sistema de troca de mensagens. O usuário inicialmente indica seu apelido e poderá enviar mensagens para outro usuário conectado ou para todos.