Tutorial de soquetes

Este é um tutorial simples sobre o uso de soquetes para comunicação entre processos.

O modelo do servidor cliente

A maioria das comunicações interprocessos usa o *modelo de servidor cliente* . Esses termos se referem aos dois processos que estarão se comunicando. Um dos dois processos, o *cliente* , se conecta ao outro processo, o *servidor* , normalmente para fazer uma solicitação de informações. Uma boa analogia é uma pessoa que faz uma ligação para outra pessoa.

Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established.

Notice also that once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a *socket*.
A socket is one end of an interprocess communication channel. The two processes each establish their own socket.

The steps involved in establishing a socket on the *client* side are as follows:

1. Create a socket with the socket() system call
2. Connect the socket to the address of the server using the connect() system call
3. Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

The steps involved in establishing a socket on the *server* side are as follows:

1. Create a socket with the socket() system call
2. Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the listen() system call
4. Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
5. Send and receive data

Socket Types

When a socket is created, the program has to specify the *address domain* and the *socket type*. Two processes can communicate with each other only if their sockets are of the same type and in the same domain.

There are two widely used address domains, the *unix domain*, in which two processes which share a common file system communicate, and the *Internet domain*, in which two processes running on any two hosts on the Internet communicate. Each of these has its own address format.

The address of a socket in the Unix domain is a character string which is basically an entry in the file system.

The address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32 bit address, often referred to as its IP address).
In addition, each socket needs a port number on that host.
Port numbers are 16 bit unsigned integers.
The lower numbers are reserved in Unix for standard services. For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses.
However, port numbers above 2000 are generally available.

There are two widely used socket types, *stream sockets*, and *datagram sockets*. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each uses its own communciations protocol.

Stream sockets use TCP (Transmission Control Protocol), which is a reliable, stream oriented protocol, and datagram sockets use UDP (Unix Datagram Protocol), which is unreliable and message oriented.

The examples in this tutorial will use sockets in the Internet domain using the TCP protocol.

Sample code

C code for a very simple client and server are provided for you. These communicate using stream sockets in the Internet domain. The code is described in detail below. However, before you read the descriptions and look at the code, you should compile and run the two programs to see what they do.

server.c
client.c

Download these into files called `server.c` and `client.c` and compile them separately into two executables called `server` and `client`.

They probably won't require any special compiling flags, but on some solaris systems you may need to link to the socket library by appending `-lsocket` to your compile command.

Ideally, you should run the client and the server on separate hosts on the Internet. Start the server first. Suppose the server is running on a machine called `cheerios`. When you run the server, you need to pass the port number in as an argument. You can choose any number between 2000 and 65535. If this port is already in use on that machine, the server will tell you this and exit. If this happens, just choose another port and try again. If the port is available, the server will block until it receives a connection from the client. Don't be alarmed if the server doesn't do anything;

It's not supposed to do anything until a connection is made.

Here is a typical command line:

```
server 51717
```

Para executar o cliente, é necessário passar dois argumentos, o nome do host no qual o servidor está sendo executado e o número da porta na qual o servidor está atendendo às conexões.

Aqui está a linha de comando para conectar-se ao servidor descrito acima:

```
cheerios do cliente 51717
```

O cliente solicitará que você insira uma mensagem.
Se tudo funcionar corretamente, o servidor exibirá sua mensagem no stdout, enviará uma mensagem de confirmação ao cliente e será encerrada.
O cliente imprimirá a mensagem de confirmação do servidor e será encerrada.

Você pode simular isso em uma única máquina executando o servidor em uma janela e o cliente em outra. Nesse caso, você pode usar a palavra-chave `localhost`como o primeiro argumento para o cliente.

O código do servidor usa várias construções de programação feias e, portanto, vamos analisá-lo linha por linha.

```
#include <stdio.h>
```

Esse arquivo de cabeçalho contém declarações usadas na maioria das entradas e saídas e geralmente é incluído em todos os programas em C.

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

Este arquivo de cabeçalho contém definições de vários tipos de dados usados em chamadas do sistema. Esses tipos são usados nos próximos dois arquivos de inclusão.

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

O arquivo de cabeçalho socket.h inclui várias definições de estruturas necessárias para soquetes.

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

O arquivo de cabeçalho in.h contém constantes e estruturas necessárias para endereços de domínio da Internet.

---

erro nulo (char * msg)

```
{
  erro (msg);
  saída (1);
}
```

Esta função é chamada quando uma chamada do sistema falha. Ele exibe uma mensagem sobre o erro `stderre` aborta o programa. A página de manual perror fornece mais informações.

```
int main (int argc, char * argv [])
{
  int sockfd, newsockfd, portno, clilen, n;
```

`sockfd`e `newsockfd`são descritores de arquivo, ou seja, subscritos de matriz na tabela de descritores de arquivo . Essas duas variáveis armazenam os valores retornados pela chamada do sistema de soquete e pela chamada de sistema aceita.

`portno` armazena o número da porta na qual o servidor aceita conexões.

`clilen`armazena o tamanho do endereço do cliente. Isso é necessário para a chamada do sistema de aceitação.

né o valor de retorno para as chamadas `read()`e `write()`; isto é, contém o número de caracteres lidos ou gravados.

```
buffer de char [256];
```

O servidor lê caracteres da conexão do soquete nesse buffer.

```
struct sockaddr_in serv_addr, cli_addr;
```

A `sockaddr_in`é uma estrutura que contém um endereço na Internet. Essa estrutura é definida em `netinet/in.h`.

Aqui está a definição:

```
struct sockaddr_in
{
  short sin_family; / * deve ser AF_INET * /
  u_short sin_port;
  struct in_addr sin_addr;
  char sin_zero [8]; / * Não usado, deve ser zero * /
};
```

Uma `in_addr`estrutura, definida no mesmo arquivo de cabeçalho, contém apenas um campo, um chamado não assinado por muito tempo `s_addr`.

A variável `serv_addr`conterá o endereço do servidor e `cli_addr`o endereço do cliente que se conecta ao servidor.

```
if (argc <2)
{
   fprintf (stderr, "ERRO, nenhuma porta fornecida
");
   saída (1);
}
```

O usuário precisa passar o número da porta na qual o servidor aceitará conexões como argumento. Este código exibe uma mensagem de erro se o usuário não conseguir fazer isso.

```
sockfd = soquete (AF_INET, SOCK_STREAM, 0);   erro
if (sockfd <0)
("soquete de abertura de ERRO");
```

A `socket()`chamada do sistema cria um novo soquete. São necessários três argumentos. O primeiro é o domínio do endereço do soquete.

Lembre-se de que existem dois domínios de endereço possíveis, o domínio unix para dois processos que compartilham um sistema de arquivos comum e o domínio da Internet para dois hosts na Internet. A constante de símbolo `AF_UNIX`é usada para o primeiro e `AF_INET`para o último (na verdade existem muitas outras opções que podem ser usadas aqui para fins especializados).

The second argument is the type of socket. Recall that there are two choices here, a stream socket in which characters are read in a continuous stream as if from a file or pipe, and a datagram socket, in which messages are read in chunks. The two symbolic constants are `SOCK_STREAM` and `SOCK_DGRAM`.

The third argument is the protocol. If this argument is zero (and it always should be except for unusual circumstances), the operating system will choose the most appropriate protocol. It will choose TCP for stream sockets and UDP for datagram sockets.

The socket system call returns an entry into the file descriptor table (i.e. a small integer). This value is used for all subsequent references to this socket. If the socket call fails, it returns -1.

Nesse caso, o programa exibe uma mensagem de erro e sai. No entanto, é improvável que essa chamada do sistema falhe.

Esta é uma descrição simplificada da chamada de soquete; existem inúmeras outras opções para domínios e tipos, mas essas são as mais comuns. A página do manual socket () possui mais informações.

```
bzero ((char *) & serv_addr, sizeof (serv_addr));
```

A função `bzero()`define todos os valores em um buffer para zero. São necessários dois argumentos, o primeiro é um ponteiro para o buffer e o segundo é o tamanho do buffer. Assim, essa linha é inicializada `serv_addr`em zeros. ----

```
portno = atoi (argv [1]);
```

O número da porta na qual o servidor escutará as conexões é passado como argumento e esta instrução usa a `atoi()`função para converter isso de uma sequência de dígitos em um número inteiro.

```
serv_addr.sin_family = AF_INET;
```

A variável `serv_addr`é uma estrutura de tipo `struct sockaddr_in`. Essa estrutura possui quatro campos. O primeiro campo é `short sin_family`, que contém um código para a família de endereços. Sempre deve ser definido como a constante simbólica `AF_INET`.

```
serv_addr.sin_port = htons (número da porta);
```

O segundo campo de `serv_addr`é `unsigned short sin_port`, que contém o número da porta. No entanto, em vez de simplesmente copiar o número da porta para esse campo, é necessário convertê-lo na ordem de bytes da rede usando a função `htons()`que converte um número de porta na ordem de bytes do host em um número de porta na ordem de bytes da rede.

```
serv_addr.sin_addr.s_addr = INADDR_ANY;
```

O terceiro campo de `sockaddr_in`é uma estrutura do tipo `struct in_addr`que contém apenas um único campo `unsigned long s_addr`. Este campo contém o endereço IP do host. Para o código do servidor, esse sempre será o endereço IP da máquina na qual o servidor está sendo executado e há uma constante simbólica `INADDR_ANY`que obtém esse endereço.

```
if (bind (sockfd, (struct sockaddr *) & serv_addr, sizeof (serv_addr)) <0)
  erro ("ERRO na ligação");
```

A `bind()`chamada do sistema vincula um soquete a um endereço, nesse caso, o endereço do host e o número da porta atuais nos quais o servidor será executado. São necessários três argumentos, o descritor do arquivo de soquete, o endereço ao qual está vinculado e o tamanho do endereço ao qual está vinculado. O segundo argumento é um ponteiro para uma estrutura de tipo `sockaddr`, mas o que é passado é uma estrutura de tipo `sockaddr_in`e, portanto, isso deve ser convertido no tipo correto. Isso pode falhar por vários motivos, sendo o mais óbvio que esse soquete já está em uso nesta máquina. O manual bind () tem mais informações.

```
ouvir (sockfd, 5);
```

A `listen`chamada do sistema permite que o processo escute no soquete as conexões. O primeiro argumento é o descritor de arquivo de soquete e o segundo é o tamanho da fila de pendências, ou seja, o número de conexões que podem estar aguardando enquanto o processo está lidando com uma conexão específica. Isso deve ser definido como 5, o tamanho máximo permitido pela maioria dos sistemas. Se o primeiro argumento for um soquete válido, essa chamada não poderá falhar e, portanto, o código não verificará erros. A página do manual listen () tem mais informações.

```
clilen = sizeof (cli_addr);
newsockfd = accept (sockfd, (struct sockaddr *) & cli_addr, & clilen);   erro
if (newsockfd <0)
("ERRO ao aceitar");
```

A `accept()`chamada do sistema faz com que o processo seja bloqueado até que um cliente se conecte ao servidor. Assim, ele ativa o processo quando uma conexão de um cliente foi estabelecida com êxito. Ele retorna um novo descritor de arquivo e toda a comunicação nessa conexão deve ser feita usando o novo descritor de arquivo. O segundo argumento é um ponteiro de referência para o endereço do cliente na outra extremidade da conexão, e o terceiro argumento é o tamanho dessa estrutura. A página do manual accept () tem mais informações.

```
bzero (buffer, 256);
n = leitura (newsockfd, buffer, 255);
if (n <0) erro ("ERRO lendo do soquete");
printf ("Aqui está a mensagem:% s
", buffer);
```

Observe que somente chegaríamos a esse ponto depois que um cliente se conectasse com êxito ao nosso servidor. Esse código inicializa o buffer usando a `bzero()`função e depois lê do soquete. Observe que a chamada de leitura usa o novo descritor de arquivo, aquele retornado por `accept()`, não o descritor original do arquivo retornado por `socket()`. Observe também que o `read()`bloco será bloqueado até que haja algo para ler no soquete, ou seja, após o cliente ter executado a `write()`.

Ele lerá o número total de caracteres no soquete ou 255, o que for menor, e retornará o número de caracteres lidos. A página do manual read () tem mais informações.

```
n = write (newsockfd, "recebi sua mensagem", 18);
if (n <0) error ("ERRO gravando no soquete");
```

Depois que uma conexão é estabelecida, as duas extremidades podem ler e gravar na conexão. Naturalmente, tudo o que é escrito pelo cliente será lido pelo servidor e tudo o que for escrito pelo servidor será lido pelo cliente. Esse código simplesmente grava uma mensagem curta no cliente. O último argumento da gravação é o tamanho da mensagem. A página do manual write () tem mais informações.

```
  retornar 0;
}
```

Isso encerra o main e, portanto, o programa. Como main foi declarado como do tipo int, conforme especificado pelo padrão ascii, alguns compiladores reclamam se ele não retorna nada.

Código do cliente

Como antes, percorreremos o programa `client.c`linha por linha.

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

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

Os arquivos de cabeçalho são os mesmos do servidor com uma adição. O arquivo `netdb.h`define a estrutura `hostent`, que será usada abaixo.

```
erro nulo (char * msg)
{ erro
  (msg);
  saída (0);
}
int main (int argc, char * argv [])
{
  int sockfd, portno, n;
  struct sockaddr_in serv_addr;
  struct hostent * server;
```

A `error()`função é idêntica à do servidor, assim como as variáveis `sockfd, portno`, e `n`. A variável `serv_addr`conterá o endereço do servidor ao qual queremos nos conectar. É do tipo [struct sockaddr_in](#) .

A variável `server`é um ponteiro para uma estrutura do tipo `hostent`. Essa estrutura é definida no arquivo de cabeçalho da `netdb.h`seguinte maneira:

```
struct hostent
{
  char * h_name; / * nome oficial do host * /
  char ** h_aliases; / * lista de alias * /
  int h_addrtype; / * tipo de endereço do host * /
  int h_length; / * comprimento do endereço * /
  char ** h_addr_list; / * lista de endereços do servidor de nomes * /
  #define h_addr h_addr_list [0] / * endereço, para compatibilidade com versões anteriores * /
};
```

Ele define um computador host na Internet. Os membros dessa estrutura são:

```
h_name Nome oficial do host.
h_aliases Uma matriz terminada com zero de
             nomes alternativos para o host.
h_addrtype O tipo de endereço que está sendo retornado;
             atualmente sempre AF_INET.
h_length O comprimento, em bytes, do endereço.
h_addr_list Um ponteiro para uma lista de endereços de rede
             para o host nomeado. Os endereços de host são
             retornados na ordem de bytes da rede.
```

Observe que `h_addr`é um alias para o primeiro endereço na matriz de endereços de rede.

```
buffer de char [256];
if (argc <3)
{
  fprintf (stderr, "use% s hostname port
", argv [0]);
  saída (0);
}
portno = atoi (argv [2]);
sockfd = soquete (AF_INET, SOCK_STREAM, 0);   erro
if (sockfd <0)
("soquete de abertura de ERRO");
```

Todo esse código é igual ao do servidor.

```
servidor = gethostbyname (argv [1]);
if (server == NULL)
{
  fprintf (stderr, "ERRO, esse host não existe
");
  saída (0);
}
```

A variável `argv`[1] contém o nome de um host na Internet, por exemplo `cs.rpi.edu`. A função:

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

Pega esse nome como argumento e retorna um ponteiro para uma `hostent`informação que contém esse host.

O campo `char *h_addr`contém o endereço IP.

Se essa estrutura for NULL, o sistema não pôde localizar um host com esse nome.

Antigamente, essa função funcionava pesquisando um arquivo de sistema chamado, `/etc/hosts`mas com o crescimento explosivo da Internet, tornou-se impossível para os administradores de sistema manter esse arquivo atualizado. Assim, o mecanismo pelo qual essa função funciona é complexo, geralmente envolve a consulta de grandes bancos de dados em todo o país. A [página do manual gethostbyname ()](#) possui mais informações.

```
bzero ((char *) & serv_addr, sizeof (serv_addr));
serv_addr.sin_family = AF_INET;
bcopy ((char *) servidor-> h_addr,
      (char *) & serv_addr.sin_addr.s_addr,
      servidor-> h_length);
serv_addr.sin_port = htons (número da porta);
```

Este código define os campos em `serv_addr`. Muito disso é o mesmo que no servidor. No entanto, como o campo `server->h_addr`é uma cadeia de caracteres, usamos a função:

```
void bcopy (char * s1, char * s2, comprimento int)
```

que copia `length`bytes de `s1`para `s2`. ----

```
  erro if (connect (sockfd, & serv_addr, sizeof (serv_addr)) <0) ("conexão de erro");
```

A `connect`função é chamada pelo cliente para estabelecer uma conexão com o servidor. São necessários três argumentos, o descritor do arquivo de soquete, o endereço do host ao qual deseja se conectar (incluindo o número da porta) e o tamanho desse endereço. Esta função retorna 0 em caso de sucesso e -1 se falhar. A página do manual connect () tem mais informações.

Observe que o cliente precisa saber o número da porta do servidor, mas não precisa saber seu próprio número de porta. Isso geralmente é atribuído pelo sistema quando `connect`é chamado.

```
  printf ("Digite a mensagem:");
  bzero (buffer, 256);
  fgets (buffer, 255, stdin);
  n = gravação (sockfd, buffer, strlen (buffer));
  if (n <0)
    error ("ERRO gravando no soquete");
  bzero (buffer, 256);
  n = leitura (sockfd, buffer, 255);
  if (n <0)
    erro ("ERRO lendo do soquete");
  printf ("% s
", buffer);
  retornar 0;
}
```

O código restante deve ser bastante claro. Ele solicita que o usuário insira uma mensagem, usa `fgets`para ler a mensagem de stdin, grava a mensagem no soquete, lê a resposta do soquete e exibe essa resposta na tela.

Aprimoramentos no código do servidor

The sample server code above has the limitation that it only handles one connection, and then dies. A "real world" server should run indefinitely and should have the capability of handling a number of simultaneous connections, each in its own process. This is typically done by forking off a new process to handle each new connection.

The following code has a dummy function called `dostuff(int sockfd)`.

This function will handle the connection after it has been established and provide whatever services the client requests. As we saw above, once a connection is established, both ends can use `read` and `write` to send information to the other end, and the details of the information passed back and forth do not concern us here.

To write a "real world" server, you would make essentially no changes to the main() function, and all of the code which provided the service would be in `dostuff()`.

To allow the server to handle multiple simultaneous connections, we make the following changes to the code:

1. Put the accept statement and the following code in an infinite loop.
2. After a connection is established, call `fork()`#### to create a new process.
3. The child process will close `sockfd`#### and call `#dostuff`#####, passing the new socket file descriptor as an argument. When the two processes have completed their conversation, as indicated by `dostuff()`#### returning, this process simply exits.
4. The parent process closes `newsockfd`####. Because all of this code is in an infinite loop, it will return to the accept statement to wait for the next connection.

Here is the code.

```
while (1)
{
  newsockfd = accept(sockfd,
              (struct sockaddr *) &cli_addr, &clilen);
  if (newsockfd < 0)
    error("ERROR on accept");
  pid = fork();
  if (pid < 0)
    error("ERROR on fork");
  if (pid == 0)
  {
    close(sockfd);
    dostuff(newsockfd);
    exit(0);
  }
  else
    close(newsockfd);
} /* end of while */
```

Clique aqui para obter um programa completo de servidor que inclua essa alteração. Isso será executado com o programa client.c.

O problema dos zumbis

The above code has a problem; if the parent runs for a long time and accepts many connections, each of these connections will create a zombie when the connection is terminated. A zombie is a process which has terminated but but cannot be permitted to fully die because at some point in the future, the parent of the process might execute a `wait` and would want information about the death of the child. Zombies clog up the process table in the kernel, and so they should be prevented. Unfortunately, the code which prevents zombies is not consistent across different architectures. When a child dies, it sends a SIGCHLD signal to its parent. On systems such as AIX, the following code in `main()` is all that is needed.

```
signal(SIGCHLD,SIG_IGN);
```

Isto diz para ignorar o sinal SIGCHLD. No entanto, em sistemas executando o SunOS, você deve usar o seguinte código:

```
void * SigCatcher (int n)
{
  wait3 (NULL, WNOHANG, NULL);
}
...
int main ()
{
  ...
  sinal (SIGCHLD, SigCatcher);
  ...
```

A função `SigCatcher()`será chamada sempre que os pais receberem um sinal SIGCHLD (ou seja, sempre que uma criança morrer). Por sua vez, isso chamará `wait3`que receberá o sinal. O sinalizador WNOHANG está definido, o que faz com que seja uma espera sem bloqueio (um dos meus oxímoro favoritos).

Tipos alternativos de tomadas

Este exemplo mostrou um soquete de fluxo no domínio da Internet. Esse é o tipo mais comum de conexão. Um segundo tipo de conexão é um soquete de datagrama. Você pode usar um soquete de datagrama nos casos em que apenas uma mensagem está sendo enviada do cliente para o servidor e apenas uma mensagem sendo devolvida. Existem várias diferenças entre um soquete de datagrama e um soquete de fluxo.

1. Os datagramas não são confiáveis, o que significa que, se um pacote de informações for perdido em algum lugar da Internet, o remetente não será informado (e, é claro, o destinatário não sabe sobre a existência da mensagem). Por outro lado, com um soquete de fluxo, o protocolo TCP subjacente detectará que uma mensagem foi perdida porque não foi reconhecida e será retransmitida sem o processo nos dois lados sabendo disso.
2. Os limites da mensagem são preservados nos soquetes de datagramas. Se o remetente enviar um datagrama de 100 bytes, o receptor deverá ler todos os 100 bytes de uma só vez. Isso pode ser contrastado com um soquete de fluxo, onde, se o remetente escrever uma mensagem de 100 bytes, o destinatário poderá lê-la em dois pedaços de 50 bytes ou 100 pedaços de um byte.
3. A comunicação é feita usando as chamadas especiais do sistema `sendto()####` e `receivefrom()####`, em vez dos mais genéricos `read()####` e `write()####`.
4. Há muito menos sobrecarga associada a um soquete de datagrama porque as conexões não precisam ser estabelecidas e quebradas, e os pacotes não precisam ser reconhecidos. É por isso que os soquetes de datagrama costumam ser usados quando o serviço a ser prestado é curto, como um serviço no horário do dia.

Clique aqui para obter o código do servidor usando um soquete de datagrama.

Clique aqui para o código do cliente usando um soquete de datagrama.

Esses dois programas podem ser compilados e executados exatamente da mesma maneira que o servidor e o cliente usando um soquete de fluxo.

A maior parte do código do servidor é semelhante ao código do soquete do fluxo. Aqui estão as diferenças.

```
meia = soquete (AF_INET, SOCK_DGRAM, 0);
```

Observe que quando o soquete é criado, o segundo argumento é a constante simbólica SOCK_DGRAM em vez de SOCK_STREAM. O protocolo será UDP, não TCP. ----

```
fromlen = sizeof (struct sockaddr_in);
while (1)
{
  n = recvfrom (sock, buf, 1024,0, (struct sockaddr *) & from, & fromlen);
  erro if (n <0) ("recvfrom");
```

Servidores que usam soquetes de datagrama não usam `listen()`ou o `accept()`sistema chama. Depois que um soquete é vinculado a um endereço, o programa chama `recvfrom()`para ler uma mensagem. Essa chamada será bloqueada até que uma mensagem seja recebida. A `recvfrom()`chamada do sistema recebe seis argumentos. Os três primeiros são iguais aos da `read()`chamada, o descritor do arquivo de soquete, o buffer no qual a mensagem será lida e o número máximo de bytes. O quarto argumento é um argumento inteiro para sinalizadores. Normalmente, isso é definido como zero. O quinto argumento é um ponteiro para uma estrutura sockaddr_in. Quando a chamada retornar, os valores dessa estrutura serão preenchidos para a outra extremidade da conexão (o cliente). O tamanho dessa estrutura estará no último argumento, um ponteiro para um número inteiro. Essa chamada retorna o número de bytes na mensagem. (ou -1 em uma condição de erro). A página do manual recfrom () tem mais informações.

```
  n = sendto (meia, "Recebi sua mensagem
", 17,
          0, (struct sockaddr *) & from, fromlen);     erro
  if (n <0)
("sendto");
```

```
  }
}
```

Para enviar um datagrama, a função `sendto()`é usada. Isso também leva seis argumentos. Os três primeiros são iguais aos de uma `write()`chamada, o descritor do arquivo de soquete, o buffer no qual a mensagem será gravada e o número de bytes a serem gravados. O quarto argumento é um argumento int chamado flags, que normalmente é zero. O quinto argumento é um ponteiro para uma `sockadd_in`estrutura. Isso conterá o endereço para o qual a mensagem será enviada. Observe que, nesse caso, como o servidor está respondendo a uma mensagem, os valores dessa estrutura foram fornecidos pela chamada de retorno. O último argumento é o tamanho dessa estrutura. Observe que esse não é um ponteiro para um int, mas um valor int em si. A página do manual sendto () possui mais informações.

O código do cliente para um cliente de soquete de datagrama é o mesmo que para um soquete de fluxo com as seguintes diferenças.

- a chamada do sistema de soquete tem SOCK_DGRAM em vez de SOCK_STREAM como seu segundo argumento.
- não há `connect()`chamada de sistema ****
- em vez de `read`**** e `write`****, o cliente usa `recvfrom`**** e `sendto` ****, que são descritos em detalhes acima.

Soquetes no domínio Unix

Aqui está o código para um cliente e servidor que se comunica usando um soquete de fluxo no domínio Unix.

U_server.c

U_client

A única diferença entre um soquete no domínio Unix e um soquete no domínio da Internet é a forma do endereço. Aqui está a estrutura de endereço para um endereço de domínio Unix, definido no arquivo de cabeçalho .

```
struct sockaddr_un
{
short sun_family; / * AF_UNIX * /
char sun_path [108]; / * nome do caminho (gag) * /
};
```

O campo `sun_path`tem a forma de um nome de caminho no sistema de arquivos Unix. Isso significa que o cliente e o servidor precisam estar executando o mesmo sistema de arquivos. Depois que um soquete é criado, ele permanece até ser excluído explicitamente, e seu nome aparecerá com o `ls`comando, sempre com tamanho zero. Soquetes no domínio Unix são praticamente idênticos aos pipes nomeados (FIFOs).

Projetando Servidores

Existem várias maneiras diferentes de projetar servidores. Esses modelos são discutidos em detalhes em um livro de Douglas E. Comer e David L. Stevens intitulado *Internetworking with TCP / IP Volume III: Programação e Aplicativos de Servidor Cliente* publicado por Prentice Hall em 1996. Estes são resumidos aqui.

Servidores simultâneos orientados a conexão

O servidor típico no domínio da Internet cria um soquete de fluxo e inicia um processo para lidar com cada nova conexão que recebe. Esse modelo é apropriado para serviços que farão muita leitura e gravação por um longo período de tempo, como um servidor telnet ou um servidor ftp. Esse modelo possui uma sobrecarga relativamente alta, porque iniciar um novo processo é uma operação demorada e porque um soquete de fluxo que usa o protocolo TCP possui uma sobrecarga alta do kernel, não apenas no estabelecimento da conexão, mas também na transmissão de informações. No entanto, uma vez estabelecida a conexão, a transmissão de dados é confiável em ambas as direções.

Servidores iterativos e sem conexão

Servidores que fornecem apenas uma única mensagem ao cliente geralmente não envolvem bifurcação e geralmente usam um soquete de datagrama em vez de um soquete de fluxo. Os exemplos incluem um daemon de dedo ou um servidor no horário do dia ou um servidor de eco (um servidor que apenas faz eco de uma mensagem enviada pelo cliente). Esses servidores manipulam cada mensagem conforme as recebe no mesmo processo. Há muito menos sobrecarga nesse tipo de servidor, mas a comunicação não é confiável. Uma solicitação ou resposta pode se perder na Internet e não há mecanismo interno para detectar e lidar com isso.

Servidores simultâneos de processo único

Um servidor que precisa da capacidade de lidar com vários clientes simultaneamente, mas onde cada conexão é dominada por E / S (ou seja, o servidor passa a maior parte do tempo bloqueado aguardando uma mensagem do cliente) é candidato a um único servidor simultâneo de processo. Nesse modelo, um processo mantém um número de conexões abertas e escuta cada uma delas para uma mensagem. Sempre que recebe uma mensagem de um cliente, ele responde rapidamente e, em seguida, ouve a próxima. Este tipo de serviço pode ser feito

com a `select`chamada do sistema.

classifique este artigo:
classificação atual **:** classificação média: 1.2 (29233 votos) (1 = muito bom 6 = terrível)
Sua classificação:
Muito bom (1) Bom (2) ok (3) médio (4) ruim (5) terrible (6) enviar sua classificação

voltar
Suporte este site