联航精英训练营

UNIGRESS ELITE TRAINING CAMP

Linux高级编程（八）

张勇涛

SOCKET编程

预备知识

字节序

- 由于不同的计算机系统采用不同的字节序存储数据,同样一个4字节的32位整数,在内存中存储的方式就不同. 字节序分为小端字节序(Little Endian)和大端字节序(Big Endian), Intel处理器大多数使用小端字节序, Motorola处理器大多数使用大端(Big Endian)字节序;

小端字节序例子

一个4字节的值为0x1234567的整数与高低字节对应关系:

01	23	45	67
Byte3	Byte2	Byte1	Byte0
高位字节----------低位字节			

将在内存中按照如下顺序排放：

内存地址序号	字节在内存中的地址	16进制值
0x03	Byte3	01
0x02	Byte2	23
0x01	Byte1	45
0x00	Byte0	67

大端字节序例子

一个4字节的值为0x1234567的整数与高低字节对应关系:

01	23	45	67
Byte3	Byte2	Byte1	Byte0
高位字节----------低位字节			

将在内存中按照如下顺序排放：

内存地址序号	字节在内存中的地址	16进制值
0x03	Byte0	67
0x02	Byte1	45
0x01	Byte2	23
0x00	Byte3	01

网络字节序

- TCP/IP协议规定，网络数据流应采用**大端字节序**，即**低地址高字节**.

- 为使网络程序具有可移植性，使同样的C代码在大端和小端计算机上编译后都能正常运行，可以调用以下库函数做网络字节序和主机字节序的转换。

网络字节序和主机字节序转换

```
#include <arpa/inet.h>

uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

这些函数名很好记，h表示host，n表示network，l表示32位长整数，s表示16位短整数。例如htonl表示将32位的长整数从主机字节序转换为网络字节序，例如将IP地址转换后准备发送。如果主机是小端字节序，这些函数将参数做相应的大小端转换然后返回，如果主机是大端字节序，这些函数不做转换，将参数原封不动地返回。

htons()——“Host to Network Short”
主机字节顺序转换为网络字节顺序（对无符号短型进行操作2bytes）

htonl()——“Host to Network Long”
主机字节顺序转换为网络字节顺序（对无符号长型进行操作4bytes）

ntohs()——“Network to Host Short”
网络字节顺序转换为主机字节顺序（对无符号短型进行操作2bytes）

ntohl()——“Network to Host Long ”
网络字节顺序转换为主机字节顺序（对无符号长型进行操作4bytes）

地址格式转换

Linux提供将点分格式的地址转于长整型数之间的转换函数。如： inet_addr()能够把一个用数字和点表示IP地址的字符串转换成一个无符号长整型。

inet_ntoa()（“ntoa”代表“Network to ASCII”）；

包括：inet_aton, inet_ntoa, inet_addr等。

IP地址的转换

- inet_aton()
  - 将strptr所指的字符串转换成32位的网络字节序二进制值

```
#include <arpa/inet.h>
int inet_aton(const char *strptr,struct in_addr *addrptr);
```

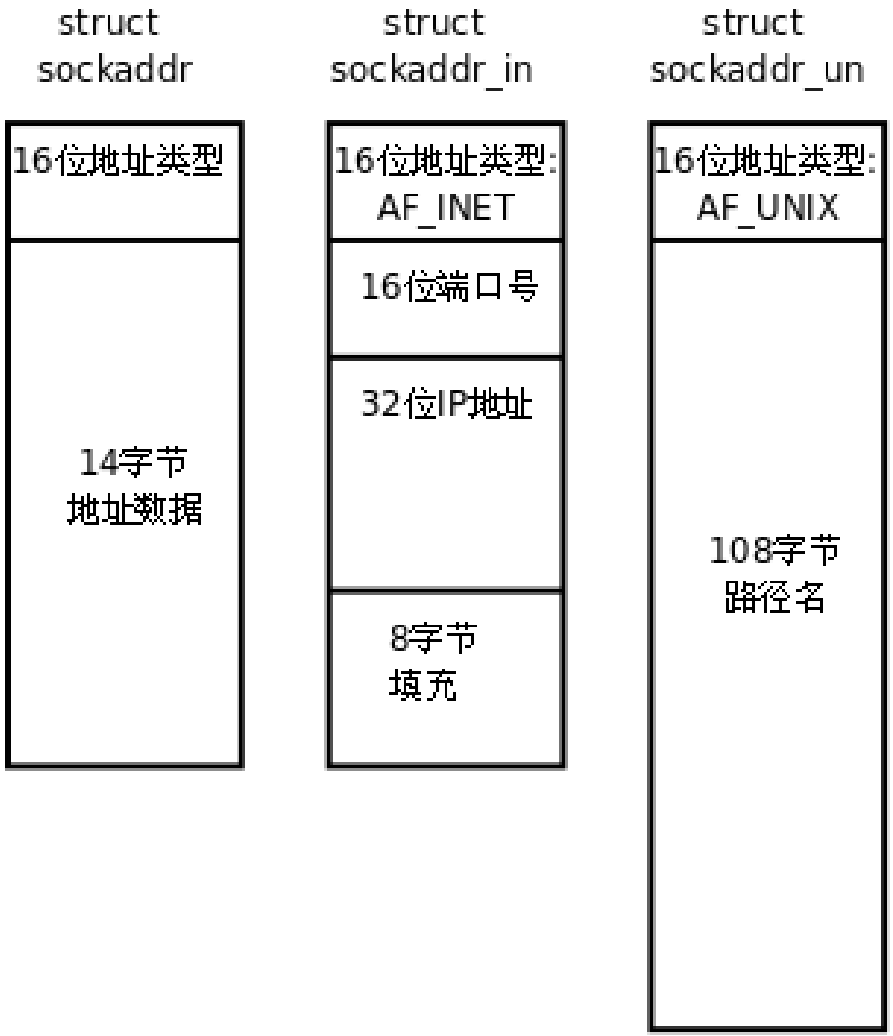
- inet_addr()
  - 功能同上，返回地址。

```
int_addr_t inet_addr(const char *strptr);
```

- inet_ntoa()
  - 将32位网络字节序二进制地址转换成点分十进制的串。

```
char *inet_ntoa(stuct in_addr inaddr);
```

socket地址的数据类型



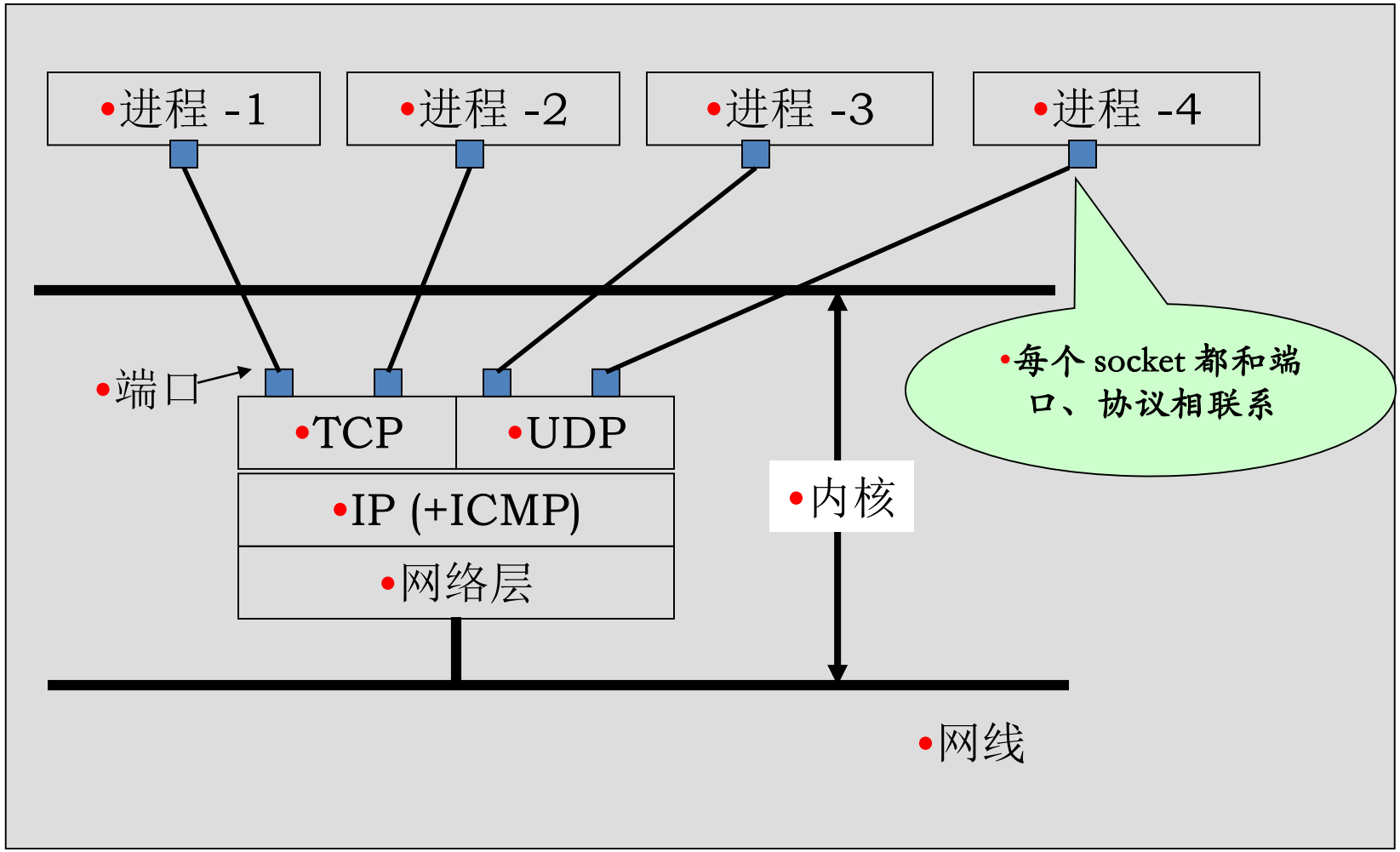
IPv4地址用sockaddr_in结构体表示，包括16位端口号和32位IP地址，IPv6地址用sockaddr_in6结构体表示，包括16位端口号、128位IP地址和一些控制字段。UNIX Domain Socket的地址格式定义在sys/un.h中，用sockaddr_un结构体表示。

端口号

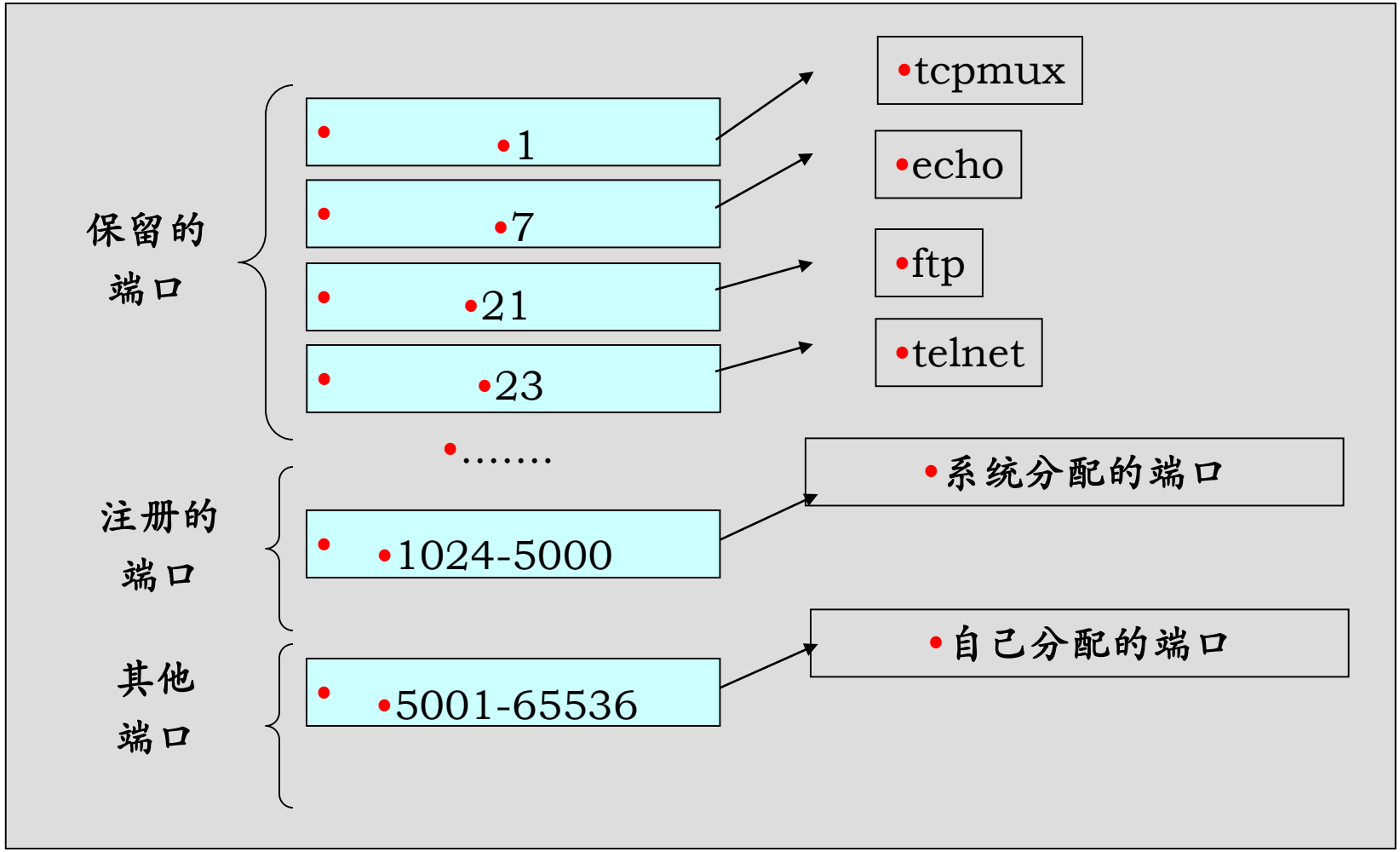
- 为了区分一台主机接收到的数据包应该递交给哪个进程来进行处理，使用端口号
- TCP端口号与UDP端口号**独立**
- 端口号一般由**IANA** (Internet Assigned Numbers Authority) 管理
  - 众所周知端口：1~1023，1~255之间为大部分众所周知端口，256~1023端口通常由UNIX占用
  - 注册端口：1024~49151
  - 动态或私有端口：49151~65535

用“netstat -n”命令，以数字格式显示地址和端口信息

套接字和端口



套接字和端口



为什么需要socket

- 普通的I/O操作过程
  - 打开文件 - > 读/写操作 - > 关闭文件
- TCP/IP协议被集成到操作系统的内核中，引入了新型的“I/O”操作
  - 进行网络操作的两个进程在不同的机器上，如何连接?
  - 网络协议具有多样性，如何进行统一的操作
- 需要一种通用的网络编程接口：Socket

socket的概念

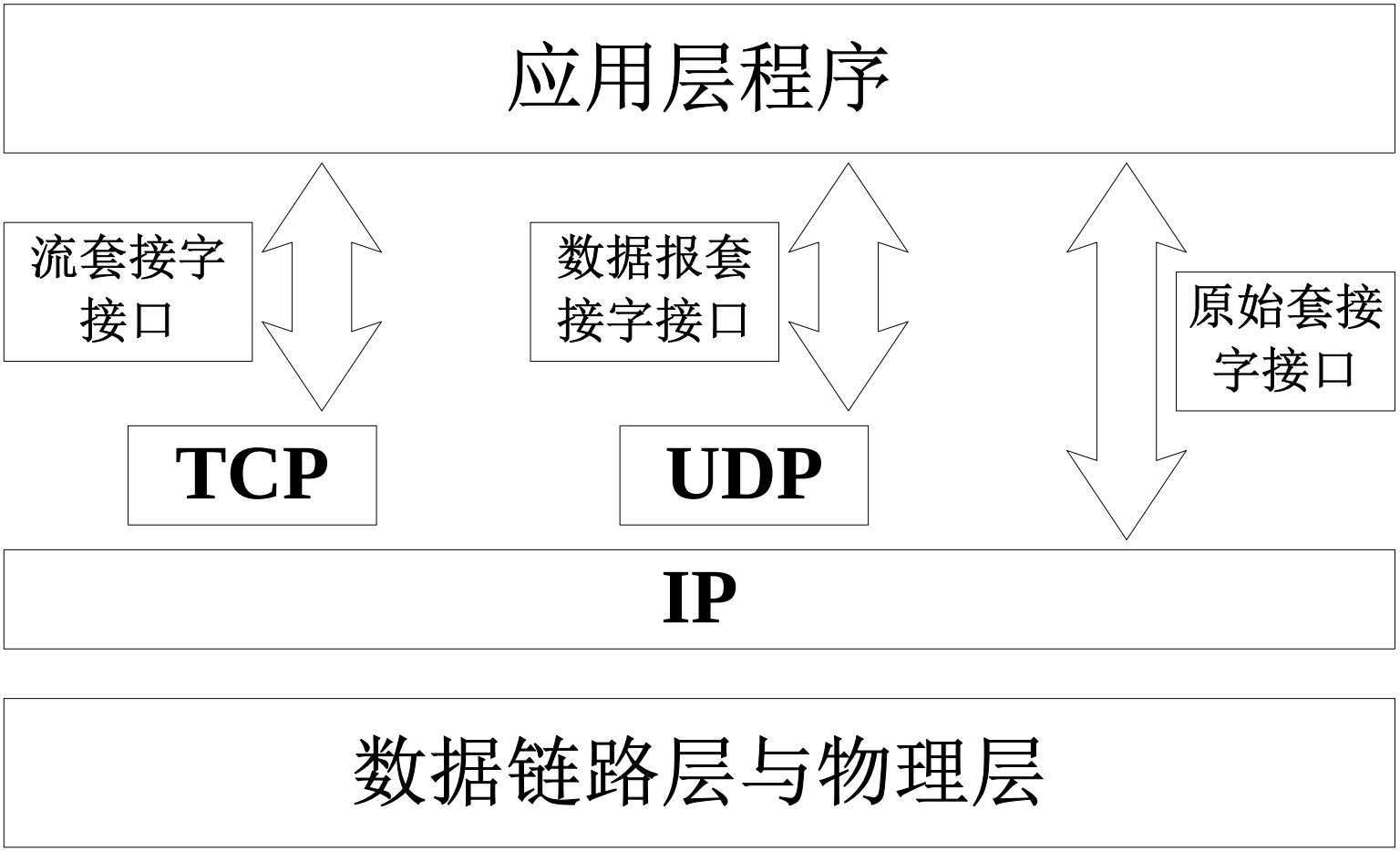
socket这个词可以表示很多概念：

1. 在TCP/IP协议中，“IP地址+TCP或UDP端口号”唯一标识网络通讯中的一个进程，“IP地址+端口号”就称为socket。
2. 在TCP协议中，建立连接的两个进程各自有一个socket来标识，那么这两个socket组成的socket pair就唯一标识一个连接。socket本身有“插座”的意思，因此用来描述网络连接的一对一关系。
3. TCP/IP协议最早在BSD UNIX上实现，为TCP/IP协议设计的应用层编程接口称为socket API。

socket类型

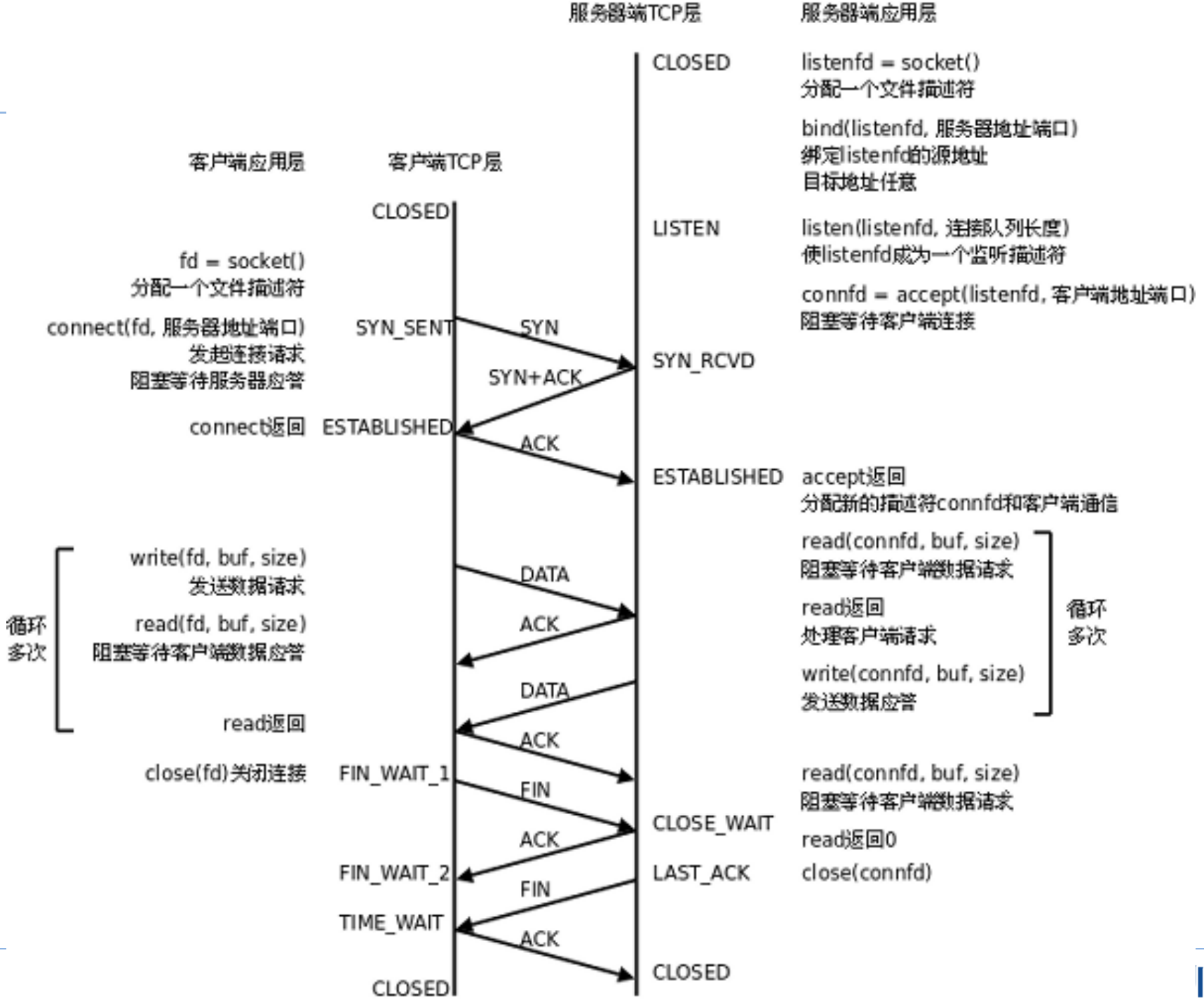
- 流式套接字(SOCK_STREAM)
  - 提供了一个面向连接、可靠的数据传输服务，数据无差错、无重复的发送且按发送顺序接收。内设置流量控制，避免数据流淹没慢的接收方。数据被看作是字节流，无长度限制。
- 数据报套接字(SOCK_DGRAM)
  - 提供无连接服务。数据包以独立数据包的形式被发送，不提供无差错保证，数据可能丢失或重复，顺序发送，可能乱序接收。
- 原始套接字(SOCK_RAW)
  - 可以对较低层次协议，如IP、ICMP直接访问。

socket的位置

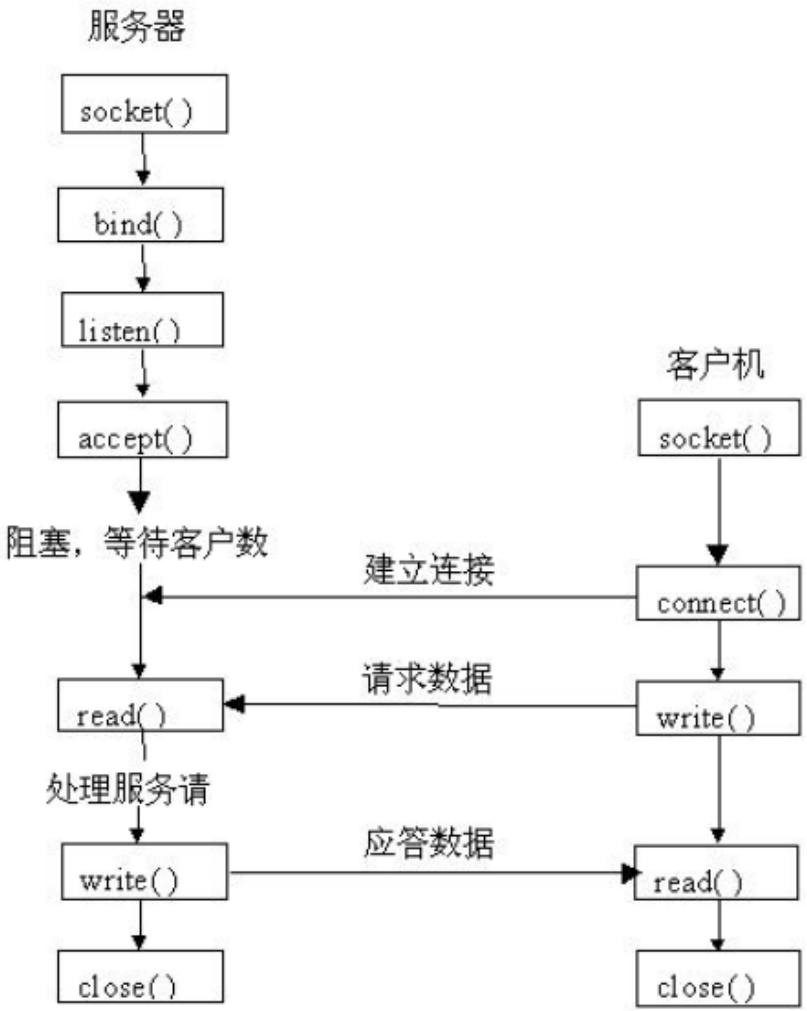


基于TCP协议的网络程序

TCP协议通讯流程


服务器端TCP层
服务器端应用层
CLOSED
listenfd = socket()
分配一个文件描述符
bind(listenfd, 服务器地址端口)
绑定listenfd的源地址
目标地址任意
客户端应用层
客户端TCP层
CLOSED
LISTEN
listen(listenfd, 连接队列长度)
使listenfd成为一个监听描述符
fd = socket()
分配一个文件描述符
connnfd = accept(listenfd, 客户端地址端口)
阻塞等待客户端连接
connect(fd, 服务器地址端口)
发起连接请求
阻塞等待服务器应答
SYN_SENT
SYN
SYN_RCVD
SYN+ACK
connect返回
ESTABLISHED
ACK
ESTABLISHED
accept返回
分配新的描述符connnfd和客户端通信
write(fd, buf, size)
发送数据请求
DATA
read(connnfd, buf, size)
阻塞等待客户端数据请求
循环
多次
read(fd, buf, size)
阻塞等待客户端数据应答
ACK
read返回
处理客户端请求
循环
多次
DATA
write(connnfd, buf, size)
发送数据应答
read返回
ACK
close(fd)关闭连接
FIN_WAIT_1
FIN
read(connnfd, buf, size)
阻塞等待客户端数据请求
CLOSE_WAIT
ACK
read返回0
FIN_WAIT_2
LAST_ACK
close(connnfd)
FIN
TIME_WAIT
ACK
CLOSED
CLOSED

基于数据流的socket编程流程



最简单的TCP网络程序

- 见server1.c

socket

int **socket**(int family, int type, int protocol);

socket()打开一个网络通讯端口，如果成功的话，就像open()一样**返回一个文件描述符**，应用程序可以像读写文件一样用read/write在网络上收发数据，**如果socket()调用出错则返回-1**。

对于IPv4，family参数指定为**AF_INET**。

对于TCP协议，type参数指定为**SOCK_STREAM**，表示面向流的传输协议。

如果是UDP协议，则type参数指定为**SOCK_DGRAM**，表示面向数据报的传输协议。

protocol参数的介绍从略，指定为0即可。

socket（续）

- int socket (int domain, int type, int protocol);
  - domain 是地址族
    - AF_INET // internet 协议
    - AF_UNIX // unix internal协议
  - type
    - SOCK_STREAM // 流式 socket
    - SOCK_DGRAM // 数据报 socket
    - SOCK_RAW // raw socket
  - protocol 参数通常置为0

bind

- int **bind**(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
- bind()成功返回0，失败返回-1。
- 服务器程序所监听的**网络地址**和**端口号**通常是固定不变的，客户端程序得知服务器程序的地址和端口号后就可以向服务器发起连接，因此服务器需要调用bind绑定一个固定的网络地址和端口号。
- bind()的作用是将参数**sockfd**和**myaddr**绑定在一起，使sockfd这个用于网络通讯的文件描述符监听myaddr所描述的地址和端口号。
- struct sockaddr *是一个**通用指针类型**，myaddr参数实际上可以接受多种协议的sockaddr结构体，而它们的长度各不相同，所以需要第三个参数addrlen指定结构体的长度。

sockaddr_in 初始化

- bzero(&servaddr, sizeof(servaddr));
  servaddr.sin_family = AF_INET;
  servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
  servaddr.sin_port = htons(SERV_PORT);

- 首先将整个结构体清零，然后设置地址类型为**AF_INET**，网络地址为**INADDR_ANY**，这个宏表示本地的任意IP地址，因为服务器可能有多个网卡，每个网卡也可能绑定多个IP地址，这样设置可以在所有的IP地址上监听，直到与某个客户端建立了连接时才确定下来到底用哪个IP地址，端口号为**SERV_PORT**，我们定义为8000。

The "*sock_addr*" structure

Step 1: 初始化该数据结构

```
struct sockaddr_in   my_addr;        /* My (client) Internet address      */
```

⋮

```
/* Set My(client's) IP Address ------------------------------------ */
my_addr.sin_family      = AF_INET;        /* Address Family To Be Used */
my_addr.sin_port        = htons (8000);   /* Port number to use */
my_addr.sin_addr.s_addr = htonl (INADDR_ANY);    /* My IP address */
```

Step 2: 填充信息

这些数据结构的一般用法

1. 首先，定义一个sockaddr_in的结构实例，并将它清零。比如：

```
struct sockaddr_in  addr;
memset(&addr,0,sizeof(struct  sockaddr_in));
```

2. 然后，为这个结构赋值，比如：

```
addr.sin_family=AF_INET;
addr.sin_port=htons(8000);
addr.sin_addr.s_addr=htonl(INADDR_ANY);
```

3. 第三步：在函数调用中使用时，将这个结构强制转换为sockaddr类型。如：

```
bind(listenfd, (sockaddr*)(&addr),&addrlen);
accept(listenfd,(sockaddr*)(&addr),&addrlen);
```

listen

- int listen(int sockfd, int backlog);
- listen()成功返回0，失败返回-1。

- 典型的服务器程序可以同时服务于**多个客户端**，当有客户端发起连接时，服务器调用的accept()返回并接受这个连接，如果有大量的客户端发起连接而服务器来不及处理，尚未accept的客户端就处于连接等待状态，**listen()声明sockfd处于监听状态**，并且**最多**允许有backlog个客户端处于连接待状态，如果接收到更多的连接请求就忽略。

accept

- int accept(int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);

  三方握手完成后，服务器调用**accept()**接受连接，如果服务器调用accept()时还没有客户端的连接请求，就**阻塞等待直到有客户端连接上来。**

- **cliaddr是一个传出参数**，accept()返回时传出**客户端的地址和端口号。**

- **addrlen**参数是一个**传入传出参数**（value-result argument），传入的是调用者提供的缓冲区cliaddr的长度以避免缓冲区溢出问题，传出的是客户端地址结构体的实际长度（有可能没有占满调用者提供的缓冲区）。
- 如果给cliaddr参数传NULL，表示不关心客户端的地址。

服务器程序结构

```
while (1)
 {
     cliaddr_len = sizeof(cliaddr);
     connfd = accept(listenfd,
                     (struct sockaddr *)&cliaddr, &cliaddr_len);
     n = read(connfd, buf, MAXLINE);
     ...
     close(connfd);
}
```

整个是一个while死循环，**每次循环处理一个客户端连接。**

由于**cliaddr_len是传入传出参数**，每次调用accept()之前应该重新赋初值。

accept()的参数listenfd是先前的监听文件描述符，而accept()的返回值是另外一个文件描述符connfd，之后与客户端之间就通过这个connfd通讯，最后关闭connfd断开连接，而不关闭listenfd，再次回到循环开头listenfd仍然用作accept的参数。

accept()成功返回一个文件描述符，出错返回-1。

客户端程序

- client1.c
- 由于客户端不需要固定的端口号，因此不必调用bind()，客户端的端口号由内核自动分配。
- 注意，客户端不是不允许调用bind()，只是没有必要调用bind()固定一个端口号，**服务器也不是必须调用bind()，但如果服务器不调用bind()，内核会自动给服务器分配监听端口，每次启动服务器时端口号都不一样，客户端要连接服务器就会遇到麻烦。**

- int **connect**(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
- connect()成功返回0，出错返回-1。
- 客户端需要调用connect()连接服务器，**connect和bind的参数形式一致，区别在于bind的参数是自己的地址，而connect的参数是对方的地址。**

执行过程

先编译运行服务器:

```
$ ./server
 Accepting connections ...
```

然后在另一个终端里用**netstat**命令查看:

```
$ netstat -apn|grep 8000
 tcp        0      0 0.0.0.0:8000            0.0.0.0:*               LISTEN
```

8148/server可以看到server程序监听8000端口，IP地址还没确定下来。

现在编译运行客户端:

```
$ ./client abcd
Response from server:
ABCD
```

错误处理

- 见wrap.c

读写控制

- 见client2.c

使用fork并发处理多个client的请求

- 服务器通常是要同时服务多个客户端的
- 网络服务器通常用fork来同时服务多个客户端，父进程专门负责监听端口，每次accept一个新的客户端连接就fork出一个子进程专门服务这个客户端。

- 但是子进程退出时会产生**僵尸进程**，父进程要注意处理**SIGCHLD**信号和调用**wait**清理僵尸进程。

例子

```
listenfd = socket(...);
bind(listenfd, ...);
listen(listenfd, ...);
while (1)
{
      connfd = accept(listenfd, ...);
      n = fork();
      if (n == -1)
     {
            perror("call to fork");
            exit(1);
      } else if (n == 0)
      {
            close(listenfd);
            while (1)
            {
                  read(connfd, ...);
                  ...
                  write(connfd, ...);
            }
            close(connfd);
            exit(0);
      } else
            close(connfd);
}
```

小测试

- 首先启动server，然后启动client，然后用Ctrl-C使server终止，这时马上再运行server
- $ ./server
- bind error: Address already in use
- 这是因为：虽然server的应用程序终止了，但TCP协议层的连接并没有完全断开，因此不能再次监听同样的server端口。
- 用netstat命令查看一下： netstat -apn |grep 8000
- 原因：server终止时，socket描述符会自动关闭并发FIN段给client，client收到FIN后处于CLOSE_WAIT状态，但是client并没有终止，也没有关闭socket描述符，因此不会发FIN给server，因此server的TCP连接处于FIN_WAIT2状态。

setsockopt

- client终止时自动关闭socket描述符，server的TCP连接收到client发的FIN段后处于TIME_WAIT状态。**TCP协议规定**：主动关闭连接的一方要处于TIME_WAIT状态，等待两个MSL（maximum segment lifetime）的时间后才能回到CLOSED状态，因为我们先Ctrl-C终止了server，所以**server是主动关闭连接的一方**，在TIME_WAIT期间仍然不能再次监听同样的server端口。MSL在RFC1122中规定为两分钟，但是各操作系统的实现不同，在Linux上**一般经过半分钟后就可以再次启动server了。**
- **在server的TCP连接没有完全断开之前**不允许重新监听是不合理的，因为，TCP连接没有完全断开指的是**connfd**（127.0.0.1:8000）没有完全断开，而我们重新监听的是**listenfd**（0.0.0.0:8000），虽然是占用同一个端口，但IP地址不同，**connfd**对应的是与某个客户端通讯的一个具体的IP地址，而**listenfd**对应的是**wildcard address。**
- 解决这个问题的方法是使用**setsockopt()**设置socket描述符的选项**SO_REUSEADDR**为**1**，表示允许创建端口号相同但IP地址不同的多个socket描述符。
- 在server代码的**socket()**和**bind()**调用之间插入如下代码：
- int opt = 1;
- **setsockopt**(listenfd, SOL_SOCKET, **SO_REUSEADDR**, &opt, sizeof(opt));

select

- 定义函数 int select(int n,fd_set * readfds,fd_set * writefds,fd_set * exceptfds,struct timeval * timeout);

  函数说明 select()用来等待文件描述符状态的改变。

  参数n代表**最大的文件描述符**加1，参数readfds、writefds 和exceptfds 称为描述符组，是用来回传该描述符的读，写或例外的状况。

select操作

- select底下的宏提供了处理这三种描述词组的方式:
  FD_CLR(int fd,fd_set* set)；用来清除描述词组set中相关fd 的位
  FD_ISSET(int fd,fd_set *set)；用来测试描述词组set中相关fd 的位是否为真
  FD_SET（int fd,fd_set*set）；用来设置描述词组set中相关fd的位
  FD_ZERO（fd_set *set）；用来清除描述词组set的全部位

  参数 timeout为结构timeval，用来设置select()的等待时间，其结构定义如下

```
struct timeval
{
    time_t tv_sec;
    time_t tv_usec;
};
```

  如果参数timeout设为NULL则表示select（）没有timeout。

返回值

执行成功则返回文件描述符状态已改变的个数

如果返回0代表在描述符状态改变前已超过timeout时间

当有错误发生时则返回-1，错误原因存于errno，此时参数readfds，writefds，exceptfds和timeout的值变成不可预测。

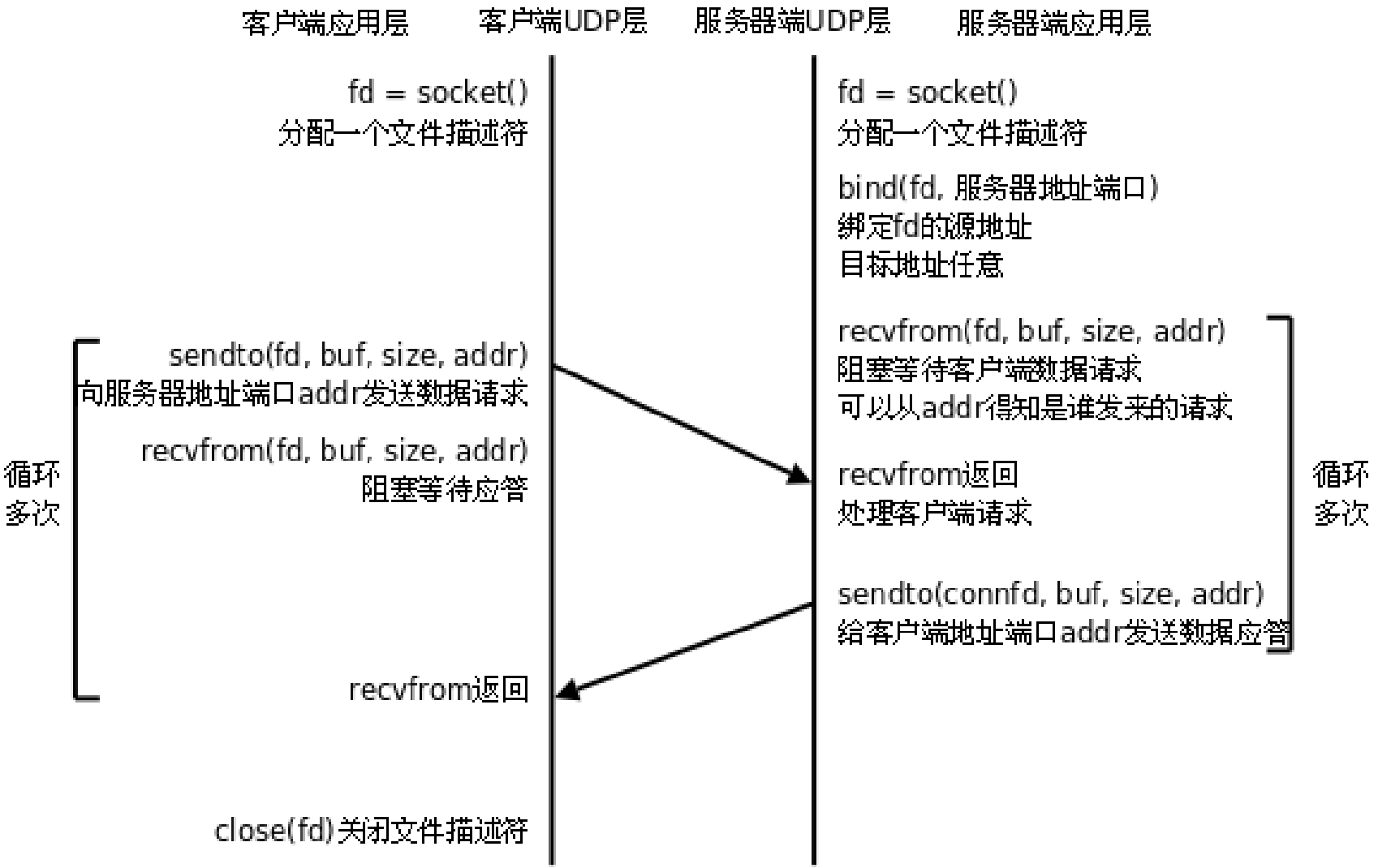
EBADF 文件描述词为无效的或该文件已关闭
EINTR 此调用被信号所中断
EINVAL 参数n 为负值。
ENOMEM 核心内存不足

使用select

- select是网络程序中很常用的一个系统调用，它可以同时监听多个阻塞的文件描述符（例如多个网络连接），哪个有数据到达就处理哪个，这样，不需要fork和多进程就可以实现并发服务的server.

基于UDP协议的网络程序

UDP通讯流程



UDP例子

- 见server.c
- 和client.c

UNIX DOMAIN SOCKET

- 虽然网络socket也可用于同一台主机的进程间通讯（通过loopback地址127.0.0.1），但是UNIX Domain Socket用于IPC更有效率：

  不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。

  这是因为，**IPC机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。**UNIX Domain Socket也提供面向流和面向数据包两种API接口，类似于TCP和UDP，**但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。**

- UNIX Domain Socket是**全双工的**，API接口语义丰富，相比其它IPC机制有明显的优越性，目前已成为使用最广泛的IPC机制，比如X Window服务器和GUI程序之间就是通过UNIX Domain Socket通讯的。

- 使用UNIX Domain Socket的过程和网络socket十分相似，也要先调用socket()创建一个socket文件描述符，address family指定为**AF_UNIX**，type可以选择SOCK_DGRAM或SOCK_STREAM，protocol参数仍然指定为0即可。

- UNIX Domain Socket与网络socket编程**最明显的不同在于地址格式不同，用结构体sockaddr_un表示，网络编程的socket地址是IP地址加端口号，而UNIX Domain Socket的地址是一个socket类型的文件在文件系统中的路径**，这个socket文件由bind()调用创建，如果调用bind()时该文件已存在，则bind()错误返回。

服务器模型

服务器模型

- 在网络程序里面,一般的来说都是许多客户机对应一个服务器。
- 为了处理客户机的请求, 对服务端的程序就提出了特殊的要求。

- 目前最常用的服务器模型.
  - 循环服务器:

    循环服务器在同一个时刻只可以响应一个客户端的请求
  - 并发服务器:

    并发服务器在同一个时刻可以响应多个客户端的请求

服务器模型

- **循环服务器：UDP服务器**
  - UDP循环服务器的实现非常简单:
    - UDP服务器每次从套接字上读取一个客户端的请求,处理, 然后将结果返回给客户机.
  - 可以用下面的算法来实现(伪代码):

```
socket(...);
bind(...);
while(1)
{
    recvfrom(...);
    process(...);
    sendto(...);
}
```

服务器模型

- 因为UDP是非面向连接的,没有一个客户端可以老是占住服务端.
- 只要处理过程不是死循环, 服务器对于每一个客户机的请求总是能够满足

服务器模型

- **循环服务器:TCP服务器**
  - TCP循环服务器的实现也不难:

    TCP服务器接受一个客户端的连接,然后处理,完成了这个客户的所有请求后,断开连接.
  - TCP循环服务器一次只能处理一个客户端的请求.
  - 只有在这个客户的所有请求都满足后, 服务器才可以继续后面的请求.
  - 这样如果有一个客户端占住服务器不放时,其它的客户机都不能工作了.因此,TCP服务器一般很少用循环服务器模型的.

服务器模型

- 算法如下:

```
socket(...);
bind(...);
listen(...);
while(1)
{
    accept(...);
    while(1)
    {
        read(...);
        process(...);
        write(...);
    }
    close(...);
}
```

服务器模型

- **并发服务器:TCP服务器**
  - 为了弥补循环TCP服务器的缺陷,人们又想出了并发服务器的模型.

    并发服务器的思想是每一个客户机的请求并不由服务器直接处理,而是服务器创建一个子进程来处理.
  - TCP并发服务器可以解决TCP循环服务器客户机独占服务器的情况.
  - 为了响应客户机的请求,服务器要创建子进程来处理. 而创建子进程是一种非常消耗资源的操作.

服务器模型

- 算法如下:

```
socket(...);
bind(...);
listen(...);
 while(1)
{
     accept(...);
     if(fork(..)==0)
     {
          while(1)
          { read(...); process(...); write(...); }
          close(...);
          exit(...);
      }
      close(...);
}
```

服务器模型

- **并发服务器:UDP服务器**
  - 人们把并发的概念用于UDP就得到了并发UDP服务器模型.
  - 并发UDP服务器模型其实是简单的.和并发的TCP服务器模型一样是创建一个子进程来处理的 算法和并发的TCP模型一样.
  - 除非服务器在处理客户端的请求所用的时间比较长以外,人们实际上很少用这种模型.

服务器模型

- **I/O多路复用并发服务器**

```
初始化(socket,bind,listen);
 while(1)
{
    设置监听读写文件描述符(FD_*);
    调用select;
    如果是倾听套接字就绪,说明一个新的连接请求建立
    {
        建立连接(accept);
        加入到监听文件描述符中去;
     }
    否则说明是一个已经连接过的描述符
    {
        进行操作(read或者write);
    }
}
```

服务器模型

- 多路复用I/O可以解决资源限制的问题.此模型实际上是将UDP循环模型用在了TCP上面.
- 这也就带来了一些问题.如由于服务器依次处理客户的请求,所以可能会导致有的客户会等待很久.

参考书目

- 《UNIX网络编程（卷1）》

专业铸就品质 梦想成就未来

The Specialty Casts Quality,the Dream Gains Future.