Tutorial de Sockets - Parte I

Por: Frederico Perim

O que é um Socket?

Você já deve ter ouvido faler sobre Sockets e talvez esteja imaginando do que se trata exatamente.

Bem, resumindo: através de Sockets nos comunicamos com outros programas usando discritores de arquivos Unix padrão. O que? Tudo bem. Você deve ter ouvido algum guru em Unix dizer,"Tudo no Unix é um arquivo!". O que esta pessoa esta querendo dizer, na verdade, é o fato de que quando sistemas Unix realizam qualquer tipo de E/S(Entrada/Saída), eles o fazem lendo ou escrevendo através de um descritor de arquivo.Um descritor de arquivo é simplesmente um inteiro associado a um arquivo aberto.Mas(e aqui está o detalhe), este arquivo pode ser uma conexão de rede,um FIFO, um pipe, um terminal, um arquivo no disco, ou qualquer outra coisa.Tudo no Unix(Linux) é um arquivo!.Assim quando você quiser se comunicar com outro programa na internet você vai usar um descritor de arquivo.Apartir dai, você realiza chamadas de socket send() e recv().

"Mas,peraí!", você deve estar pensando. "Se é um descritor de arquivo, porque não usar as chamadas read() e write() para se comunicar através de um Socket?"a resposta é,"Você pode!"a resposta completa é,"Você pode, mas send() e recv() oferecem melhor controle sobre a transmissão de dados".

"Tá bom, e daí?"Que tal isto: existem vários tipos de Sockets.Tem o DARPA(Sockets de Internet), Sockets Unix, CCITT X.25 e provavelmente outros dependendo da sua versão Unix. Este artigo abordará somente o primeiro: Sockets de Internet.

Dois Tipos de Sockets

O que é isto? Existem dois tipos de Sockets de Internet? Sim. Bem,na verdade não.Tem mais, mas não quero assustá-lo. Irei abordar apenas dois tipos.O primeiro é o "Stream Sockets", o outro "Datagram Sockets", que apartir de agora serão chamados de "SOCK_STREAM" e "SOCK_DGRAM", respectivamente. Os "Datagram Sockets" também são conhecidos como sockets sem conexão, embora você possa usar a chamada connect().

Os "Stream Sockets" são fluxos de comunicação confiáveis. Se você enviar 2 itens na ordem "1,2", eles irão chegar chegar na ordem "1,2" no outro extremo do link.Eles também são livres de erros.Qualquer erro que você encontrar é apenas ilusão da sua mente :-).

O que realmente utiliza um "Stream Socket"? Você já deve ter ouvido falar no programa Telnet.Pois bem, todos os caracters que você digita precisam chegar na mesma ordem em que você os digitou logo este aplicativo consegue isto através de "Stream Sockets".Além disso os browsers, que utilizam o protocolo HTTP, usam "Stream Sockets" para receber páginas.

Como os "Stream Sockets" conseguem este alto grau de qualidade de transmissão de dados? Eles utilizam um protocolo chamado "Protocolo de Controle de Transmissão", mais conhecido como "TCP". TCP assegura que os dados chegarão sequencialmente e livres de erros. Você deve ter ouvido falar que TCP é a melhor metade do TCP/IP, onde IP significa Protocolo de Internet. IP cuida basicamente do roteamento e não é responsável pela integridade dos dados.

Legal. E os "Datagram Sockets"? Porque são conhecidos como sem conexão? Porque não são confiáveis? Bem, alguns fatos: Se você enviar um datagrama, ele pode chegar. Ele pode chegar fora de ordem. Se chegar , os dados contidos no pacote estarão livres de erros.

"Datagram Sockets" também usam o IP para roteamento, mas eles não utilizam TCP, e sim o "UDP".

Porque são sem conexão? Bem , basicamente devido ao fato de você não precisar manter uma conexão aberta como os "Stream Sockets". Você constrói um pacote, anexa um cabeçalho IP com informações de destino, e envia. Não precisa haver conexão. Ele são mais utilizados para transferências pacote por pacote de informações. Alguns aplicativos que utilizam UDP: tftp, bootp,etc.

"Chega!", você diz. "Como estes programas funcionam se um pacote pode se perder na rota?"Bem, meu caro amigo, cada um tem se próprio protocolo acima do UDP. Por exemplo, o protocolo tftp diz que para cada pacote que é enviado, o receptor tem que enviar um pacote de volta que diz, "Eu recebi!"(um pacote "ACK") Se o emissor do pacote original não receber uma resposta, digamos, em cinco segundos , ele irá retransmitir o pacote até que receba um "ACK". Este procedimento é muito importante quando você for implementar aplicações que utilizam "SOCK_DGRAM".

Alguma Teoria de Rede

Já que mencionei algumas camadas de protocolos, é hora de falar como as redes realmente funcionam, e mostrar alguns exemplos de como pacotes SOCK_DGRAM são construídos. Você pode pular esta parte se não estiver interessado.

É hora de aprender sobre Encapsulamento de Dados. Isto é muito importante. Basicamente, é isto: um pacote é criado, o pacote é empacotado("encapsulado") em um cabeçalho pelo primeiro protocolo (digamos, o protocolo TFTP), então tudo (cabeçalho TFTP incluído) é empacotado novamente pelo próximo protocolo (por exemplo, UDP), novamente pelo próximo (IP) , e então finalmente pelo último protocolo no hardware(camada física) , por exemplo Ethernet.

Quando outro computador receber o pacote, o hardware retira o cabeçalho Ethernet, o kernel retira os cabeçalhos IP e UDP, e o programa TFTP retira o cabeçalho TFTP, e finalmente se tem os dados.

Agora finalmente posso falar sobre o temível Modelo de Camadas de Rede. Este modelo descreve um sistema de funcionalidade de rede que tem muitas vantagens sobre outros modelos. Por exemplo, você pode escrever programas de sockets que não se importam como os dados são fisicamente transmitidos (serial, thin ETHERNET,AUI, seja lá o que for) porque programas nos níveis mais baixos tomam conta disto pra você.

O hardware de rede e a topologia são transparentes para o programador.

Bem sem mais enrolação, irei apresentar o modelo completo:

- Camada de Aplicação (Application Layer)
- Camada de Transporte (TCP,UDP)
- Camada de Internet (IP)
- Camada de Acesso de Rede (Ethernet, ATM)

Nesse momento você provavelmente deve estar imaginando como estas camadas correspondem ao encapsulamento dos dados.
Veja quanto trabalho para construir um simples pacote? E você ainda tem que digitar os cabeçalhos do pacote usando "cat"!. Brincadeirinha. Tudo que você tem que fazer para utilizar "Sockets Streams" é enviar (send()) os dados. Com relação aos "Sockets Datagram" é empacotar os pacotes num método de sua escolha e enviar(sendto()) os dados. O kernel constrói a Camada de Transporte e a Camada de Internet pra você e o hardware cuida das camadas mais baixas.
Bem aqui termina nossa breve introdução em teoria de redes. AH, esqueci de mencionar alguma coisa sobre roteamento: nada! Isso mesmo , não irei falar sobre isso. O roteador retira o cabeçalho IP, consulta a tabela de roteamento , etc,etc.

Estruturas e Manipulação de Dados

Bem finalmente é hora de falar sobre programação. Nesta parte , vou cobrir alguns tipos de dados usados pelas interfaces de sockets, já que algumas são um pouco complicadas.

Primeiro a mais fácil: um descritor de socket. O descritor de socket é do seguinte tipo:

int

Apenas um simples int.

As coisas começam a se complicar daqui em diante, então vamos lá. Saiba isto: existem dois tipos de ordenamentos de byte: primeiro byte mais significante(também chamados de "octet"), e primeiro byte menos significante, chamado de "Network Byte Order". Algumas máquinas armazenam números internamente em Network Byte Order, algumas não. Quando digo que algo deve estar em "Network Byte Order", você tem que chamar uma função (htons()) para mudar de "Host Byte Order". Se não for dito "Network Byte Order" , então você pode deixar como "Host Byte Order".

A primeira estrutura contém informações sobre tipos de sockets:

```
strut sockaddr {
unsigned short sa_family; // tipo de endereço, AF_xxx
char sa_data[14]; // 14 bytes do endereço de protocolo
};
```

sa_family pode ser uma variedade de coisas, mas será AF_INET para tudo que fizermos neste documento. sa_data contém um endereço de destino e número de porta para o socket. Isto é um tanto confuso, já que não queremos empacotar o endereço sa_data manualmente.
Para tratar de strut sockaddr, programdores criaram uma estrutura paralela: struct sockaddr_in("in" para "internet")

```
struct sockaddr_in {
short int sin_family; // tipo de endereço
unsigned short int sin_port; //Número da porta
struct in_addr sin_addr; //Endereço da Internet
unsigned char sin_zero[8]; //Mesmo tamanho de struct sockaddr
};
```

Esta estrutura torna mais fácil referenciar elementos de um endereço de socket. Note que sin_zero (que é incluído para acomadar a estrutura ao tamanho de struct sockaddr) deve ser setada para zeros através da função memset(). Ainda, e isto é muito importante, um ponteiro para um struct sockaddr_in pode ser convertido para um struct sockaddr e vice-versa. Assim mesmo que socket() queira um struct sockaddr*, você pode ainda usar um struct sockaddr_in e converter no momento apropriado. Note também que sin_family corresponde a sa_family em strutct sockaddr e deve ser setado para "AF_INET". Finalmente ,sin_port e sin_addr devem estar em "Network Byte Order".
"Mas um momento! Como pode toda estrutura, struct in_addr sin_addr, estar em Network Byte Order?" Esta questão requer uma análise cuidadosa da estrutura struct in_addr :

```
// Endereço de Internet
struct in_addr {
unsigned long s_addr;
};
```

Então se você declarou ina para ser do tipo struct sockaddr_in, então ina.sin_addr.s_addr referencia um endereço IP de 4 bytes (em "Network Byte Order").

Convertendo Valores

Houve muito falatório sobre conversão para Network Byte Order. Agora é hora de ação!

Tudo bem. Existem dois tipos que você pode converter: um short de Host Byte Order para Network Byte Order. Comece com "h" para "host", seguindo com "to", e então "n" para "network", e "s" para "short" : h-to-n-s ou htons() (leia-se: "Host to Network Short").

Nem é tão difícil assim...

Você pode usar combinações de "n", "h", "s" e "l" que quiser, sem contar as estúpidas. Por exemplo , não existe uma função stolh() ("Short to Long Host") - pelo menos não aqui. Mas existem :

- htons() -- "Host to Network Short"
- htonl() -- "Host to Network Long"
- ntohs() -- "Network to Host Short"
- ntohl() -- "Network to Host Long"

Agora, você deve estar começando a entender. Você pode pensar, "O que eu faço para mudar a ordem de byte em char?" Então você pensa , "Deixa pra lá". Você também deve imaginar que sua máquina 68000 já usa "Network Byte Order" , e não precisa chamar htonl() no seus endereços IP. Você estaria certo, MAS se você tentar portar para uma máquina que tem ordem de byte de rede reverso seu programa não vai rodar. Tente escrever programas portáveis a outras arquiteturas! Este é um mundo Unix (Linux). Lembre-se: pode seus bytes em Network Byte Order antes de botá-los na rede.
Uma último esclarecimento: porque sin_addr e sin_port precisam estar em Network Byte Order em um struc sockaddr_in, mas sin_family não? A resposta: sin_addr e sin_port são encapsulados no pacote nas layers IP e UDP, respectivamente. Assim, eles devem estar em Network Byte Order. No entanto, o campo sin_family não é enviado pela rede, ele pode estar em Host Byte Order.

Endereços IP e como Lidar com eles

Felizmente para você , existem várias funções que te permitem manipular endereços IP. Não é necessário tratá-los manualmente e colocá-los em um long com um operador <<.
Primeiro , digamos que você tem um endereço IP "10.12.110.57" e deseja armazená-lo em um struct sockaddr_in ina. A função que você deseja usar , inet_addr(), converte um endereço IP na notação de números e pontos em um unsigned long. A forma é a seguinte:

```
ina.sin_addr.s_addr = inet_addr("10.12.110.57");
```

Note que int_addr() retorna o endereço em Network Byte Order, você não precisa chamar htonl(). Beleza!!!
Agora, o código acima não é muito robusto porque não existe checagem de erro. inet_addr() retorna -1 em caso de erro. Lebra de números binários? (unsigned) -1 corresponde ao endereço IP 255.255.255.255! Este é o endereço de broadcast!. Lembre-se de checar erros apropriadamente.
Na verdade , há uma forma interface melhor que inet_addr(): é chamada inet_aton() ("aton" significa "ascii to nerwork");

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

E aqui está um exemplo prático: este exemplo fará mais sentido quando abordamos bind() e connect()!!

```
struct sockaddr_in meu_end;
meu_end.sin_family = AF_INET; //host byte order
meu_end.sin_port = htons(MYPORT); //short, network byte order
inet_aton("10.12.110.57", &(meu_end.sin_addr));
memset(&(meu_end.sin_zero), ' \0' , 8); // zera o resto da estrutura
```

inet_aton, diferente de praticamente todas as outras funções relacionadas a sockets, retorna um número diferente de zero em caso de sucesso, e zero caso contrário (Se alguém souber porque , me diga) E o endereço é passado de volta em inp.
Infelizmente , nem todas plataformas implementam inet_aton() assim , embora seu uso seja recomendado, a função inet_addr() será usada neste artigo.
Tudo bem, agora você pode converter um string de endereço IP em suas representações binárias. E a forma inversa? E se você tem um struc in_addr e quer mostrar na notação de pontos e números? Neste caso , você terá que usar a função inet_ntoa() ("ntoa" significa "network to asscii") da seguinte forma:

```
printf ("%s", inet_ntoa(ina.sin_addr));
```

Isto irá mostrar o endereço IP. Note que inet_ntoa() tem um struct in_addr como argumento, não um long. Note também que ela retorna um ponteiro para um char. Isto aponta para um array estaticamente armazenado em inet_ntoa() assim toda vez que você chamar inet_ntoa() ela irá sobrescrever o último endereço IP que você chamou. Por exemplo:

```
char *a1, *a2;
//
//
a1 = inet_ntoa(ina1.sin_addr); isto é 192.168.4.14
a2 = inet_ntoa(ina2.sin_addr); isto é 10.14.110.60
printf ("endereço 1: %s\n", a1);
printf("endereço 2: %s\n", a2);
a saída é:
endereço 1: 10.14.110.60
endereço 2: 10.14.110.60
```

Se você precisar salvar o endereço, use strcpy() em seu próprio array de caracteres.

Conclusão

No próximo artigo desta série nos aprofundaremos nas chamadas de sistemas e portas desconhecidas, como o caso do Bind. Até lá!
Arquivo original em: http://www.ecst.csuchico.edu/~beej/guide/net/

Tutorial de Sockets - Parte II

Por: Frederico Perim

Chamadas de Sistema

Agora iremos abordar as chamadas de sistema que permitem acessar a funcionalidade de rede em um ambiente Linux. Quando você chama uma dessas funções, o kernel assume o controle e realiza todo trabalho pra você automaticamente.

O problema que a maioria das pessoas tem encontrado é na ordem em que chamar essas funções. Neste aspecto as páginas man são inuteis, como você já deve ter percebido. Para ajudar nessa árdua situação, tentarei abordar as chamadas de sistema nas próximas linhas aproximadamente na mesma ordem em que você precisará chamá-las em seus programas.

socket() - Acesso ao Descritor de Arquivo!

Imagino que não posso mais adiar. Vou ter que falar sobre a chamada socket().

```
int socket (int domínio, int tipo, int protocolo);
```

Mas o que são estes argumentos? Primeiro, domínio deve ser setado como "AF_INET", assim como struct sockaddr_in (acima). A seguir, o argumento tipo diz ao kernel qual tipo de socket, ou seja: SOCK_STREAM ou SOCK_DGRAM. Finalmente, ajuste protocolo como "0" para que socket() escolha o protocolo correto baseado no argumento tipo. (Nota: Existem vários domínios, Existem vários tipos . Consulte o manual socket() de seu 1 . Também existe uma forma melhor de conseguir o protocolo. Consulte getprotobyname()).
Socket simplesmente retorna o descritor que você vai usar em chamadas de sistema mais tarde, ou -1 em caso de erro. A variável global errno é setada com o valor de erro.
Legal, agora continue lendo e faça mais algumas chamadas de sistema para isto fazer sentido.

Bind() - Em que porta estou?

Uma vez que tenha um socket, você deve associá-lo a uma porta em sua máquina.(Isso é mais comum quando seu programa ouvir (listen()) conexões remotas em uma porta específica.
O número da porta é usado pelo kernel para ajustar um pacote que estiver recebendo a um determinado processo.Se você for somente usar a função connect(), isto pode não ser necessário.
Aqui está uma pequena sinopse da chamada de sistema bind():

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

sockfd é o descritor de socket retornado por socket(). meu_end é um ponteiro a um struct sockaddr que contém informações de seu endereço (porta e endereço IP). addrlen pode ser setado para sizeof(struct sockaddr).
Opa! Isso é um pouco complicado para absorver de uma vez, então vamos a um exemplo:

```
main()
{
    int sockfd;
    struct sockaddr_in meu_end;
    sockfd = socket(AF_INET, SOCK_STREAM, 0); // realizar checagem de erro
    meu_end.sin_family = AF_INET;         // host byte order
    meu_end.sin_port = htons(3490 );     // converte em short, network byte order
    meu_end.sin_addr.s_addr = inet_addr("10.12.110.57");
    memset(&(meu_end.sin_zero), '\0', 8); // zerar o resto da estrutura
    // não se esqueça de realizar checagem de erro em bind():
    bind(sockfd, (struct sockaddr *)&meu_end, sizeof(struct sockaddr));
}
```

Note que meu_end.sin_port está em "Network Byte Order", assim como meu_end.sin_addr.s_addr. Outra coisa a ser esclarecida é que os arquivos de cabeçalho(header) podem ser diferentes dependendo de seu sistema. Para ter certeza acesse o man de seu sistema.

Bind() - Em que porta estou? (continuação)

Por último, a respeito de bind() , devo mencionar que o processo de obter seu endereço IP e/ou porta pode ser automatizado.

```
meu_end.sin_port = 0; // escolhe uma porta aleatória
meu_end.sin_addr.s_addr = INADDR_ANY;  // utiliza meu endereço IP
```

Veja, ao setar meu_end.sin_port para 0 bind() pade escolher a porta pra você.Da mesma forma, ao setar meu_end.sin_addr.s_addr para INADDR_ANY você faz com que o sistema use o endereço IP da máquina onde o processo está rodando.
Se você é daqueles que notam pequenos detalhes, deve ter percebido que não coloquei INADDR_ANY em "Network Byte Order". No entanto, eu sei que INADDR_ANY é realmente zero. Assim alguns puristas irão dizer que poseria haver uma dimensão paralela onde INADDR_ANY seja, digamos, 12 e meu código não seria tão portável. Então tudo bem.

```
meu_end.sin_port = htons(0); //escolhe uma porta aleatória
meu_end.sin_addr.s_addr = htonl(INADDR_ANY); //usa meu endereço IP
```

Agora o programa está totalmente portável. Eu apenas queria esclarecer, já que a maioria do código que você irá encontrar não se importará rodar INADDR_ANY através de htonl().
bind() também retorna -1 em caso de erro e seta erro para o valor do erro.

Outra coisa a ser mencionada quando você for chamar bind(): não escolha qualquer número de porta. Todas as portas abaixo de 1024 são RESERVADAS(ao menos que você seja super usuário)Você pode ter qualquer porta acima disso até 65535 (contanto que não estejam sendo utilizadas por outro programa).
Algumas vezes você pode notar que ao tentar rodar o servidor, bind() falha, dizendo: "Address already in use", ou endereço já em uso. O que isso significa? Bem, um socket que foi conectado ainda está pendente no kernel e está bloqueando a porta. Você pode esperar(um minuto ou mais), ou adicionar código a seu programa permitindo reutilizar a porta da seguinte forma:

```
int pos = 1;
// anula a mesagem de erro "Address already in use"
if
(setsockopt(listener,SOL_SOCKET,SO_REUSEADDR,&pos,sizeof(int)) == -1) {
perror("setsockopt");
exit(1);
}
```

Mais um pequeno detalhe sobre bind(): em determinados momentos você não vai precisar chamá-lo. Se estivar se conectando (connect()) a uma máquina remota você não precisa se preocupar qual é sua porta local, simplesmente chame connect(), e ela irá verificar se existe algum socket conectado, e irá utilizar bind() em uma porta local que não estiver sendo utilizada, caso necessário.

connect() - Já era hora!

Vamos imaginar por alguns minutos que você seja uma aplicação telnet. Seu usário ordena para que você retorne um descritor socket.Você atende e chama socket(). A seguir, o usuário lhe diz para conectar em "10.12.110.57" na porta "23" (a porta padrão telnet). O que você faz agora?

Para sua sorte, programa, você está na seção sobre connect(), como se conectar a host remoto. Então vamos lá!

A chamada connect() tem o seguinte protótipo:

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

sockfd é nosso descritor de socket, retornado pela chamada socket(), serv_addr é uma struct sockaddr contendo informações de porta e endereço IP, e addrlen pode ser sizeof(struct sockaddr).
As coisas não estão começando a fazer sentido? Vamos a um exemplo:

```
main()
{
  int sockfd;
  struct sockaddr_in end_dest; //aqui irá conter o endereço de destino
  sockfd = socket(AF_INET, SOCK_STREAM, 0); //realize checagem de erro!
  end_dest.sin_family = AF_INET; // host byte order
  end_dest.sin_port = htons(POSTA_DEST); //converte para short, network byte order
  end_dest.sin_addr.s_addr = inet_addr("10.12.110.57" );
  memset(&(end_dest.sin_zero), '\0', 8); //zera o resto da estrutura
  //não esqueça de checar potenciais erros em connect()
  connect(sockfd, (struct sockaddr *)&end_dest, sizeof(struct sockaddr));
  .
  .
  .
```

Novamente tenha certeza de checar o valor de retorno de connect()- irá retorna -1 em caso de erro e setar a variável errno.
Note também que não chamamos bind(). Basicamente, não nos importamos com nossa porta local, somente aonde iremos (host remoto). O kernel escolhe um porta local, e o host ao qual nos conectamos obtém esta informação automaticamente

Tutorial de Sockets - Parte III

Por: Frederico Perim

listen() - Tem alguém me chamando?

Hora de mudar um pouco o rumo da prosa. E se você não quiser se conectar a um host. Digamos que você queira esperar por conexões remotas e tratá-las de alguma forma. É um processo de dois passos: primeiro você escuta(listen()), e então aceita (accept()) a conexão.

A chamada listen() é muito simples, mas requer um pouco de explicação:

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

sockfd é o já conhecido descritor de socket da chamada socket(). backlog é o número de conexões permitidas. O que isso significa? Bem , as conexões ficarão na fila (queue) até que você as aceite(accept()) e isto é o limite de quantas conexões poderão ficar na fila. A maioria dos sistmas limita este número para cerca de 20, mas você pode tranquilamente usar 5 ou 10.
Novamente, como sempre, listen(0 retorna -1 em caso de erro e seta errno.
Bem, como você deve ter imaginado, precisamos chamar bind() antes de listen() ou o kernel irá escutar em uma porta remota. Então se você for escutar por conexões remotas, a sequência de chamadas de sistema que você vai executar é:

```
socket();
bind();
listen();
/* accept() vem aqui */
```

Eu vou deixar assim ao invés de código fonte, já que é bastante elucidativo.(O código na seção accept() abaixo. é mais completo.)A parte mais importante da coisa é a chamada accept().

4.5. accept() - "Obrigado por chamar a porta 3490."

Prepare-se - a chamada accept() é um tanto estranha! O que vai acontecer é o seguinte: alguém em algum lugar bem distante vai tentar se conectar(connect()) a sua máquina na porta em que você estiver ouvindo(listen()). Está conexão vai ficar na fila (queue) esperando ser aceita(accept()). Você chama accept() e diz para retornar a conexão pendente. A seguir você vai ter um descritor socket novinho para usar nessa conexão. Você ,de uma hora pra outra, tem duas conexões pelo preço de uma! A original ainda está ouvindo na porta especificada e a outra está pronta para enviar(send()) e receber (recv()).
A chamada é da seguinte forma:

```
int accept (int sockfd, void *addr, int *addrlen);
```

sockfd, bem não vou ser repetitivo. addr será geralmente um ponteiro para um struct sockaddr_in. Aqui é onde a informação sobre a conexão vai ficar(e com isso você pode determinar qual host está chamando de qual porta.) addrlen é um variável inteira local que deve ser setada para sizeof(struct sockaddr_in) antes que seu endereço seja passado para accept(). accept() não colocará mais bytes que isto em addr. Se colocar menos, irá mudar o valor de addrlen para refletir essa mudança.
Adivinha? accept() retorna -1 e o resto você já deve saber.

Como antes, isso é muito para absorver de uma vez, então segue um fragmento de código:

```
// a porta que usuários estarão se conectando
//quantas conexões pendentes serão suportadas
main()
{
   int sockfd, novo_fd; //ouve em sockfd, novas conexões em novo_fd
   struct sockaddr_in meu_end; //informação do meu endereço
   struct sockaddr_in outro_end; //informação do endereço do usuário
   int sin_size;
   sockfd = socket(AF_INET, SOCK_STREAM, 0); //checagem de erro
   meu_end.sin_family = AF_INET; //host byte order
   meu_end.sin_port = htons(3490 ); // converte para short, network byte order
   meu_end.sin_addr.s_addr = INADDR_ANY; //preenche com meu IP
   memset(&(meu_end.sin_zero), '\0', 8); //zera o resto da estrutura
   //não esquecer de fazer checagem de erros para essas chamadas
   bind (sockfd, (struct sockaddr *)&meu_end, sizeof(struct sockaddr));
   listen(sockfd, 10 );
   sin_size = sizeof(struct sockaddr_in);
   novo_fd = accept(sockfd, &outro_end, &sin_size);
```

Novamente note que nós usaremos outro_end para todas as chamadas send() e recv().Se você não vai servir novas conexões , você pode fechar (close()) sockfd para cancelar outras conexões na mesma porta, se assim desejar.

send() e recv() - Fale comigo!

Estas duas funções servem para se comunicar através de "SOCK_STREAM". Se você deseja usar datagram sockets ("SOCK_DGRAM") , então você vai usar sendto() e recvfrom(), abaixo.
A chamada send():

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

sockfd é nosso velho conhecido descritor de socket para onde você envia os dados (seja um retornado por socket() ou por accept()). msg é um ponteiro para os dados que você deseja enviar, e len é o tamanho dos dados em bytes. Ajuste flags como 0. (Consulte o manual de send() para mais informações a respeito de flags).
Um código ilustrativo seria:

```
char *msg = "Oi tudo bem?";
int tamanho, bytes_enviados;
.
.
.
tamanho = strlen(msg);
bytes_enviados = send(sockfd, msg, tamanho, 0);
.
.
.
```

send() retorna o número de bytes realmente enviados - isso pode ser menos do que você mandou enviar. Algumas vezes você envia um monte de dados e send() não pode dar conta. Assim ela vai enviar o máximo possível , e confiar que você mande o resto mais tarde. Lembre-se, se o valor retornado por send não for igual ao valor de tamanho, depende de você mandar o resto da string. A boa notícia: se o pacote for pequeno (menor que 1K), a função provavelmente vai dar um jeito de enviar tudo de uma vez.
A função recv é similar em muitos aspectos:

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

sockfd é o descritor de onde você lê os dados, buf é onde você lê as informações contidas em buffer, len é o tamanho máximo de buffer, e flags poder ser 0.
recv() retorna o número de bytes realmente lidos em buffer, ou -1 em caso de erro.
Espere! recv() pode retornar zero. Isso significa somente uma coisa: o host remoto fechou sua conexão! Um valor de retorno 0 é a forma que recv() utiliza para você saber que isso aconteceu.
Ufa! Foi fácil, não? Agora você pode receber e enviar dados através de "stream sockets".

sendto() e recvfrom() -- Fale comigo no estilo DGRAM

"Isso é legal e simples",você diz. "Mas e o datagramas sem conexão?". Sem problemas. Aí vai.

Já que "SOCK_DGRAM" não estão conectados a um host remoto,adivinhe que tipo de informação precisamos dar antes de enviarmos um pacote? Acertou! O endereço de destino! Aqui está o escopo:

```
int sendto (int sockfd, const void *msg, int len , unsigned int
flags, const struct sockaddr *to, int tolen);
```

Como você pode ver, esta chamada é basicamente igual a send() com a adição de dois pedaços de informação. to é um ponteiro para um struct sockaddr (que você provavelmente terá como uma struct sockaddr_in e então vai converter no momento apropriado) que contém o endereço IP e a porta. tolen simplesmente ser sizeof(struct sockaddr).
Assim como send(), sendto() retorna o número de bytes que realmente foram enviados(que novamente pode ser menos do que você queria enviar). ou -1 em caso de erro.
Igualmente similar são recv(0 e recvfrom(). A sinopse de recvfrom() é:

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
struct sockaddr *from, int *fromlen);
```

De novo, isso é como recv() com a adição de alguns campos. from é um ponteiro para um struct sockaddr que será preenchida com o endereço IP e a porta da máquina que enviou os dados. fromlen é um ponteiro para uma variável inteira local que deve ser inicializada como sizeof(struct sockaddr).Quando a função retornar , fromlen vai conter o tamanho do endereço armazenado em from.
recvfrom retorna o número de bytes recebidos , ou -1.
Lembre-se , se você conectar um datagrama(SOCK_DRAM), você pode simplesmente usar send() e recv() para todas suas transações. O pacote em si ainda é um "SOCK_DGRAM" e os pacotes ainda usam UDP, mas a interface socket irá automaticamente acrescentar as informações de origem e destino pra você.

Tutorial de sockets - Parte IV

Por: Frederico Perim

close() e shutdown() - Saia daqui!

Nossa! Você enviou e recebeu dados o dia inteiro, e já está não aguenta mais. Você está pronto para fechar a conexão em um descritor socket. Isso é fácil. Você pode usar a função close() do Unix:

```
close(sockfd);
```

Isso irá evitar escrever ou ler através deste socket. Se alguém tentar se conectar, receberá uma mensagem de erro.
Caso você queira ter mais controle sobre como o socket fecha,você pode usar a função shutdown(). Ela te permite encerrar a conexão em uma determinada direção, ou em ambas(como close()):

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

socfd é o socket que você deseja encerrar, é how pode ser o seguinte:

- 0 - encerra o recebimento;
- 1 - encerra o envio;
- 2 - encerra o recebimento e envio(como close()).

shutdown() retorna 0 em caso de sucesso, e -1 caso contrário.
Se você insistir em usar shutdown() em sockets de datagrama, ele vai fazer com que socket fique indisponível para futuras chamadas recv() e send()(lembre-se de que você pode usá-las caso use connect() em seus "SOCK_DGRAM").
É importante ressaltar que shutdown() não fecha o descritor de arquivo, apenas muda sua usabilidade. Para liberar um descritor socket, você precisa usar close().

getpeername() - Quem é você?

Essa função é fácil.
É tão fácil, que quase não dedico uma seção. Mas enfim, aqui está.
A função getpeername() irá dizer quem está conectado na outra extremidade de um stream socket. A sinopse:

```
int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

sockfd é o descritor do socket conectado, addr é um ponteiro para um struct sockaddr(ou uma struct sockaddr_in) que vai ter informações sobre o outro lado da conexão, e addrlen é um ponteiro para um int, que deve ser inicializado como sizeof(struct sockaddr).
Uma vez que você tenha o endereço, você pode usar inet_ntoa() ou gethostbyaddr() para imprimir ou obter mais informação. Não , você não pode ler o nome de login. (Tá bom, Se outro computador estiver rodando um ident daemon, isso é possível.Mas isso está além do escopo deste documento).

gethostname() - Quem sou eu?

Ainda mais fácil que getpeername() é a função gethostname(). Ela retorna o nome do computador onde o programa está rodando. O nome pode então ser usado por gethostbyname(), abaixo, para determinar o endereço IP da sua máquina local.

O que poderia ser mais divertido? Aqui está o protótipo:

```
int gethostname (char *hostname, size_t size);
```

Os argumentos são simples: hostname é um ponteiro para um array de caracteres que contém o hostname logo após o retorno da função, e size e o tamanho em bytes do array hostname.
A função retorna 0 em caso de sucesso, e -1 caso contrário como sempre.

DNS - Você diz "receita.fazenda.gov.br", eu digo "161.148.231.100"

Caso você não saiba o que significa DNS, simples:"Domain Name Service",ou Serviço de Nome de Domínio. Rapidamente, você diz o endereço no formato que seja legível ao humanos , e o DNS retorna o endereço IP (assim você pode usar bind(),connect(),sendto() , ou o que precisar.)Dessa forma, quando alguém digita:

```
% telnet receita.fazenda.gov.br
```

telnet pode descobrir que precisa se conectar a "161.148.231.100".
Mas como funciona? Você vai usar a função gethostbyname():

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

Como você pode ver, a função retorna um ponteiro para um struct hostent,que tem o seguinte layout:

```
struct hostent {
char *h_name;
char **h_aliases;
int  h_addrtype;
int  h_length;
char **h_addr_list;
};
```

E aqui está a descrição dos campos na struct hostent:

- h_name -- Nome oficial do host
- h_aliases -- Um array terminado em nulo de nomes alternativos do host
- h_addrtype -- O tipo de endereço retornado, geralmente AF_INET
- h_length -- O tamanho do endereço em bytes
- h_addr_list -- Um array terminado em zero de endereços de rede de um host. Endereços de rede estão em "Network Byte Order"
- * h_addr_list[0] -- O primeiro endereço em h_addr_list.

gethostbyname() retorna um ponteiro para um struct hostent preenchido , ou nulo em caso de erro. (Mas errno não é setada, e sim h_errno. Veja herror(), abaixo).
Mas como é usada? Algumas vezes (ao ler manuais), apenas "jogar" a informação no leitor não é suficiente. A função é mais fácil de usar do que parece.

```
/*
** getip -- um demo que procura nomes de host
*/
int main(int argc, char *argv[])
{
   struct hostent *h;
   if (argc != 2) { //checa a linha de comando
      fprintf(stderr, "Uso: getip endereço\n");
      exit(1);
   }
   if ((h=gethostbyname(argv[1])) == NULL) { // pega informação do host
      herror("gethostbyname");
      exit(1);
   }
   printf ("Nome do Host : %s\n", h->h_name);
   printf ("Endereço IP  : %s\n", inet_ntoa(*((struct in_addr *)h->h_addr_list[0] )));
   return 0;
```

```
}
```

Com gethostbyname(), você não pode usar perror() para imprimir a mensagem de erro.Ao invés, chame herror().
É muito simples. Você passa o string que contém o nome da máquina("receita.fazenda.gov.br") para gethostbyname(), e então pega a informação de um struct hostent retornado.
A única coisa estranha deve ser na hora de imprimir o endereço IP, acima. h->h_addr_list[0] é um char*, mas inet_ntoa() precisa receber um struct in_addr. Então eu converto h->h_addr_list[0] para um struct in_addr*,e depois retiro a referência para chegar aos dados.

Tutorial de sockets - Parte V

Por: Frederico Perim

Cliente-Servidor

Vivemos em mundo cliente-servidor. Quase tudo relacionado a rede lida com processos no cliente falando com processos no servidor e vice-versa. Pegue o telnet por exemplo. Quando você conecta na porta 23 com o telnet(cliente), um programa no outro host (chamado telnetd, o servidor) aparece. Ele gerencia a conexão, mostra um prompt de login, etc.

O par cliente-servidor pode se comunicar através de "SOCK_STREAM","SOCK_DGRAM", ou qualquer outra coisa (contanto que estejam falando a mesma coisa).

Alguns exemplos de pares cliente-servidor são: telnet/telned, ftp/ftpd ou bootp/bootpd. Toda vez que você usa o FTP, existe um programa remoto, ftpd, que atende suas conexões.

Geralmente haverá apenas um servidor na máquina, e este servidor vai controlar múltiplos clientes usando fork(). A rotina básica é: o servidor espera a conexão, aceita (accept()) , e então cria um processo filho para gerenciar a conexão (através de fork()). Isto é o que nosso servidor faz na próxima seção.

Um Simples Servidor Stream

Tudo o que este servidor faz é enviar a string "Olá , Beleza!\n" através de uma conexão stream. Tudo o que você precisa para testar este servidor e rodá-lo em uma janela, e usar o telnet em outra.

```
 //  a porta na qual clientes estarão se conectando
 // quantas conexões pendentes serão permitidas
void sigchld_handler(int s)
{
    while(wait(NULL) &gt; 0);
}
int main(void)
{
 int sockfd, nova_fd;  // escutar em fd, novas conexões em  new_fd
 struct sockaddr_in meu_end;    // minha informação de endereço
 struct sockaddr_in outro_end; // informação de endereço externo
 int sin_size;
 struct sigaction sa;
 int yes=1;
 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
        perror("socket");
        exit(1);
}
if
```

```
(setsockopt(sockfd,SOL_SOCKET,SO_REUSEADDR,&yes,sizeof(int)) == -1)
{
            perror("setsockopt");
            exit(1);
          }
          meu_end.sin_family = AF_INET;         // host byte order
          meu_end.sin_port = htons(3490 ); // converte para short, network byte order
          meu_end.sin_addr.s_addr = INADDR_ANY; // automaticamente preenche com meu endereço IP
          bzero(&(meu_end.sin_zero), 8);        // zera o resto da estrutura
          if (bind(sockfd, (struct sockaddr *)&meu_end, sizeof(struct sockaddr)) == -1)
{
                perror("bind");
                exit(1);
          }
          if (listen(sockfd, 10 ) == -1)
{
                 perror("listen");
                 exit(1);
          }
          sa.sa_handler = sigchld_handler; // retira todos os processos ociosos
          sigemptyset(&sa.sa_mask);
          sa.sa_flags = SA_RESTART;
          if (sigaction(SIGCHLD, &sa, NULL) == -1) {
             perror("sigaction");
             exit(1);
          }
          while(1) {  // loop principal accept()
             sin_size = sizeof(struct sockaddr_in);
                if ((nova_fd = accept(sockfd, (struct sockaddr *)&outro_end, &sin_size)) == -1)
{
                    perror("accept");
                    continue;
                }
                printf("servidor: recebeu conexão de %s\n",
          inet_ntoa(outro_end.sin_addr));
          if (!fork()) { // este é o processo-filho
                    close(sockfd); // o filho não precisa de escutar conexões
                if (send(nova_fd, "Olá , Beleza!\n", 14, 0) == -1)
                       perror("send");
                   close(nova_fd);
                   exit(0);
                 }
           close(nova_fd);  // o processo principal não precisa disso
```

```
        }
      return 0;
   }
```

Você deve ter notado que o código inteiro está na função main(), sinta-se a vontade caso queira compartimentar em funções menores.Mais um detalhe. sigaction() deve ser novidade pra você. Essa função cuida para que os processos originados quando fork() termina , sejam tratados apropriadamente(ou seja ao invés de inundar o sistema com processos inúteis, sigaction() assegura que eles sejam "exterminados").

A seguir apresentarei um cliente que pode ser usado para receber os dados do servidor acima.

```
    /*
   ** cliente.c -- um "Stream Socket" cliente
   */
   // porta em que o cliente vai se conectar
   //  número máximo de bytes que você pode receber de uma vez
   int main(int argc, char *argv[])
   {
      int sockfd, numbytes;
      char buf[100 ];
      struct hostent *he;
      struct sockaddr_in outro_end; // informação do endereço externo
      if (argc != 2) {
         fprintf(stderr,"uso: cliente nomedohost\n");
         exit(1);
      }
      if ((he=gethostbyname(argv[1])) == NULL) {  // pega informação do host
         perror("gethostbyname");
         exit(1);
      }
      if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
         perror("socket");
         exit(1);
      }
      outro_end.sin_family = AF_INET;    // host byte order
      outro_end.sin_port = htons(3490 );  // converte em short,
network byte order
      outro_end.sin_addr = *((struct in_addr *)he-&gt;h_addr);
      bzero(&(outro_end.sin_zero), 8);   // zera o resto da estrutura
      if (connect(sockfd, (struct sockaddr *)&outro_end, sizeof(struct sockaddr)) == -1)
{
         perror("connect");
         exit(1);
      }
      if ((numbytes=recv(sockfd, buf, 100 -1, 0)) == -1)
```

```
{
        perror("recv");
        exit(1);
    }
    buf[numbytes] = '\0';
    printf("Recebido: %s",buf);
    close(sockfd);
    return 0;
}
```

Note que se você não rodar o servidor antes, connect() retorna "Connection refused!", ou conexão recusada.

Sockets de Datagrama (Datagram Sockets)

Bem não tenho muito o que falar aqui, então apenas irei apresentar dois códigos:dg_cliente.c e dg_servidor. dg_servidor espera por conexões em alguma máquina na porta 4950. dg_cliente envia um pacote para esta porta, na máquina especificada, contendo a entrada que o usuário digitou na linha de comando.

```
/*dg_servidor
**
*/
// porta em que o cliente se conecta
int main(void)
{
    int sockfd;
    struct sockaddr_in meu_end;    // minhas informações de endereço
    struct sockaddr_in outro_end; // informações do cliente
    int addr_len, numbytes;
    char buf[100 ];
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
//aqui está o detalhe
        perror("socket");
        exit(1);
    }
    meu_end.sin_family = AF_INET;
    meu_end.sin_port = htons(4950 );
    meu_end.sin_addr.s_addr = INADDR_ANY;
    bzero(&(meu_end.sin_zero), 8);
    if (bind(sockfd, (struct sockaddr *)&meu_end, sizeof(struct sockaddr)) == -1)
{
        perror("bind");
        exit(1);
    }
    addr_len = sizeof(struct sockaddr);
```

```
        if ((numbytes=recvfrom(sockfd,buf, 100 -1, 0,
                        (struct sockaddr *)&outro_end, &addr_len)) == -1)
{
            perror("recvfrom");
            exit(1);
        }
        printf("recebeu pacote de %s\n",inet_ntoa(outro_end.sin_addr));
        printf("tamanho do pacote: %d bytes \n",numbytes);
        buf[numbytes] = '\0';
        printf("pacote contém: \"%s\"\n",buf);
        close(sockfd);
        return 0;
    }
// porta que será usada pelo cliente
int main(int argc, char *argv[])
{
    int sockfd;
    struct sockaddr_in outro_end; // informções de endereço
    struct hostent *he;
    int numbytes;
    if (argc != 3) {
        fprintf(stderr,"uso: cliente hostname mensagem\n");
       exit(1);
    }
    if ((he=gethostbyname(argv[1])) == NULL) { // pega informação do host("gethostbyname");
       exit(1);
    }
    if ((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
 {
        perror("socket");
       exit(1);
    }
    outro_end.sin_family = AF_INET;
    outro_end.sin_port = htons(4950 );
    outro_end.sin_addr = *((struct in_addr *)he-&gt;h_addr);
    bzero(&(outro_end.sin_zero), 8);
    if ((numbytes=sendto(sockfd, argv[2], strlen(argv[2]), 0,
(struct sockaddr *)&outro_end, sizeof(struct sockaddr))) == -1)
{
    perror("recvfrom"); exit(1);
}
printf("enviou %d bytes para %s\n", numbytes,
inet_ntoa(outro_end.sin_addr));
```

```
close(sockfd);
return 0;
}
```

Tutorial de sockets - Parte VI

Por: Frederico Perim

Técnicas Avançadas

Não são realmente avançadas , mas não são tão básicas como os tópicos que já abordamos. Na verdade, se você chegou até aqui considere-se com bons conhecimentos básicos de programação de redes Unix.

Blocking

Você provavelmente notou quando executou o programa ____ , acima, que ele espera até que um pocote chegue. O que aconteceu é que ele chamou a função recvfrom(), e não havia dados, então recvfrom() está em estado "sleep", esperando pacotes.

Várias funções utilizam esse procedimento, tais como: accept() e todas as variantes de recv(). A razão pela qual realizam isso é porque a elas é permitido tal comportamento. Quando você cria um socket com a função socket(), o kernel vai ajustá-la para blocking. Se você não quiser que um socket realize isso, então você deve chamar a função fcntl():

```
sockfd = socket (AF_INET, SOCK_STREAM, 0);
fcntl (sockfd, F_SETFL, O_NONBLOCK);
.
.
.
```

Ao setar um socket para o estado non-blocking , você pode efetivamente "consultar" o socket para obter informações. Se você tentar ler um socket em estado non-blocking e não houver dados, ele retorna -1 .
Genericamente falando, no entanto, este tipo de "consulta" é uma má idéia. Se você manter seu programa ocupado esperando por dados em um socket, você vai consumir sua CPU ao ponto de tornar as coisas complicadas. Uma solução mais elegante para checar se exitem dados esperando para serem lidos vem na próxima seção sobre select().

select()

Esta função é um tanto estranha, mas muito útil. Analise a seguinte situação: você é um servidor e deseja escutar por conexões e ao mesmo tempo continuar recebendo dados das que já existem.
Sem problema, apenas uma chamada accept() e algumas recv(). Opa , não tão rápido!. E se a chamada accept() estiver em blocking? Como você vai receber dados ao mesmo tempo? "Use sockets non-blocking!" De forma alguma! Você não quer comprometer sua CPU. E então?
select() lhe dar o poder de monitorar várias conexões ao mesmo tempo. Ela lhe dirá quais estão prontas para receber, quais estão prontas para enviar, e quais leventaram exeções, se você realmente deseja saber.
Sem mais demora, aqui está a sinopse:

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

A função monitora conjuntos de descritores de arquivos, em particular readfds, writefds e exceptfds. Se você deseja saber se pode ler da entrada padrão e algum socket, sockfd, acrescente o descritor de arquivo 0 e sockfd ao conjunto readfds. Ao parâmetro numfds deve ser atribuído os maiores valores do descritor de arquivo mais um. Nesse exemplo, ele deve ser sockfd+1, já que seguramente é maior que a entrada padrão (0).
Quando select() retorna, readfs será modificado para refletir qual descritor você selecionou que está pronto para receber. Você pode testá-los com o macro FD_ISSET(), abaixo.
Antes de continuar, irei abordar a forma de manipular estes conjuntos. Cada conjunto é do tipo fd_set. Os seguintes macros operam nesse tipo:

- FD_ZERO (fd_set *set) -- limpa o conjunto
- FD_SET (int fd, fd_set *set) -- adiciona fd ao conjunto
- FD_CLR (int fd, fd_set *set) -- remove fd do conjunto

- FD_ISSET (int fd, fd_set *set) -- verifica se fd está no conjunto

Finalmente, o que é a estranha struct timeval? Bem, algumas vezes você não quer esperar para sempre alguém enviar dados. Talvez a cada 96 segundos você deseja mostrar "Ainda esperando..." no terminal embora nada tenha acontecido. Esta estrutura lhe permite especificar um período de tempo. Se esse período se excedeu e select() ainda não encontrou nenhum descritor de arquivo, select() retornará para que você continue processando.

struct timeval tem os seguintes campos:

```
struct timeval {
int tv_sec;  // segundos
int tv_usec;  //microsegundos
```

Apenas atribua a tv_sec o número de segundos para esperar, e a tv_usec o número de microsegundos para esperar. Sim, isto é microsegundos, não milisegundos. Existem 1000 microsegundos em um milisegundo, e 1000 milisengundos em um segundo. Assim, exitem 1000000 de microsegundos em um segundo. Porque "usec"? O u deve parecer com a letra grega μ que nós usamos para "micro". Ainda, quando a função retorna, timeout deve ser atualizado para mostrar o tempo restante. Isto depende da sua plataforma Unix.
Uau! Nós temos um timer em microsegundos! Bem, não conte com isso. O padrão timeslice do Unix é por volta de 100 milisegundos, por isso você deve esperar esse tempo independentemente do valor que você atribuir a struct timeval.
Outras coisas de interesse. Se você setar os campos em sua struct timeval para 0, select() irá encerrar imediatamente, consultando todos os descritores em seu conjunto. Se você atribuir NULL a timeout, então select() nunca irá encerrar e vai esperar até que o primeiro descritor esteja pronto. Finalmente, se você não se importar em esperar por algum conjunto, pode atribuir NULL na chamada select().
O código a seguir espera 2.5 segundos para alguma coisa aparecer na entrada padrão:

```
// descritor de arquivo para entrada
padrão
int main(void)
{
   struct timeval tv;
   fd_set readfds;
   tv.tv_sec = 2;
   tv.tv_usec = 500000;
   FD_ZERO(&readfds);
   FD_SET (0 , &readfds);
   //não  se importa com writefds e exceptfds:
   select (0 +1, &readfds, NULL, NULL, &tv);
   if (FD_ISSET(0 , &readfds))
      printf ("uma tecla foi pressionada!\n");
   else
      printf("Tempo esgotado.\n");
   return 0;
}
```

Alguns de vocês devem estar pensando que esta é uma ótima forma de esperar por dados em um socket UDP (SOCK_DGRAM) - e você está certo, pode ser. Alguns versões Unix podem usar select() dessa forma, outras não. Você deve consultar suas páginas man quando tentar fazer isso.
Algumas versões Unix atualizam o tempo em sua struct timeval para refletir a quantidade de tempo que ainda falta antes de se esgotar. Mas outras não. Não confie nisso se quiser que seu programa seja portável. (Use a função gettimeofday() se você precisar saber o tempo decorrido. É um saco, eu sei, mas tem que ser desse jeito.
Mais uma nota a respeito de select() : se você tiver um socket escutando (listen()), você pode checar se há uma nova conexão colocando o descritor socket desejado em um conjunto readfds.

E esta é, meus caros amigos, uma rápida análise da poderosa função select().
Mas, devido a pedidos, aqui esta um exemplo mais aprofundado. Infelizmente, a diferença entre o exemplo mais simples acima e este, é bastante significante. Mas dê uma olhada , e leia a descrição que a vem a seguir.

Este programa funciona como um servidor de chat multi-usuário simples. Rode-o em uma janela, e então conecte-se através do telnet ("telnet nomedohost 9034") de várias janelas diferentes. Quando você digita uma mensagem em uma janela, ele deverá aparecer em todas as outras.

```
// porta em que estaremos escutando
int main (void)
{
   fd_set mestre;  // descritor de arquivo mestre
   fd_set read_fds;    //  descritor de arquivo temporário para select()
   struct sockaddr_in  meu_end;   //endereço do servidor
   struct sockaddr_in  end_remoto;  // endereço do cliente
   int fdmax;  // número máximo do descritor de arquivo
   int listener;   // descritor de socket ouvindo
   char buf[256];  // buffer com os dados do cliente
   int nbytes;
   int yes=1;  // necessário para função
   setsockopt() SO_REUSEADDR, abaixo
   int addrlen;
   FD_ZERO(&mestre);   //limpa os conjuntos mestre e temporário
   FD_ZERO(&read_fds);
   //  pega o descritor de escuta
   if ((listener = socket (AF_INET, SOCK_STREAM, 0)) == -1)
{
      perror ("socket");
      exit (1);
   }
   // evita a mensagem de erro "endereço já em uso"
   if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof (int)) == -1)
{
      perror ("setsockopt");
      exit (1);
   }
   // conecta
   meu_end.sin_family = AF_INET;
   meu_end.sin_addr.s_addr = INADDR_ANY;
   meu_end.sin_port = htons (9034 );
   memset (&(meu_end.sin_zero), '\0', 8);
   if (bind (listener,  (struct sockaddr *)&meu_end, sizeof(meu_end)) == -1) {
      perror("bind");
      exit(1);
   }
   //escuta
```

```
    if (listen(listener, 10)  == -1) {
       perror("listen");
       exit(1);
    }
    //adiciona o socket em escura ao conjunto master
    FD_SET(listener, &master);         //  rastreia   o maior descritor de arquivo
    fd_max = listener; // até agora, este é o maior
    // loop principal
    fot (;;) {
       read_fds = master; // copia
       if (select (fdmax+1,  &read_fds, NULL, NULL, NULL) == -1)
{
          perror ("select");
          exit(1);
       }
       // percorre as conexões existentes em busca de dados
       fot (i = 0; i &lt;= fdmax; i++) {
                if  ( FD_ISSET ( i,  &read_fds))  { nós encotramos uma!!!
                If ( i == listener) {
                //trata novas conexões
                addrlen = sizeof (end_remoto);
                if (( newfd = accept(listener, &end_remoto,  &addrlen)) == -1)
{
                perror ("accept");
                } else {
                 FD_SET ( newfd , &mestre); //adiciona ao mestre
                   If ( newfd &gt; fdmax) {    // rastreia o maior
                   fdmax = newfd;
                   }
                   printf ("servidor : nova conexão de %s no  socket %d\n",
                      inet_ntoa (end_remoto.sin_addr), newfd);
                }
             } else {
                // cuida dos dados do cliente
                if  (( nbytes = recv (i, buf,  sizeof (buf), 0))  &lt;= 0)
{
                   // recebeu erro ou a conexão foi fechada pelo cliente
                   if (nbytes == 0) { // conexão encerrada
                      printf ("servidor: socket %d desligado\n",  i);
                   } else {
                      perror ("recv");
                   }
                   close (i); // tchau!!
```

```
                    FD_CLR( i , &master); // remove do conjunto mestre
                } else {
                    // temos alguns dados do cliente
                    for (j = 0; j &lt;= fdmax; j++)  { //envia a todos os clientes
                        if (FD_ISSET (j, &master)) { // exceto a nós e ao socket em escuta
                            if (j != listener && j != 1) {
                                if (send ( j, buf, nbytes, 0) == -1) {
                                perror ("send");
                                }
                            }
                        }
                    }
                }
            } // Isso é muito feio!!
        }
    }
    return 0;
}
```

Note que eu tenho dois conjuntos de descritores de arquivo: mestre e read_fds. O primeiro, mestre, tem todos os descritores de socket que estão conectados, assim como o descritor socket que está escutando novas conexões.
A razão pela qual eu tenho um conjunto mestre é que select() na verdade muda o conjunto que você passa a ele para refletir quais sockets estão prontos para ler (receber dados) . Já que eu tenho que verificar as conexões de uma chamada select() para outra, eu tenho que armazená-los em algum lugar seguro. No último minuto , eu copio o mestre em read_fds , e então chamo select().
Mas isso não significa que toda vez que tenho uma nova conexão, não tenho que acrescentá-la ao conjunto mestre? Sim! E toda vez que uma conexão é encerrada, eu tenho que retirá-la do conjunto mestre? Sim, tem.
Note que eu checo para ver quando o socket que esta escutando (listener) está pronto para receber dados. Caso afirmativo, significa que tenho uma nova conexão pendente , então eu aceito (accept()) e adiciono ao conjunto mestre. Da mesma forma , quando uma conexão do cliente está pronta para receber, e recv() retorna 0, eu sei que o cliente encerrou a conexão, então tenho que removê-lo do conjunto mestre.
Se , no entanto, a chamada recv() do cliente retornar um valor diferente de zero, eu sei que algum dado foi recebido. Então eu percorro toda lista mestre e envio o dado para o resto dos clientes.
É isso aí caros amigos, aqui termina nossa análise da função select().

Tutorial de sockets - Parte VII

Por: Frederico Perim

Lidando com send() parciais

Lembra da seção sobre send() quando eu disse que este pode não enviar todos os bytes que você pediu? Ou seja, você quer enviar 512 bytes, mas retorna 412. O que aconteceu com os outros 100 bytes?

Bem, eles ainda estão no buffer esperando serem enviados. Devido a circunstâncias além do seu controle, o kernel decidiu não enviar os dados de uma vez, e agora meu amigo, cabe a você enviá-los.

Você poderia escrever uma função com esta para fazer isso:

```
int enviatudo (int s , char *buff, int *len)
{
    int total = 0 ; // quantos bytes enviamos?
    int bytes_resta = *len; // quantos ainda restam para enviar
```

```
    int n;
    while ( total < *len) {
        n = send (s, buf+total, bytes_resta, 0);
        if ( n == -1  ) { break; }
        total += n;
        bytes_resta -= n;
    }
    *len = total; // retorna o número que realmente foi enviado
    return n==-1?-1: 0; // retorna -1 em caso de falha, 0 caso sucesso
}
```

Nesse exemplo, s é o socket para onde você quer enviar os dados, buf é o buffer contendo os dados, e len é um ponteiro para um int contendo o número de bytes no buffer.
A função retorna -1 caso ocorra erro. Além disso, o número de bytes realmente enviados é retornado em len. Isso será o mesmo número de bytes que você pediu para enviar, contanto que não ocorra erro. enviatudo() vai fazer o possível para enviar os dados , mas caso ocorra erro, ela retorna para você imediatamente.
Para completar , aqui está um exemplo de chamada para esta função:

```
char  buf[10] =  "Hrerer!";
int len;
len = strlen (buf);
if (enviatudo ( s, buf, &len) == -1) {
    perror ("enviatudo");
    printf ("Nós somente enviamos %d bytes porque houve erro!!\n", len);
}
```

O que acontece no lado do receptor quando parte do pacote chega? Se os pacotes são de tamanhos variáveis, como o receptor sabe quando um pacote termina e outro começa? Sim , cenários reais são chatos assim. Você provavelmente vai ter que empacotar (lembra-se da seção sobre encapsulamento no começo do documento?) Continue lendo para mais detalhes.

Filho do Empacotamento de Dados

O que realmente significa empacotar dados? No mais simples caso, significa que você anexa um cabeçalho com informações de identificação do pacote ou o tamanho do pacote, ou ambos.

Como deve ser a aparência? Bem, é apenas algum dado binário que representa qualquer coisa que você julgue necessário para completar o projeto.

Nossa... isso é vago.

Tudo bem. Por exemplo, digamos que você tem programa de chat multi-usuário que utiliza SOCK_STREAMS. Quando um usuário digitar alguma coisa, dois pedaços de informações precisam ser transmitidos ao servidor. O que foi dito e quem disse.

Até agora tudo brm? Qual o problema?

O problema é que as mensagens podem ser de tamanhos variados. Uma pessoa chamada "José" pode dizer, "Oi, e outra pessoa chamada "Carlos" pode dizer, "Como vai, tudo bem?"

Então você envia (send()) tudo isso para os clientes. Seu fluxo de saída se parece com isso:

```
J o ã o O i C a r l o s C o m o v a i ,  t u d o b e m ?
```

E assim por diante. Quando o cliente sabe quando uma mensagem começa e a outra termina? Você poderia , se quisesse, fazer todas as mensagens do mesmo tamanho e apenas chamar enviatudo() que nós implementamos acima. Mas isso desperdiça largura de banda! Nós não queremos enviar (send()) 1024 bytes apenas para que João diga "Oi".
Então nós "empacotamos" os dados em um pequeno cabeçalho e estrutura de pacote. Tanto o cliente quanto o servidor sabem como empacotar e desempacotar estes dados. Não olhe agora , mas estamos começando a definir um protocolo que descreve como um cliente e um servidor devem se comunicar!!
Neste caso, assuma que o nome do usuário tem um tamanho fixo de 8 caracteres, completados com "\0". E então vamos assumir que esse dado é de tamanho variável, com no máximo 128 caracteres. Vamos dar uma olhada em uma estrutura de pacote que podemos usar nessa situação:

1. tam (1 byte, unsigned) - O tamanho do pacote, contando os 8 bytes do nome do usuário e dados do char.
2. nome (8 bytes) - O nome do usuário , completados com NULL se necessário
3. chatdara (n - bytes) - Os dados em si, não mais que 128 bytes. O tamanho do pacote deve ser calculado como o tamanho desses dados mais 8 (o tamanho do campo nome acima).

Porque escolhi os limites 8 e 128 bytes para os campos? Eu chutei, assumindo que seriam grandes o suficiente. Talvez, no entanto, 8 bytes seja muito restritivo para suas necessidades, a escolha depende de você.
Usando a definição de pacote acima , o primeiro pacote consistiria das seguintes informações (em hex e ASCII).

OA 74 6F 6D 6E OO OO OO OO OO OO 48 69 (tamanho) J o ã o (complemento) O i

(O tamanho é armazenado em Ordem de Byte de Rede (Network Byte Order). Nesse caso , é somente um byte então não importa, mas geralmente você vai querer todos seus inteiros binários em NBO em seus pacotes)

Quando você estiver enviando estes dados, você pode estar seguro em usar uma função similar a enviatudo(), acima, assim você sabe que todos os dados são enviados, mesmo que leve múltiplas chamadas send() para completar o processo.

Da mesma forma, quando estiver recebendo dados , você precisa fazer um pouco de trabalho extra. Para ter certeza, você deve assumir que recebeu parte dos dados. Assim precisamos chamar recv() várias vezes até que o pacote completo tenha chegado.

Mas como? Bem , nós sabemos o número de bytes que precisamos receber ao todo para que o pacote esteja completo, já que este número é anexado na frente do pacote. Nós também sabemos que o tamanho máximo do pacote é: 1+8+128, ou 137 bytes (porque foi assim que definimos o pacote.)

O que você pode fazer é declarar um array grande o suficiente para dois pacotes. Este é a sua array de trabalho onde você irá reconstruir pacotes assim que chegarem.

Toda vez que receber (recv()) dados, você os passa para a array e checa se o pacote está completo. Ou seja, o número de bytes no buffer (array) é maior ou igual em relação ao número especificado no cabeçalho (+1, porque o tamanho no cabeçalho não o incluí o byte para o tamanho em si) Se o número de bytes no buffer é menor que 1, o pacote não está completo, obviamente. Você vai ter que fazer algo especial em relação a isso, já que o primeiro byte é lixo e não pode ser confiável para se averiguar o tamanho do pacote.

Uma vez que pacote esteja completo, você pode fazer o que quiser. Use e remova de seu buffer.

Nossa você já absorveu isso na sua mente? Bem , então ai vai a segunda parte: você pode ter lido todo o primeiro pacote e parte do segundo com uma única chamada recv(). Ou seja, você tem um buffer com um pacote completo , e parte do próximo pacote! Que saco (Mas isso é porque você fez um buffer grande o suficiente para suportar dois pacotes!)

Já que você sabe o tamanho do primeiro pacote através de seu cabeçalho, e andou checando o número de bytes no buffer , você pode subtrair e calcular quantos bytes pertencem ao segundo (incompleto) pacote. Quando você já cuidou do primeiro , pode limpá-lo do buffer e mover parte do segundo para frente do buffer para que esteja pronto para próxima chamada recv().

Bem, realmente não é algo trivial . Agora você precisa praticar e logo tudo isso fará sentido naturalmente. Eu juro!!!!

EXEMPLO DE APLICAÇÃO SOCKET – SIMONE MARKENSON

```
/*************************************************************************
 * Arquivo uici.h
 * ********************************************************************** */


#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <errno.h>
#include <utmp.h>


#define BLKSIZE 1024
#define MAXBACKLOG 5


typedef unsigned short u_port_t;
int u_open (u_port_t porta);
int u_listen (int fd, char *hostn);
int u_connect(u_port_t porta,char *inetp);
int u_close( int fd);
ssize_t u_read(int fd,void *buf,size_t nbyte);
ssize_t u_write(int fd, void *buf, size_t nbyte);
void u_error( char *s);
int u_sync (int fd);

/*************************************************************************
 * Funcoes utilizadas para comunicacao via socket
 * ********************************************************************** */

#include "uici.h"

int u_open(u_port_t porta)
{
        int sock;
        struct sockaddr_in server;

        if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
                return -1;

        server.sin_family = AF_INET;
        server.sin_addr.s_addr = INADDR_ANY;
        server.sin_port = htons((short)porta);

        if ((bind(sock, (struct sockaddr *)&server, sizeof(server)) < 0) ||
                (listen(sock, MAXBACKLOG) < 0))
                return -1;
```

```
        return sock;

}

int u_close(int fd)
{
        return close(fd);
}

ssize_t u_read(int fd, void *buf, size_t size)
{
        ssize_t retval;
        while (retval = read(fd, buf, size), retval == -1 && errno == EINTR);
        return retval;
}

ssize_t u_write(int fd, void *buf, size_t size)
{
        ssize_t retval;
        while (retval = write(fd, buf, size), retval == -1 && errno == EINTR);
        return retval;
}

int u_listen(int fd, char *hostn)
{
        struct sockaddr_in net_client;
        int len = sizeof(struct sockaddr);
        int retval;
        struct hostent *hostptr;

        while ( ((retval =
                accept(fd, (struct sockaddr *) (&net_client), &len)) == -1) &&
                (errno == EINTR) )
                ;
        if (retval == -1)
                return retval;
        hostptr = gethostbyaddr((char *)&(net_client.sin_addr.s_addr), 4, AF_INET);

        if (hostptr == NULL)
                strcpy(hostn, "unknown");
        else
                strcpy(hostn, (*hostptr).h_name);
        return retval;
}
int u_connect(u_port_t porta, char *hostn)
{

        struct sockaddr_in server;
        struct hostent *hp;
        int sock;
        int retval;

        hp = gethostbyname(hostn);
        if ( !(hp = gethostbyname(hostn)) ||
                ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) )
```

```
                return -1;

        memcpy((char *)&server.sin_addr, hp->h_addr_list[0], hp->h_length);

        server.sin_port = htons((short)porta);
        server.sin_family = AF_INET;

        while ( ((retval =
                connect(sock, (struct sockaddr *)&server, sizeof(server))) == -1)
                && (errno == EINTR) ) ;
        if (retval == -1) {
                close(sock);
                return -1;
        }
        return sock;
}
```

Exemplo de Cliente:

```
#include "uici.h"

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

{
        u_port_t port;
        int fd,i,bw,br;
        char buf[BLKSIZE]="request";
        char bufin[BLKSIZE];

        if (argc!=3) {
                fprintf(stderr,"Utilizacao: %s host port\n\r",argv[0]);
                exit(1);
        }
        port = (u_port_t) atoi(argv[2]);
        fd = u_connect(port,argv[1]);
        if (fd<0) {
                perror("Conexao");
                exit(1);
        }
        fprintf(stderr," Conexao estabelecida com %s\n\r",argv[1]);
        /* envia mensagem para o servidor */
        bw = u_write(fd,buf,sizeof(buf));
        br = u_read(fd,bufin,sizeof(bufin));
        printf("Client: Mensagem recebida: %s\n\r",bufin);
        u_close(fd);

}
```

Exemplo de Servidor:

```
#include "uici.h"

#define PORTA1 8000
```

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

{

 u_port_t porta;
 int fd, listenfd, sock, terminal;
 char maquina[MAX_CANON];
 ssize_t br,bw;
 char mensagem[80],resposta[80]="reply";

 if (argc != 1)
 {
  fprintf(stderr, "Numero invalido de parametros\n");
  exit(1);
 }

 porta = PORTA1;

 if ((fd = u_open(porta)) == -1)
 {
        fprintf(stderr, "Nao foi possivel estabelecer conexao\n");
        close(fd);
        exit(1);
 }

 else
 {
        fprintf(stderr, "Escutando porta \n");

 }

 for (; ;){
        if ((listenfd = u_listen(fd, maquina)) >= 0){

           printf("Conectado... Aguardando mensagem\r\n");
           printf("Server: maquina %s\n\r",maquina);

           br= u_read(listenfd,mensagem,sizeof(mensagem));
           printf("Server:Mensagem recebida: %s\r\n",mensagem);
           bw = u_write(listenfd,resposta,sizeof(resposta));
           printf("Server:Mensagem enviada: %s\r\n",resposta);
        }
        else {
                printf("Server: maquina %s\n\r",maquina);
                fprintf(stderr,"Nao foi possivel conectar");
           exit(1);
        }
 }
}
```