Terminologia

Host computador onde executam aplicações do usuário ou do sistema

Gateway computador que conecta hosts de uma rede local a uma ou mais redes externas

Roteador computador intermediário que roteia pacotes na Internet

Cliente *processo* executando num *host cliente* (ou host local) que solicita *serviços* a um *Servidor* em benefício de um *usuário*.

Servidor *processo* executando num host servidor (ou host remoto) que provê *serviços* a um ou mais *clientes* **.**

Obs: o termo *cliente* ás vezes é usado para denominar o *host cliente* ou o *usuário*;

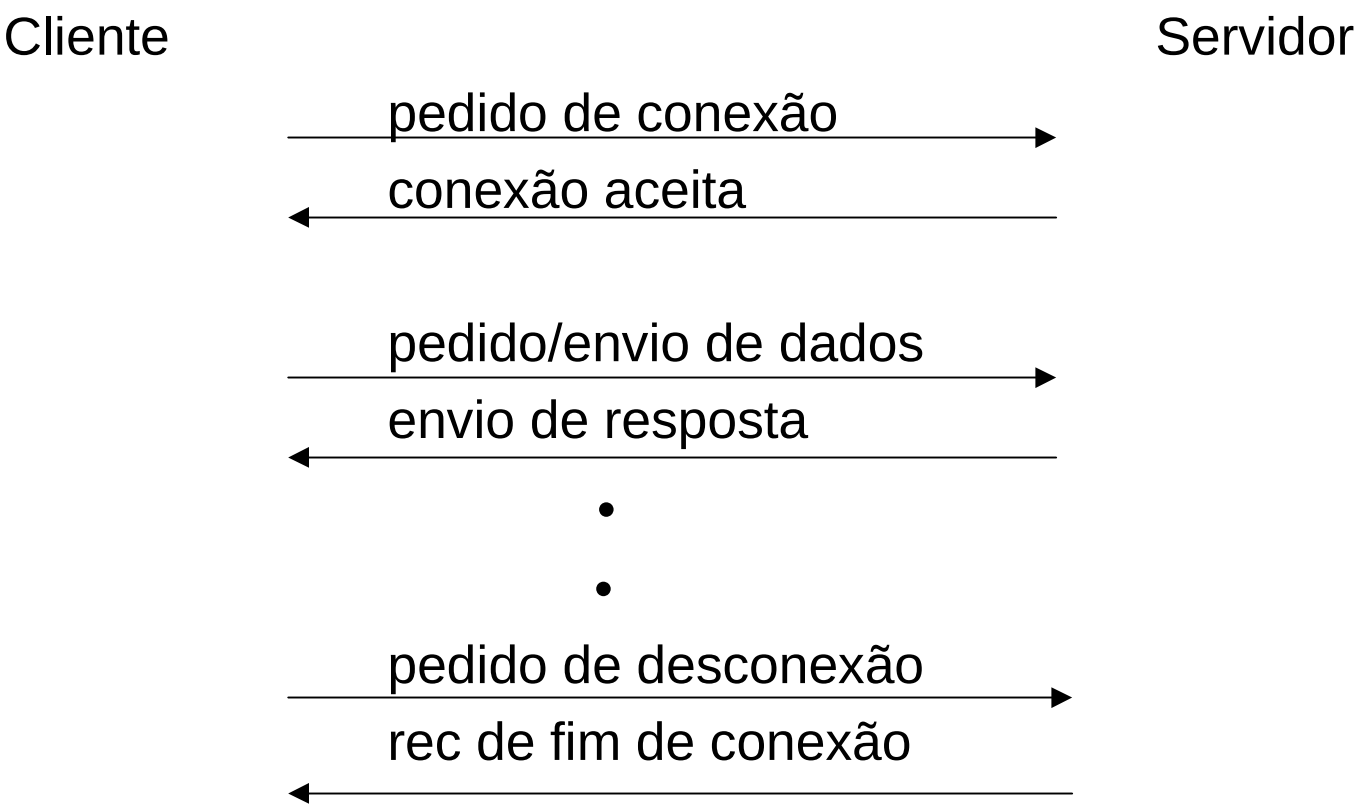
servidor ás vezes se refere ao *host servidor*.

O Modelo Cliente - Servidor

Modelo simples de comunicação entre um (processo) *cliente* e um (processo) *servidor*

também chamada de “comunicação pedido-resposta”

Exemplo:



Protocolos de Transporte Internet

TCP - Transmission Control Protocol

UDP - User Data Protocol

Atuam sobre o protocolo de rede IP - Internetworking Protocol

TCP:

- “orientado a conexão” entre cliente e servidor
- comunicação “fim a fim”: entre dois processos um em cada ponta da conexão
- orientado a fluxo contínuo de bytes (~ “pipe remoto”), mas com
- comunicação bidirecional (“full duplex”)
- dados enviados em unidades de tamanho variável: “segmentos”, de forma transparente à aplicação
- confiável: todo segmento enviado requer um ACK (reconhecimento):se ACK não chega segmento é retransmitido

- sequenciamento: bytes de segmentos são numerados, de forma a garantir:
  - entrega em ordem,
  - deteção e eliminação de duplicatas
- provê controle de fluxo através de “janelas dinâmicas” e mecanismos elaborados de deteção/prevenção de congestão
- estimativa dinâmica do Round Trip Time (RTT) provê mecanismos eficientes de retransmissão (timeouts dinâmicos)
- voltado para atuar sobre redes heterogêneas com:
  - tamanhos máximos de pacotes variáveis
  - faixas de passagem variáveis
  - topologias distintas
- ponto fraco atual: adaptação a taxas de erros grandes, comum em comunicação sem fio (wireless)

UDP:

- não orientado a conexão (“connectionless”)
- orientado a mensagens (datagramas)
- fim a fim
- preserva fronteiras de mensagens
- não confiável (“best effort”)
- não garante ordem de entrega de mensagens
- não tem controle de fluxo ou de congestão
- voltado para aplicações tipo “pedido - resposta”
- baixo overhead (não há estabelecimento ou encerramento de conexão)
- desvantagem: confiabilidade tem que ser colocada pela aplicação

OBS: TCP e UDP fazem parte do kernel nos sistemas Unix

Fases de uma comunicação TCP (entre cliente e servidor):

- Estabelecimento da conexão,
- Troca de dados (bidirecional) entre cliente e servidor,
- Encerramento da conexão.

(cada fase usa segmentos com cabeçalhos apropriados).

Estabelecimento de conexão (3-way handshake):

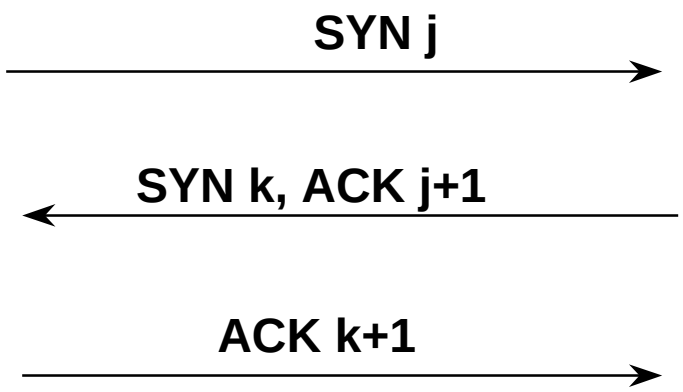
3 segmentos são trocados entre cliente e servidor:

- cliente envia segmento tipo SYN

  (pedido de conexão, com número inicial da *sequencia de numeração de bytes* no sentido cliente - servidor)
- servidor reconhece pedido de conexão enviando segmento tipo SYN com bit de reconhecimento (ACK) ligado e com *número inicial de sequencia de numeração* no sentido servidor - cliente.
- cliente envia segmento ACK reconhecendo SYN do servidor

Estabelecimento de conexão (3-way handshake)

Cliente **Servidor**



A partir desse ponto cliente e servidor passam a enviar segmentos de dados e ACKs (**Fase de troca de dados**).

Encerramento de conexão TCP:

Pode ser iniciada tanto pelo cliente como pelo servidor.

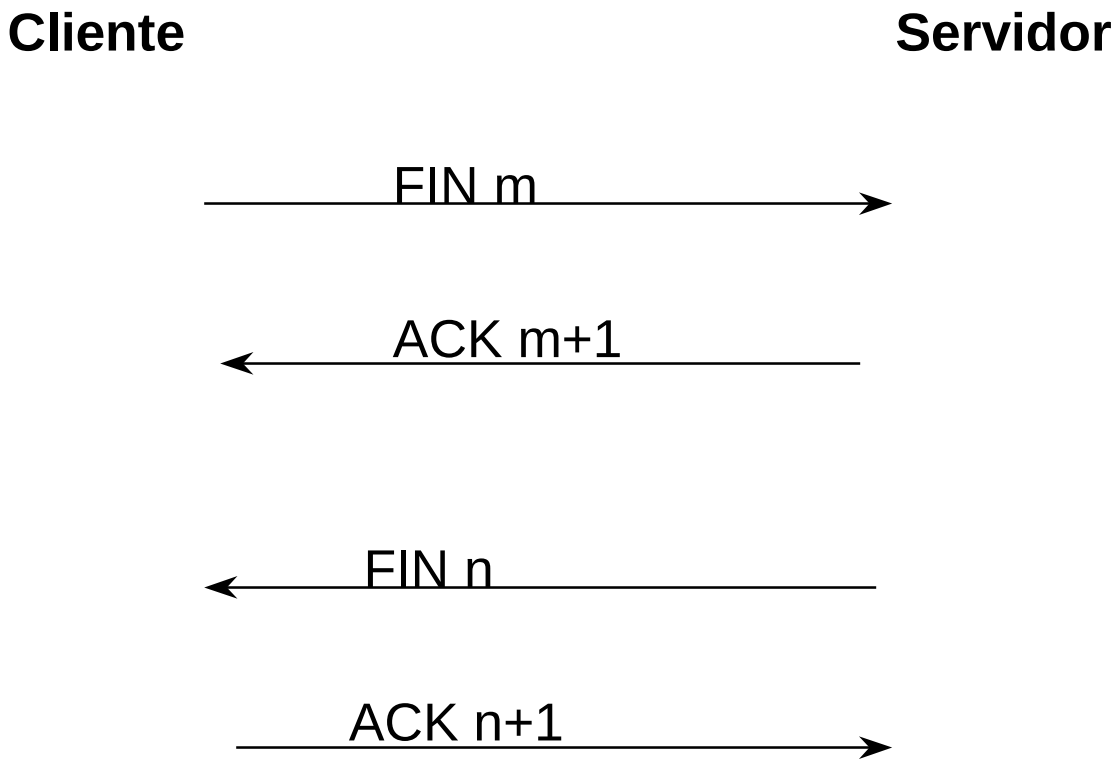
Como exemplo, suporemos iniciada pelo cliente:

- cliente envia segmento *FIN m* (sinalizando fim da conexão no sentido cliente - servidor: *"half-close*")
- servidor envia reconhecimento: *ACK m+1*
- algum tempo depois servidor envia *FIN n* (sinalizando fim da conexão no sentido servidor - cliente)
- cliente envia reconhecimento *ACK n+1*

Obs: (i) o servidor pode combinar os dois segmentos que envia num só: (*FIN n, ACK m+1*)

(ii) entre os passos 2 e 3 acima, o servidor pode enviar um ou mais segmentos de dados

Encerramento de conexão



O conceito de socket

Uma comunicação entre **dois processos** via TCP (ou UDP)
é identificada *unívocamente* por dois pares de valores,
que denotam os dois pontos da comunicação
(um em cada processo):

um “socket local” = (IP local, porta local), e

um ”socket remoto” = (IP remoto, porta remota)

(suponha, por conveniência, um processo numa máquina “local” e outro numa máquina “remota”),

onde,

IP local é o (um) número IP da máquina local,
IP remoto é o (um) número IP da máquina remota, e
porta local/remota é um número de 16 bits

Comunicação Cliente - Servidor:

é conveniente chamar a máquina Cliente de “local”,

e a máquina Servidora de “remota”

O contexto em geral define sem ambiguidade

o que é “local” e o que é “remoto”

Serviços do Unix usam “portas conhecidas”

“well known ports”:

são portas reservadas (p/ IANA) com números 0 a 1023

somente processos privilegiados podem usá-las

Num Servidor um *socket local* identifica um *serviço* oferecido,

podendo suportar múltiplas comunicações simultâneas.

Exemplos de “portas conhecidas” e seus serviços associados:

telnet 23, ftp 21, http 80, tftp 69

portas acima de 1024 podem ser usadas por aplicações comuns (embora de 1024 a 49151 possam ser “registradas”)

Sockets podem ser usados tanto para conexões TCP obtidas através do “3-way handshake” entre dois hosts, como para comunicação entre dois processos via UDP

A interface de aplicação (API) de sockets

São funções que provêm uma interface entre a aplicação do usuário e a camada TCP/UDP logo abaixo.

Função *socket*

É a 1ª função a ser chamada por uma aplicação que usa sockets.
Importante não confundi-la com o conceito de “socket local”

int socket (int family, int type, int protocol)
retorna > 0 se OK, -1 se erro

retorna um descritor de socket (um inteiro positivo pequeno), muito semelhante a um descritor de arquivo Unix (suportando, por exemplo, operações de read, write e close)

Descrição dos parâmetros:

family: uma dentre as constantes:

AF_INET	socket usa internet IPv4
AF_INET6	socket usa internet IPv6
AF_UNIX ou AF_LOCAL	socket domínio Unix
AF_ROUTE	socket para roteamento
AF_KEY	socket para uso com criptografia

type: um dentre as constantes:

SOCK_STREAM	socket será usado com TCP
SOCK_DGRAM	socket será usado com UDP
SOCK_RAW	socket usa pacotes ICMP, IGMP ou datagramas IPv4 ou IPv6

protocol:	0 para aplicações comuns

Estaremos interessados apenas nas combinações:

Família	*Tipo*	
AF_INET	SOCK_STREAM	SOCK_DGRAM
AF_UNIX	SOCK_STREAM	SOCK_DGRAM

Exemplos:

sockfd = socket(AF_INET, SOCK_STREAM, 0),

cria um socket Internet (IP4) para uso com TCP.

sockfd = socket(AF_INET, SOCK_DGRAM, 0)

cria um socket Internet (IP4) para uso com UDP.

sockfd = socket(AF_LOCAL, SOCK_STREAM, 0),

cria um socket domínio Unix para uso com TCP.

Função *connect*

*int connect (int sockfd, const struct sockaddr * servaddr, int addrlen)*

retorna 0 se OK, -1 se erro,

cliente usa para iniciar conexão com servidor remoto:

vai iniciar o ”3 way handshake" do protocolo TCP

sockfd: obtido na chamada anterior a *socket()*

servaddr: estrutura inicializada previamente com a identificação do socket remoto: (IP remoto, # porta remota)

Não é necesário que o cliente chame *bind* :

o socket local é escolhido pelo kernel e consiste do par:

(IP local, # porta transiente),

escolhida de forma a não conflitar com outras em uso

Erros possíveis (registrados na variável global *errno*):

Connection refused: o servidor não estava esperando uma conexão na porta especificada

Connection timed out: um servidor inexistente na mesma subrede foi endereçado

No route to host: foi especificado um endereço IP não alcançável pelos roteadores

A função *bind*

int bind(int sockfd, (struct sockaddr) myaddr, int socklen)*

retorna 0 se OK, -1 se erro

bind associa ao descritor *sockfd* um valor para o “socket local” passado na estrutura *myaddr*.

A aplicação tem a opção de deixar para o sistema determinar o # IP ou o # porta ou ambos;

Isto em geral é conveniente, pois:

• se o host tiver mais de um # IP (por exemplo, tem duas ou mais interfaces Ethernet) o IP “mais apropriado” para a comunicação é escolhido, i.é., aquele que será usado para envio de um pacote IP através do socket (consultando a tabela de roteamento do host), ou no caso de recepção, o IP da interface através da qual chegou o pacote IP para este host.

Se o sistema escolhe o # porta, este não conflitará com nenhuma outro # porta associado a algum outro socket.

Nesse caso se diz que o # porta é “efêmero”.

Para deixar o sistema escolher o # IP deve-se atribuir a constante INADDR_ANY ao campo para o # IP em *myaddr*

Para escolha de # porta efêmero pelo sistema deve-se atribuir a constante 0 ao campo para # porta em *myaddr*:

**myaddr.sin_port = 0;*

**myaddr.sin_addr.s_addr = INADDR_ANY*

Erros:

“Address already in use”, pode ocorrer se o socket já estiver sendo usado:wq

A função *listen*

Um socket é considerado *ativo* se a aplicação invoca *connect* com o socket, iniciando o “3-way handshake” com outro host;

se a aplicação invocar *listen*, então o socket passa a ser *passivo*, indicando que o kernel deve aceitar pedidos de conexão sobre o socket;

Clientes invocam *connect* enquanto servidores invocam *listen* seguido de *accept.*

*int listen (int sockfd, (struct sockaddr) * myaddr, int backlog)*
retorna 0 se OK, -1 se erro

O parâmetro *backlog* corresponde ao tamanho de uma fila no kernel para o número de conexões em andamento e completadas.

Em geral o sistema admite mais conexões que as especificadas em *backlog*

Conexões “em andamento”: “3 way handshake” iniciado porém ainda não concluído.

Conexões “completadas”: “3 way handshake” concluído.

A camada TCP guarda informações sobre os dois tipos de conexão em duas filas do kernel TCP.

Apenas conexões concluídas serão retiradas (FIFO) da fila através da função *accept* a ser vista

A função *accept*

Invocada por um servidor TCP para obter os dados e retirar da fila a 1ª conexão da fila de conexões concluídas.

Se a fila estiver vazia processo fica bloqueado até alguma conexão completar,

senão o kernel cria e retorna um novo descritor de socket para esta conexão, além de preencher o valor do “socket remoto” (do cliente) na estrutura *sockaddr* passada como parâmetro:

*int accept (int sockfd, (struct sockaddr) * cliaddr, int * socklen)*

retorna: valor descritor de socket (>0) se OK, -1 se erro

socklen é passado por valor-resultado, retornando o tamanho real da estrutura preenchida pelo kernel (igual ao passado, no caso de um socket internet)

A criação do novo socket permite ao servidor dedicar o seu uso para a nova conexão;

por isso êle é chamado de “socket conectado” (“connected socket”);

o socket original *sockfd* será usado pelo servidor para receber novas conexões e é chamado de “socket de escuta”

(“listening socket”);

a cada nova conexão o servidor cria um novo “socket conectado”.

Em geral o servidor cria um novo processo para tratar a nova conexão, passando para o mesmo o descritor do “socket conectado”.

Nesse caso o servidor pode tratar simultaneamente várias conexões (com durações variáveis ou imprevisíveis) e dizemos que êle é um “servidor concorrente”.

Se o servidor trata uma conexão de cada vez êle é dito “iterativo”.

Servidores iterativos são muito simples e podem ser convenientes caso o serviço provido seja de curta duração

(comparável ao tempo necesário para criar um processo para prover o serviço).

Exemplo: o serviço “daytime” (porta 13) devolve um string com o “dia e hora” correntes.

```
/**Exemplo:  server0.c -- um servidor TCP iterativo */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#define MYPORT 3490    /* the port users will be connecting to */
#define BACKLOG 10     /* how many pending connections queue will hold */

main()
{
    int sockfd, new_fd;  /* listen on sock_fd, new connection on new_fd */
    struct sockaddr_in my_addr;    /* my address information */
    struct sockaddr_in their_addr; /* connector's address information */
    int sin_size;
```

```
 if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
     perror("socket");
     exit(1);
  }
  my_addr.sin_family = AF_INET;         /* host byte order */
  my_addr.sin_port = htons(MYPORT);                /* short, network byte order */
  my_addr.sin_addr.s_addr = INADDR_ANY;    /* automatically fill with my IP */
  bzero(&(my_addr.sin_zero), 8);          /* zero the rest of the struct */
  if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr)) == -1) {
     perror("bind");
     exit(1);
  }
if (listen(sockfd, BACKLOG) == -1) {
     perror("listen");
     exit(1);
  }
```

```
sin_size = sizeof(struct sockaddr_in);
while(1) {  /* main accept() loop */
      if ((new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1) {
           perror("accept");
           continue;
       }
       printf("server: got connection from %s\n",inet_ntoa(their_addr.sin_addr));

       if (send(new_fd, "Hello, world!\n", 14, 0) == -1){
                perror("send");
                exit(1);
       }
       close(new_fd);
    }
}
```

Estruturas de Dados para Sockets

Apontador para uma “socket address structure” requerido pela maioria das funcões da API de sockets

Cada “familia de protocolos” requer uma estrutura diferente:

IPv4, IPv6, Unix, Datalink;

sockaddr-in é o nome da estrutura para IPv4

tamanho da estrutura varia com a família de protocolo

Por isso é usada uma “estrutura genérica” : *struct sockaddr,*

de 16 bytes, em todas as funções e para qq protocolo,

através de um cast da estrutura do protocolo para a genérica:

Ex:

struct sockaddr_in my_addr

*(struct sockaddr) * my_addr*

Estruturas “socket addresses” dos protocolos:

IPv4: (16 bytes)

AF_INET(2bytes)

porta (2bytes)

IP (4bytes)

Não usado

(8 bytes)

IPv6 (24 bytes)

AF_INET6(2bytes)

porta (2bytes)

flow label (4bytes)

IP (4bytes)

Unix (até 106 bytes)

AF_UNIX (2 bytes)

pathname

(variável até 104 bytes)

O apontador para uma *sockaddr* (passado da aplicação p/ o kernel) deve ser interpretado de duas formas, dependendo da função:

por **valor,** da aplicação para o kernel:

bind, connect, send, sendto

(passam o comprimento da estrutura por valor)

por **resultado,** do kernel para a aplicação:

accept, recv, recvfrom, getsockname, getpeername

(mas, passam o comprimento da estrutura por **valor-resultado**)

Exemplos:

*1) bind (sockfd, (struct sockaddr) * my_addr, sizeof(my_addr))*

2) int len = sizeof(their_addr)

*newsock= accept(sockfd, (struct sockaddr) * their_addr, len)*

Obs: o valor passado em *len* é usado pelo kernel na atualização de *their_addr,* para não sobre-escrever a área da aplicação, caso a estrutura retornada seja maior

Tipos de Servidores

Concorrentes	Iterativos
Orientado a conexão (TCP) ✔ Multiprocesso ✔ Uniprocesso	Orientado a conexão(TCP) ✔ Uniprocesso
Connectionless (UDP) Multiprocesso (Ex:TFTP)	Connectionless (UDP) ✔ Uniprocesso
Híbridos qto à conexão	✔ Híbridos qto à conexão

A função *shutdown*

- Permite fechar uma conexão TCP independentemente do contador de referências do descritor de socket (pouco útil)
- Permite fechar a conexão apenas num sentido:

  útil no exemplo de eco de arquivo com E/S multiplexada:

  o cliente não deve fechar a conexão (close) após enviar a última linha do arquivo pois linhas em transito podem não ter sido recebidas pelo cliente;

  é razoável, porém, fechar a conexão no sentido de enviar, i. é. para escrita (*half close*):

shutdown (sockfd, SHUT_WR)
retorna 0 se OK, -1 se erro

dados no buffer de tranmissão são enviados,

a conexão é fechada para escrita,

dados podem ser recebidos (lidos) mas não enviados (escritos)

Como funciona:

TCP envia segmento FIN após enviar o último segmento com dados. Servidor só envia FIN após dados terminarem (via close)

Outros modos de *shutdown:*

shutdown (sockfd, SHUT_RD)

conexão é fechada para recepção (leitura),

dados no buffer de recepção são descartados,

envio de dados permitido, nas não recepção (leitura)

shutdown(sockfd, SHUT_RDWR)

conexão é fechada nos dois sentidos,

mesmo que o contador de referências do socket seja > 0

Sockets UDP

- Úteis para aplicações do tipo “Pedido - Resposta”,

  com poucas trocas de mensagens entre Cliente e Servidor
- Usado pelos serviços DNS, NFS, SNMP e TFTP

  Servidor usa *bind,* cliente não precisa usar *connect* (embora possa)

  Bind especifica o socket local (IP, porta) para recepção de datagramas pelo servidor.

  após *bind*, servidor, fica bloqueado em *recvfrom,* à espera de um datagrama

  Cliente inicia interação enviando um pedido via *sendto*

Sendto

- especifica apenas o socket remoto,
- sistema escolhe o socket local (IP, porta aleat.)
- retorna o número de bytes enviados
  (na área de dados do datagrama UDP)
- é permitido enviar um datagrama com 0 bytes!

Recvfrom

- quando o datagrama chega o parametro com endereço de um *sockaddr_in* é preenchido com o valor do socket do remetente
- o valor do socket local foi fixado pelo *bind* anterior
- retorna o número de bytes recebidos
  (na área de dados do datagrama)

A recepção de um datagrama com 0 bytes
não significa fechamento da conexão:
não existe tal conceito com UDP!

Servidores UDP são, em geral, iterativos
pois há apenas uma porta para comunicação com clientes

Camada UDP tem um buffer onde datagramas são armazenados,
até serem retirados pela aplicação
a ordem de retirada é FIFO

O problema da confiabilidade:

Datagramas não têm garantia de entrega (“best effort”)

Se perdidos o cliente e/ou servidor podem ficar bloqueados

Temporizadores e retransmissão de datagramas podem ser
necessários para confiabilidade da aplicação

O problema dos “erros assíncronos”:

- Se cliente envia datagrama antes do servidor invocar *recvfrom:*

  datagrama é perdido!

  na verdade o protocolo IP no host destino retorna erro via mensagem ICMP com o código “port unreachable”

  esta mensagem não é tratada pela camada de socket

(porque não haveria como identificar o socket remoto)

esse problema foi chamado de “erro assíncrono”

Solução:

Cliente invoca “connect” para “ fixar” o socket remoto no Servidor para o qual quer enviar/receber datagramas

Connect não inicia o “3 way handshake” (como no caso de TCP),
apenas “registra” na camada UDP o socket remoto
para futuro uso via *send / recv*
(setpeername seria um nome mais apropriado)

Em que momentp é o “socket local” definido?
- na ausência de *bind* pelo cliente,
pelo sistema, quando o 1º datagrama é enviado,
permanece imutável para *sends* subsequentes!

Quando um socket UDP está “conectado” o cliente deve usar
apenas *send* e *recv* (ou *write e read*)

Efeitos causados pelo *"connect"*:

- *recv* só retorna datagramas provenientes do servidor cujo socket remoto foi especificado via connect pelo cliente

  os outros são descartados!

- Erros assíncronos são notificados quando o datagrama é enviado via *send* (e não quando *connect* é invocado)

  (*connect* não envolve nenhuma comunicação com o servidor)

  O erro notificado pelo sistema não é o "*port unreacheable*", do ICMP, pois UDP o converte para "*connection refused*"

- O servidor também pode chamar *connect* (TFTP faz isto!)

  a fim de permitir maior eficiência na troca de datagramas

- Para se “conectar” a um outro servidor basta o cliente chamar *connect* novamente
- Para se “desconectar” do servidor corrente o cliente pode chamar connect com o parametro

  sockaddr_in.sin_family = AF_UNSPEC

  (não envolve nenhuma comunicação com o servidor!)

Outras vantagens do uso de “*connect”* com sockets UDP*:*

maior eficiência (~30 % medido) pois:

- evita busca na tabela de roteamento para cada datagrama enviado

  (a fim de determinar a interface IP de saída do datagrama)
- a estrutura *sockaddr_in* só é copiada uma vez da aplicação para o kernel UDP

E/S Síncrona x E/S Assíncrona no sistema Unix

Definições do **POSIX:**

Uma operação de E/S é dita *síncrona* se o processo que a invoca fica bloqueado até que ela “se complete”

síncrona ~ bloqueadora

. . . *assíncrona*: o processo que a invoca não fica bloqueado

assíncrona ~ não bloqueadora

Unix possui vários modelos de operações de E/S

a maioria é *síncrona*

(em diferentes graus de bloqueio do processo invocador)

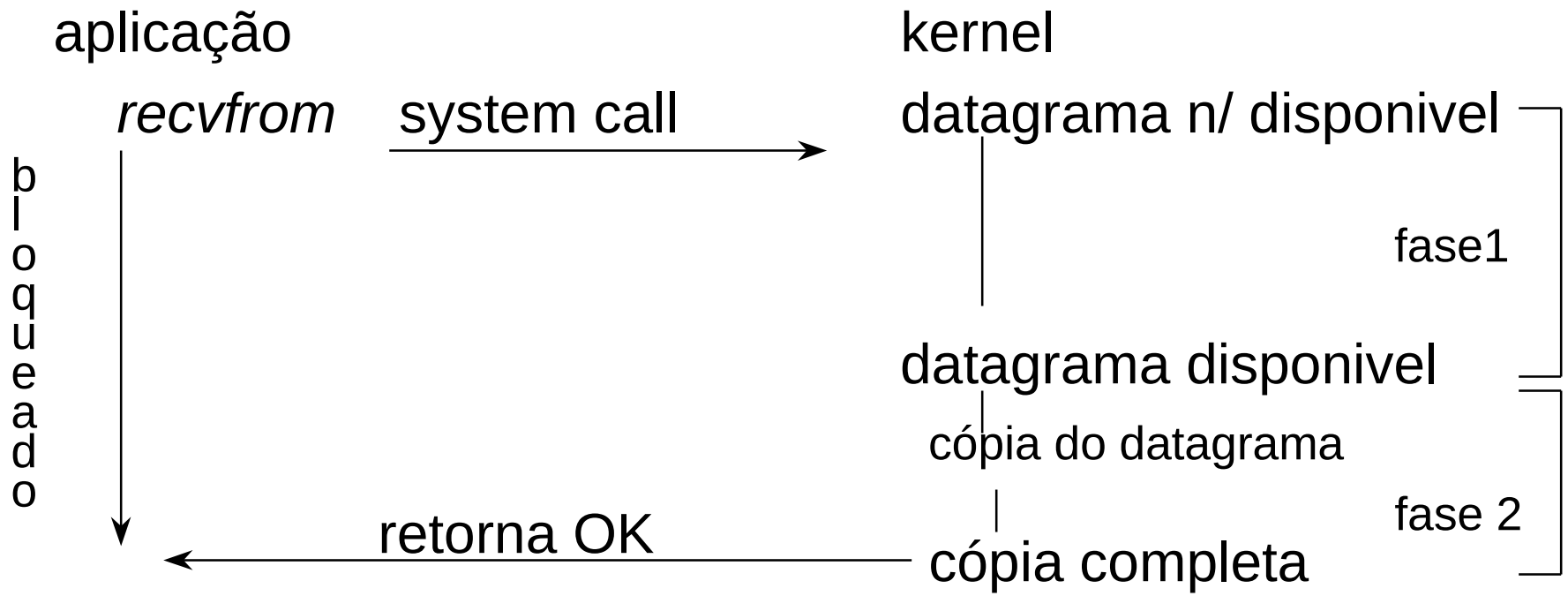
Modelos de E/S do Unix

- Operações de Entrada

  Possuem 2 fases:

  1. espera pelos dados ficarem disponíveis
  2. cópia dos dados do kernel p/ área do processo

Entrada bloqueante (default)

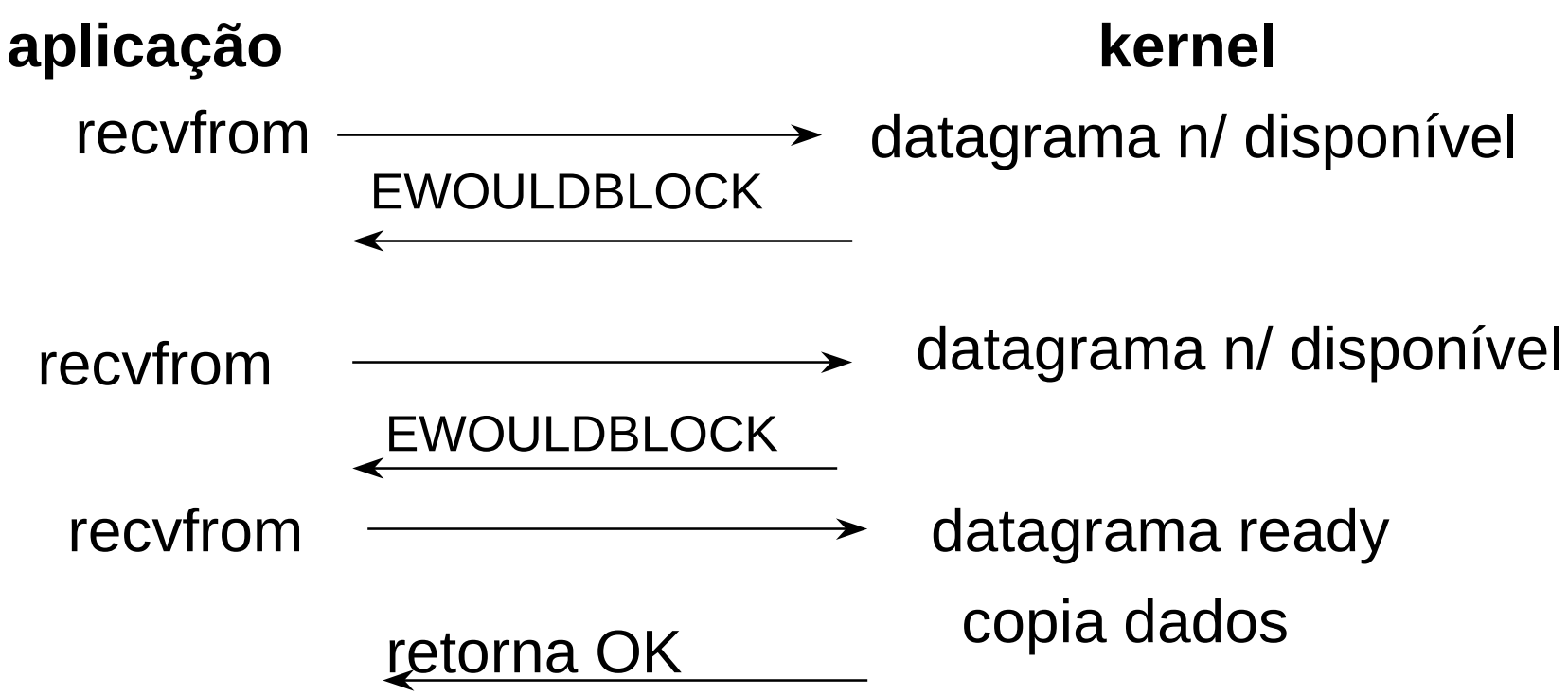


Entrada não bloqueante (precisa ligar opção p socket)

se não pode completar E/S sem processo dormir

kernel retorna (com código de erro)

Ex:



E/S multiplexada **(função *select)***

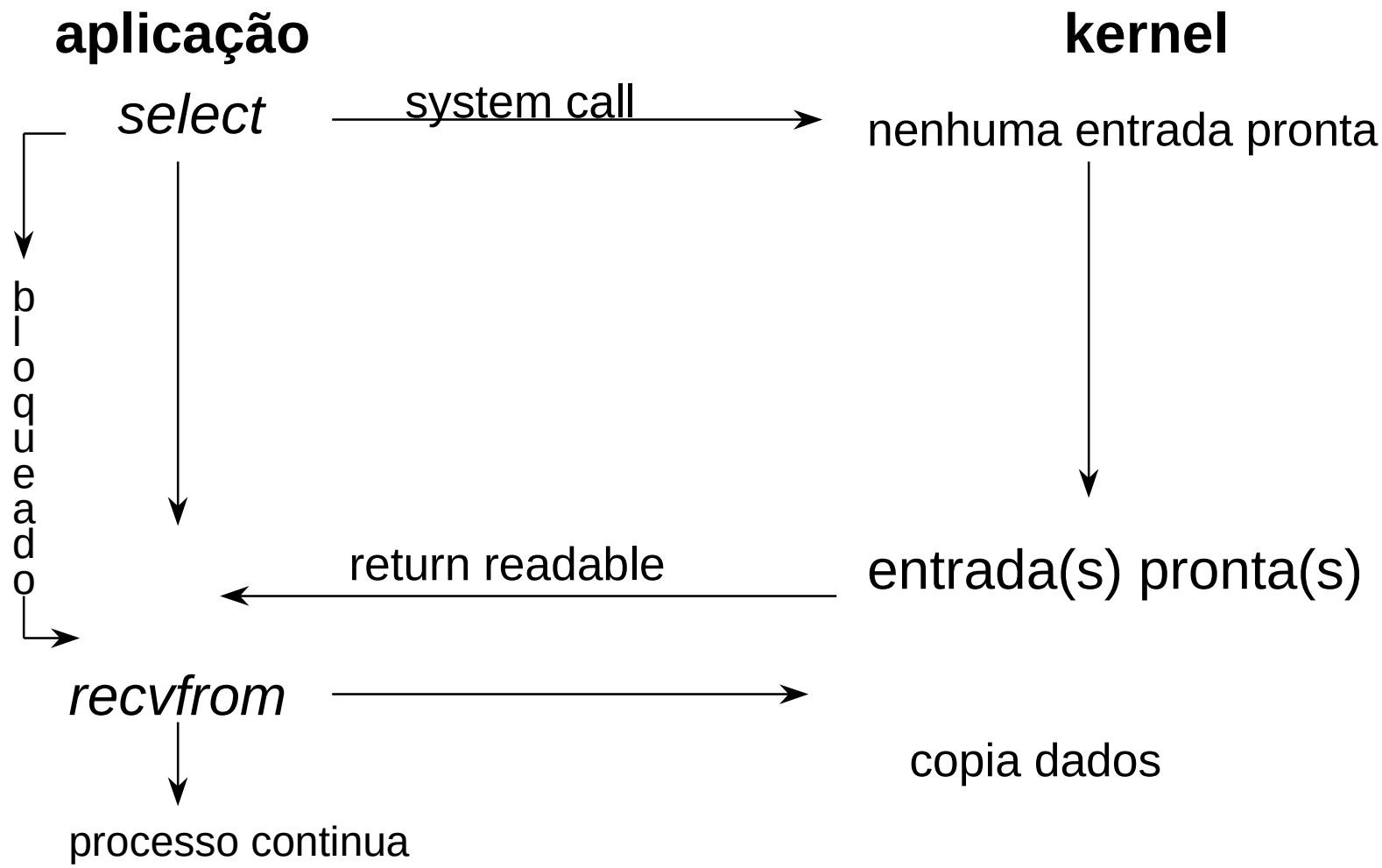
Normalmente usada em operações de entrada,

(bloqueadoras por default),

quando temos 2 ou mais operações a fazer:

- processo bloqueia ao invocar select, até uma das operações ter dados disponíveis (prontos)
- processo faz operação de E/S na entrada pronta
- desta forma entradas ocorrem na medida da sua disponibilidade (uma não bloqueia a outra)

Exemplo:


aplicação
kernel
select
system call
nenhuma entrada pronta
bloqueado
return readable
entrada(s) pronta(s)
recvfrom
copia dados
processo continua

E/S dirigida por sinal

- processo habilita tratamento de sinal SIGIO
- processo habilita socket (via função *fnctl*)
- quando dados dispoíveis kernel envia sinal SIGIO para processo
- rotina de tratamento do sinal faz operação de entrada (ou notifica programa principal:dado está disponível)

E/S assíncrona (Posix realtime extension - via *aio_read*)

- processo inicia operação, continua (kernel controlará)
- quando operação termina, inclusive cópia de dados do kernel para aplicação, processo é notificado via sinal
- tratador do sinal notifica/desvia p/ programa principal

E/S multiplexada: a função *select()*

função *select()*

- kernel bloqueia processo à espera de 1 ou mais eventos
- quando algum evento ocorre acorda processo
- ou, opcionalmente, quando temporizador expira

Eventos podem ser:

- entrada pronta para leitura (fim fase 1)
- saída disonível para escrita
- entrada ou saída gerou exceção (“out of band data”)
- temporizador (para término de operação) expirou

- Eventos podem estar associados a quaisquer descritores de E/S, não apenas *sockets*

Como associar descritores a eventos?

- Via estrutura de dados especial do tipo *fd_set*
  (pode ser imaginada como um mapa de bits),
  onde são inseridos (ligados) descritores de interesse
  por tipo de operação: entrada, saída ou exceção
- temporizador: definido através da conhecida estrutura *timeval { tsecs, tusecs}*

Funções para manipular os conjuntos de descritores:

*void FD_ZERO(fd_set * fdset)* zera todos os bits de fdset

*void FD_SET(int fd, fd_set * fdset)* liga o bit dado por fd (inserção)

*void FD_CLR(int fd, fd_set * fdset)* desliga o bit dado por fd (remoção)

*int FD_ISSET(int fd, fd_set * fdset)*

testa se o descritor dado por fd está pronto, i. é, se a fase 1 da operação de E/S terminou

A função select:

*int select(int maxfd, fd_set * readset, * writeset, * exceptset, timeval * timeout)*

retorna: número total de descritores prontos, ou

0 se timeout, -1 se erro

Obs:

- podemos definir um temporizador genérico, com resolução do relógio do sistema (~ 10ms), pondo NULL para *readset, writeset e exceptset*.
- se timeout for NULL select só retorna quando algum descritor ficar pronto

- se timeout for incializado com 0 secs, 0 usecs

  então select retorna imediatamente, com número de descritores prontos (0 se nenhum)
- maxfd = número de descritores a serem testados =

  valor maior descritor + 1

Como achar maxfd?

resposta: último descritor definido +1

- parâmetros fd_set são passados por *valor-resultado,* i.e. são mudados pelo kernel, ligando os bits dos descritores que estão prontos e desligando os outros.

  consequencia: importante chamar *FD_SET* toda vez que chamamos *select* !

Quando um descritor fica pronto?

Operação de Entrada com socket:

- quando há dados no buffer de recepção do socket

  (>= low water-mark)

  então read não bloqueia e devolve no bytes lidos.
- quando “fim de leitura” é detetado, i.é., a conexão foi fechada para leitura (read retorna 0)
- para *listening socket*: há conexões completadas pelo “3-way handshake”: *accept* não bloqueará
- há um erro pendente no socket:

  read retorna -1 e *errno* é alterada

Operação de saída sobre socket:

- há espaço disponível no buffer de transmissão do socket (>= low water-mark, 2048 default):

  write não bloqueia

- o *socket* foi fechado para escrita:

  write gera o erro SIGPIPE

- há um erro pendente no socket:

  write retorna -1 e variável *errno* é alterada

Operação de exceção

Existe uma condição de exceção pendente caso haja "out of band data" não recebido no socket.

Obs: low water-mark de entrada tem default 0; pode ser mudado para n: processo só será acordado se n ou mais bytes estiverem disponíveis no buffer de leitura

Exemplo de aplicação:

Servidor TCP e UDP iterativo *daytime*

Devolve um string com a data e hora corrente.

Usa a porta 13 tanto para TCP como para UDP

Uma solução inaceitável:

servidor fica num laço onde invoca *accept* (para atender clientes TCP) e *recvfrom* (p/ clientes UDP);

por serem bloqueadoras, os pedidos só serão atendidos de forma alternada para clientes TCP e UDP

Solução com E/S multiplexada:

Incluir os descritores de socket TCP e UDP num conjunto *fd_set* para leitura;

servidor fica num laço onde invoca *select* tendo o conjunto fd_set acima como parâmetro

```
/* srv_udptcp.c -- um exemplo de servidor iterativo TCP e UDP
   usando select                Celio G.       Marco 2000     */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <unistd.h>
#define MYPORT 3490    /*the port users will be connecting to*/

#define BACKLOG 10     /* how many pending connections queue
                          will hold */
#define MAXBUFLEN 128
#define max(x,y) ((x) > (y) ? (x) : (y) )
```

```
int udptcpsocket(int myport, char * protocol)
{
    int sockfd, type;
    struct sockaddr_in my_addr;    /* my address information */
    if (strcmp(protocol, "tcp") ==0)
       type = SOCK_STREAM;
    else
       type = SOCK_DGRAM;
    if ((sockfd = socket(AF_INET, type, 0)) == -1) {
       perror("socket");
       exit(1);
    }
    my_addr.sin_family = AF_INET;         /* host byte order */
    my_addr.sin_port = htons(myport);     /* short, network
                                             byte order */
    my_addr.sin_addr.s_addr = INADDR_ANY; /* automatically fill
                                             with my IP */
    bzero(&(my_addr.sin_zero), 8);        /* zero rest struct*/
```

```
if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct
                        sockaddr)) == -1) {
        perror("bind");
        exit(1);
    }
    if (type == SOCK_STREAM)
     if (listen(sockfd, BACKLOG) == -1) {
        perror("listen");
        exit(1);
     }
    return sockfd;
}
void daytime(char * buf)
{
   char *ctime();
   time_t time(), now;
   (void) time(&now);
   sprintf(buf, "%s", ctime(&now));
}
```

```
main()
{
    fd_set readfds;
    int tsockfd,usockfd,  new_fd;  /* listen on sock_fd, new
                                      connection on new_fd */
    int nfds, sin_size, numbytes;
    char buf[MAXBUFLEN];
    struct sockaddr_in their_addr; /* connector's address
                                      information */
    tsockfd= udptcpsocket(MYPORT, "tcp");
    usockfd= udptcpsocket(MYPORT, "udp");
    nfds = max(tsockfd, usockfd) +1;
    FD_ZERO(&readfds);
    sin_size = sizeof(struct sockaddr_in);
    while(1) {  /* main accept() loop */
      FD_SET(tsockfd, &readfds);
      FD_SET(usockfd, &readfds);
      if (select( nfds, &readfds, NULL, NULL, NULL) < 0){
       perror("select" );
       exit(1);
       }
```

```
if (FD_ISSET(tsockfd, &readfds)) { /*We have a TCP connection*/
        if ((new_fd = accept(tsockfd, (struct sockaddr *)
               &their_addr, &sin_size)) == -1) {
            perror("accept");
        }
        printf("server: got tcp connection from
               %s\n",inet_ntoa(their_addr.sin_addr));
      daytime(buf);
      if (write(new_fd, buf, strlen(buf)) == -1){
               perror("send");
               exit(1);
      }
        printf("sent: %s", buf);
        close(new_fd);
   }
```

```
if (FD_ISSET(usockfd, &readfds)){   /* We have a UDP request */
   if ((numbytes=recvfrom(usockfd,buf,MAXBUFLEN,0,(struct
               sockaddr *) &their_addr,&sin_size)) == -1) {
          perror("recvfrom");
          exit(1);
       }
       daytime(buf);
   if ((numbytes=sendto(usockfd,buf,strlen(buf),0,(struct
        sockaddr *)&their_addr,sizeof(struct sockaddr))) == -1){
          perror("sendto");
          exit(1);
       }
        printf("server: got udp request from: %s sent:
                %s",inet_ntoa(their_addr.sin_addr), buf);
     }
  } /* while */
}
```

O “super-servidor internet” *inetd*

Servidores Unix em geral provêm dezenas de *serviços*

Esses *serviços* podem ser iniciados como *daemons* quando o sistema operacional dá partida;

daemons são processos em geral permanentes, que executam em background funções de apoio ao sistema operacional;

eles não têm um terminal de contrôle.

Muitos serviços são usados esporadicamente por clientes:

seus *daemons* ocupam recursos do sistema (tabela de processos, memória virtual, etc)

Solução mais eficiente a tê-los permanentemente em execução:

disparar esses serviços quando forem requeridos por clientes:

um *daemon* especial ficaria à *“escuta*” de clientes nas “portas conhecidas” pre-definidas para esses serviços, criando um processo que “executa” o serviço quando chega um pedido do cliente

Inetd é o daemon que faz exatamente essas funções.

Serviços gerenciados por *inetd* são registrados no arquivo ascii */etc/inetd.conf,* com o formato:

nome	tipo-socket	protoc	wait/nowait	login	pathname	argumentos
ftp	stream	tcp	nowait	root	/usr/sbin/ftpd	ftpd-l
tftp	dgram	udp	wait	nobody	/usr/sbin/tftpd	tftpd

onde:

nowait significa que o serviço suportará vários clientes simultaneamente, o que é padrão para serviços TCP

wait faz o contrário e é usado pelos serviços UDP

login é o nome do usuário sob o qual o daemon vai rodar

pathname é a localização do código do serviço.

argumentos: a serm usados pelo serviço ao dar partida

No que se segue suporemos que o serviço usará conexão TCP

Funcionamento do daemon *inetd:*

1. Inicialmente *inetd* cria um *socket* para cada serviço definido em *inetd.conf*, com o tipo apropriado.
2. *inetd* chama *getservbyname* com parâmetros *nome* do serviço e *protocolo* (*getservbyname* consulta o arquivo */etc/services* para obter a "porta conhecida" de cada serviço);

   inetd faz um *bind* em cada uma dessas portas conhecidas.
3. *inetd* chama *listen* para cada um dos sockets TCP;
4. *inetd* usa E/S multiplexada, chamando *select* à espera de qualquer um dos sockets associados aos serviços ficarem "prontos" para entrada.
5. Quando *select* retorna (algum descritor está pronto) , *inetd* chama *accept* para obter um socket para a nova conexão;
6. *inetd* cria um processo filho para tratar a nova conexão, fecha o seu socket conectado e volta ao passo 4.
7. o processo filho fecha todos os descritores exceto o do socket conectado

8. Filho duplica o socket conectado sobre os descritores 0, 1 e 2 (usando a função *dup2*) e fecha o socket conectado original;
9. Se o nome de login do serviço não for *root*, filho chama *getpwnam* para obter a entrada correspondente no arquivo de passwords e chama *setgid* e *setuid* para mudar o grupo e usuário do serviço.
10. Filho dá *exec* no programa especificado em *pathname* com os argumentos correspondentes.

    A partir desse ponto o processo filho tem sua imagem (código e dados) substituída pela do programa “execed”, que herda os descritores de socket abertos 0, 1 e 2, com os quais pode se comunicar com o cliente.
11. Para obter o valor do “socket remoto” do cliente, o servidor “execed” deve chamar a função *getpeername* usando o socket conectado, pois a área de dados do processo filho que o disparou foi perdida.

Serviços UDP diferem nos passos 3 e 6 anteriores:

- o passo 3 não é executado para sockets UDP
- no passo 6 o pai salva o *pid* do filho, desabilita via FD_CLR o descritor desse socket para futuros *selects*:

  a partir desse ponto o filho monopoliza o descritor para comunicação com o cliente
- o pai habilita uma rotina de tratamento do sinal SIGCHLD e volta ao passo 4.
- Quando o filho termina, gerando o sinal SIGCHLD, a rotina de tratamento obtem o *pid* do filho e reabilita *selects* para o descritor do socket correspondente via FD_SET.

Os procedimentos acima são de um servidor UDP iterativo.

Veremos a seguir que o servidor UDP do Trivial File Transfer Protocol (TFTP) é mais esperto, implementando um serviço concorrente de transferência de arquivos com clientes.

Exemplo: Servidor UDP concorrente *tftpd*

- *tftpd:* servidor para o “trivial file transfer protocol” (TFTP)
- utiliza o protocolo UDP com a “porta conhecida” 69 para transferência de arquivos entre um cliente e um servidor;
- disparado por *inetd* ao ficar pronto para entrada o descritor de socket UDP criado para esse serviço;
- *tftpd* invoca *recvfrom* para receber o pedido do cliente em pacote contendo o *pathname* do arquivo e a direção da transferência;
- *tftpd* cria um processo filho para tratar a transferência do arquivo e termina (via *exit()*); desta forma *inetd*, ao receber um pedido de de outro cliente na porta 69, dispara uma nova instância de *tftpd*;
- processo filho muda o *“user id*” e *“group id”* para o do usuário *“nobody”* (pois herdou privilégios de *root* via *inetd*)

- o processo filho cria um descritor de socket UDP para tratar a transferência,
- chama *bind* passando esse descritor e uma estrutura *sockaddr* inicializada com 0, de forma a obter um “socket local” unívoco para a transferência,
- chama *connect* passando a estrutura *sockaddr* obtida pelo pai ao receber via *recvfrom* o pedido do cliente (a partir desse ponto a comunicação com o cliente será feita via *recv* ou *send*)
- *tftpd* envia resposta ao cliente contendo o seu novo “socket local” (embutido no cabeçalho do datagrama UDP)
- cliente utiliza o “socket remoto” obtido na resposta do servidor para comunicação com o mesmo a partir desse ponto.

Obs: é possível transferência de arquivo simultânea com outro cliente pois um novo processo seria criado pela nova instância de *tftpd* com um socket local distinto para a transferência

Instalação de *daemons*

Daemons devem executar silenciosamente em background.
Eles não devem ser interrompidos:

- por sinais vindos de um “terminal controlador”, ou
- por sinais destinados ao *grupo de processo* ou à *sessão* a que *daemon* possa pertencer;

Não devem enviar mensagens de erro ou avisos ao console ou a um terminal. Tais mensagens deveriam ser registradas num *arquivo de log* apropriado;
Devem também ser robustos quanto a erros que possam ocorrer com o sistema ou com a aplicação.
Por essas razões *daemons* devem ser instalados com certos cuidados:
é extremamente simples fazê-lo (algumas linhas em C),
mas elas parecem mágica sem uma explicação adequada.

A mágica consiste em três medidas:

desassociar o *daemon*:

- de um terminal controlador,
- de grupos de processo ou de sessão a que possa pertencer quando criado,
- enviar mensagens de erro ou de aviso emitidas pelo *daemon* a um *arquivo de log centralizado*

Para entender as duas 1as medidas temos que entender dois conceitos pouco conhecidos do Unix:

- grupo de processos (vamos usar o mnemônico *gpr*)
- *sessão*

Um *gpr* é uma coleção de um ou mais processos.

Todo processo pertence a um *gpr* e tem um identificador de *gpr*

além do seu identificador de processo (*pid*)

Se o *pid* de um processo é igual ao do seu *gpr* então êle é dito

lider do gpr

O criador de um *gpr* é o líder do *gpr*;

se êle termina, o *gpr* fica sem lider mas continua a existir;

sinais enviados a um *gpr* são enviados a cada processo do *gpr*

Uma *sessão* é uma coleção de *gprs;*

é criada por um processo que não é lider de gpr ao chamar *setsid()*

O criador de uma sessão torna-se:

- lider da *sessão*,
- lider de um *gpr*
- não tem um terminal controlador

Um processo herda o *gpr* e a *sessão* do seu pai

daemons podem ser iniciados de diversas formas:

- por *scripts* na partida do sistema operacional,
- a partir de um comando emitido de uma *shell*,
- pelo daemon *crond* ou pelo comando *at* que executa um comando em uma data/hora especificada

Em qualquer um dos casos acima, gostaríamos que o *daemon* funcionasse satisfazendo os 3 requisitos vistos anteriormente

Os seguintes passos, se executados pelo daemon, vão garantir os requisitos desejados:

1. *daemon* chama *fork*, pai termina (via *exit*()), filho continua:
   o filho herda o *grp* do pai e como tem um novo pid não pode ser um *lider de gpr*
2. Filho chama *setsid(),* criando uma sessão e tornando-se:
   líder da sessão, lider de *gpr* e sem terminal controlador!
   (veja criação de sessão no slide anterior)

O processo filho, nosso candidato a *daemon*, sendo lider de sessão poderia abrir um device tipo terminal que automaticamente se tornaria controlador da nova *sessão*. Para evitar que isso ocorra, basta repetir o passo 1 acima:

3. ele chama *fork(),* pai termina e filho continua

   (na verdade continua o *neto* do processo original!)

   como o novo filho não é mais lider de *gpr,* nem de *sessão* e não tem um terminal controlador ele satisfaz o que queríamos!

Obs: o passo 3 é uma precaução adicional e não é essencial

Outras medidas menos mágicas são:

- o daemon deve fechar todos os seus descritores de arquivos, em especial 0, 1 e 2
- deve executar num diretório que não seja parte de um *filesystem* que precise ser “umounted” pelo sistema, por exemplo em */tmp*

- deve especificar uma “máscara de criação de arquivos” diferente da default, 022, que impede que arquivos criados pelo *daemon* sejam escritos por outros processos, independentemente dos direitos que o *daemon* especifique.
- deve ignorar o sinal SIGHUP, chamando

  signal(SIGHUP, SIG_IGN) a fim de evitar que quando o pai termina no 2º fork, este sinal mate o filho (Linux não precisa)
- deve habilitar o log de mensagens através do daemon *syslogd*: para esse fim duas funções auxiliares podem ser usadas,

*openlog (char * pname, int options, int facility)*

pname é um string a ser colocado em cada mensagem do log, em geral conterá o nome do programa (do *daemon*),

options: “ou” de algumas constantes, LOG_PID é a mais comum

facility: em geral 0

A segunda função, *syslog,* faz a entrada da mensagem no arquivo de log pre-definido para *syslogd*:

*syslog(int priority, const char * message, …);*

priority: “ou” de constantes para o *nível* e *facility* da mensagem de a fim de agrupar as mensagens de forma apropriada para *syslogd*

message é uma cadeia para formatação como a do comando *printf*; pode conter o formatador *%m* que será substituído pela mensagem de erro correspondente a *errno*.

Exemplo:

syslog(LOG_INFO|LOG_USER, “rename(%s, %s) : %m, file1,file2);

registra no *log* do sistema a causa do erro ao se tentar *rename* de *file1* para *file2*.

Obs: no linux o arquivo de log do *syslogd* é */var/log/messages*

não pode ser lido por usuários comuns, mas pode ser escrito!

Exemplo de *daemon*

```
/* mydaemon.c -- Um exemplo de daemon a nivel de usuario
   daemon_init adaptado de Richard Stevens, Unix Network
   Programming vol 2, p 306    Celio G.       Abril 2000    */
#include       "myunp.h"
#include       <syslog.h>
#define MAXFD   64
void mysyslog(char * progname);
void daemon_init(const char *pname)
{
   int         i;
   pid_t       pid;
   if ( (pid = fork()) != 0)
        exit(0);                        /* parent terminates */
   /* 1st child continues */
   setsid();                    /* become session leader */
   signal(SIGHUP, SIG_IGN);     /* when 1st child dies, may kill
                                   his child*/
   if ( (pid = fork()) != 0)
        exit(0);                /* 1st child terminates */
```

```
/* 2nd child continues */
   chdir("/tmp");              /* change working directory */
   umask(0);           /* clear our file mode creation mask */
   for (i = 0; i <  MAXFD; i++) /*close all file descriptors*/
        close(i);
   openlog(pname, LOG_PID, 0); /*tell syslog to log proc pid*/
}
int main(int argc, char * argv[])
{
  int i=1;
  printf("Comecando...\n");
  daemon_init( argv[0] );     /* instals me as a daemon */
  for (; ;){
  mysyslog(argv[0]);          /* log program name */
  sleep(20);                  /* log one line every 20 secs*/
  }
}
```

```
void mysyslog(char *progname) /*appends message to log file*/
{
  FILE * fp;
  char buf[64],  * ctime();
  time_t time(), now;
  (void) time(&now);       /*get seconds since Epoch into now*/
  sprintf(buf, "%s", ctime(&now)); /* change secs to  date
                                      and time, put in buf */
  if ((fp= fopen("ERROR.LOG", "a")) ==0) /* private log file*/
    exit(1);
  /* syslog(LOG_INFO|LOG_USER, "%s  %s %m\n",progname, buf);*/
  fprintf(fp, "%s  %s\n", progname, buf);
  fclose(fp);
}
```

Unix Domain Sockets

O que são:

- Alternativa a pipes para comunicação entre processos (IPC) não relacionados por parentesco,
- Comunicação full duplex ao contrário de (named) pipes
- Processos participantes devem residir na mesma máquina
- Suporta sockets tipo datagrama e TCP (stream)

Vantagens:

- mesma API de sockets
- velocidade ~ 2 x maior que sockets Internet v4

  (usado pelo sistema Xwindows)
- permite passar descritores quaisquer entre processos não relacionados (típico de aplicações Cliente - Servidor)
- permite passar credenciais (user id, group id) do cliente a um servidor (autenticação, maior segurança)

Identificação da conexão entre processos participantes:

- via “pathname” do sistema de arquivos
- contida na estrutura *sockaddr_un:*

```
#include <sys/un.h>
struct sockaddr_un {
    unsigned short int sun_family; /*AF_LOCAL ou AF_UNIX*/
     char                    sun_path[108]; } /*null terminated name*/
```

- Exemplo de inicialização (servidor TCP):

```
struct sockaddr_un serv_addr;
char unixdomainpath[]=“/tmp/mypath”; /* deve permitir r/w */
sockfd= Socket(AF_LOCAL, SOCK_STREAM, 0);
unlink(unixdomainpath);                    /*caso exista, remova*/
bzero(&serv_addr,sizeof(serv_addr));
serv_addr.sun_family= AF_LOCAL;
strcpy(serv_addr.sun_path, unixdomainpath);
Bind(sockfd,(struct sockaddr *)&serv_addr, sizeof(serv_addr));
```

A partir desse ponto servidor TCP fará as chamadas habituais:

listen

accept

send

recv

close

No caso de um Unix Domain socket tipo datagrama:

- cliente *deve* inicializar sua estrutura *sockaddr_un,*
  colocando uma cadeia vazia no campo *sun_path:*

  strcpy(client_addr.sun_path, “”);

  client_addr.sun_family=AF_LOCAL;
- *deve* então chamar *bind*
  (o que não era obrigatório com um socket Internet v4 !)
- o cliente procede de forma habitual a partir desse ponto,
  chamando:
  connect (opcional), *sendto, recvfrom, close*