Socket programming

杨劲松 yjs@oldhand.org

2012.03.24

参考资料

- W.Richard Stevens 《UNIX网络编程》(第1卷)
- W.Richard Stevens《TCP/IP详解》(第1卷)
- W.Richard stevens《UNIX环境高级编程》
- Eric S.Raymond 《UNIX编程艺术》

学习方法

- 参考W. Richard Stevens《UNIX网络编程》(第1卷)
- 参考W. Richard Stevens《UNIX环境高级编程》
- 动手实践
  - 编写测试程序
  - 使用tcpdump/wireshark等网络调试工具进行验证

主要内容

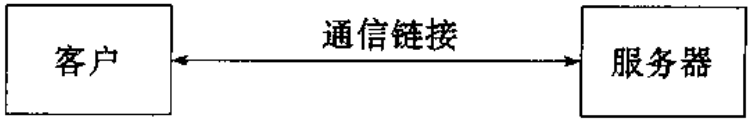
- 概述
- 预备知识
- 系统调用
- 系统I/O模型
- 调度模型
- TCP编程、多进程并发编程、多线程并发编程
- UDP编程、广播与多播编程
- UNIX domain编程
- 嵌入式环境下网络编程
- 实现简单的WEB服务器

1. 概述-主要内容

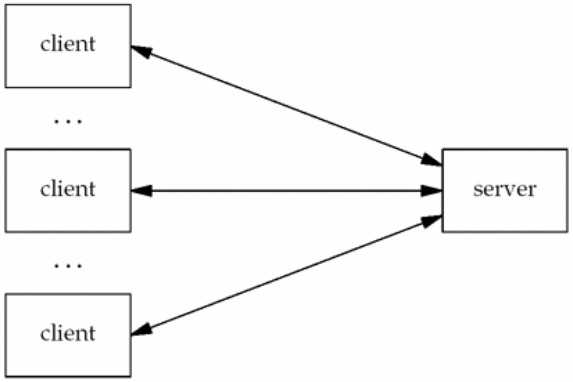
- 客户端与服务器
- Socket

1.1 概述-什么是客户端与服务器

- 大多数网络应用系统可分为两部分
  - 客户端(Client)
  - 服务器(Server)

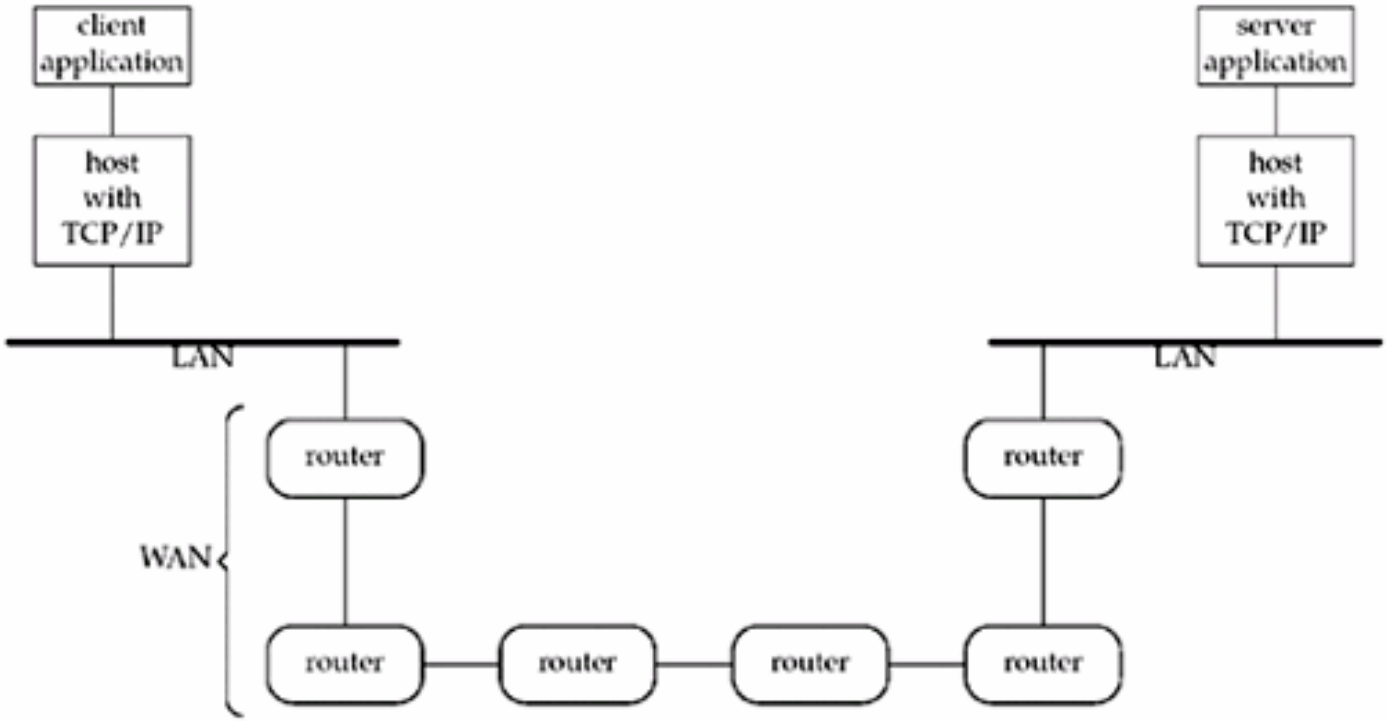


- 典型的Client/Server结构的服务
  - WEB
  - FTP
  - SMTP/POP3
  - SSH
  - ……



- 有些情况下，Client/Server代表的意义是主动/被动
  - 客户端-主动发起方
  - 服务器-被动接受方
- 有些情况下，无法区分Client/Server
  - P2P，互为Client，互为Server
  - 分布式系统

1.1 概述-什么是客户端与服务器



处于不同局域网的客户和服务器主机通过广域网连接

1.2.1 概述-Socket简介

- 1982 - Berkeley Software Distributions 操作系统引入了 sockets 作为本地进程之间通信的接口
- 1986 - Berkeley 扩展了socket 接口使之支持UNIX 下的 TCP/IP 通信
- 现在很多应用 (FTP, Telnet, etc) 都依赖这一接口

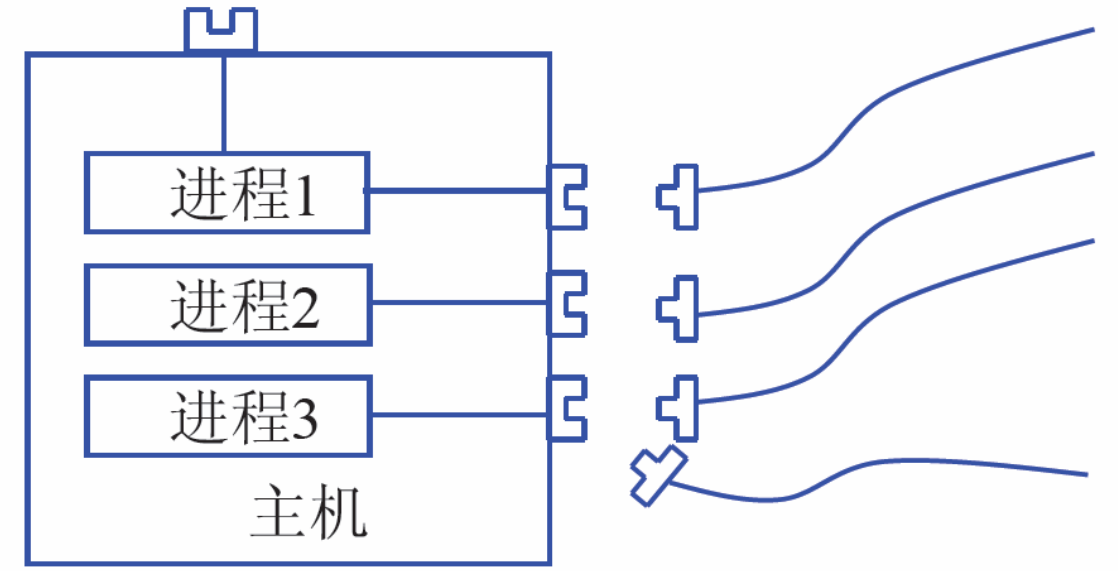
- X/Open Transport Interface (XTI)是另外一种网络编程接口

1.2.2 Socket是什么？

- 是编程的接口，是一系列的函数调用
- 是使用socket进行进程间通信的通信端点(endpoint)
- 是一种特殊的文件描述符 (everything in Unix is a file)
- 一对socket构成了交流数据的一个通道
- 并不仅限于TCP/IP
- 通信协议
  - 面向连接 (Transmission Control Protocol - TCP/IP)
  - 无连接 (User Datagram Protocol -UDP 和 Inter-network Packet Exchange - IPX)
- 一个进程可以有多个socket

1.2.2 Socket是什么？

- 一个主机可能有多个IP地址IP地址和端口合起来构成协议地址



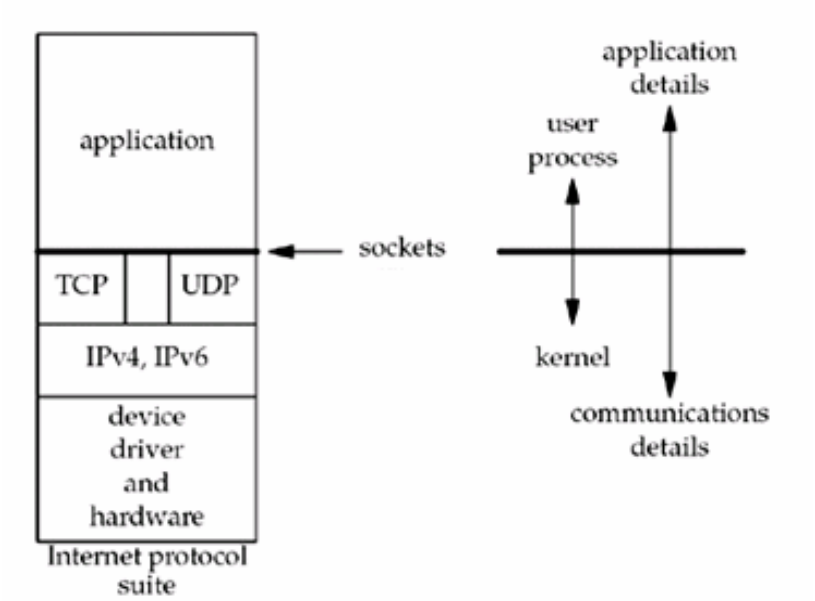
- 每个连接，对应两个IP地址，两个端口号，称为“四元组”
  - 在协议族确定的前提下，“四元组”才有意义，一般我们说“四元组”都默认是IPv4协议族
- 每个socket fd代表一个连接，其实质是个一个序号

1.2.3 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

1.2.4 什么是Socket

- 独立于具体协议的网络编程接口
- 在ISO模型中，主要位于会话层和传输层之间
- BSD Socket（伯克利套接字）是通过标准的UNIX文件描述符和其它程序通讯的一个方法，目前已经被广泛移植到各个平台。

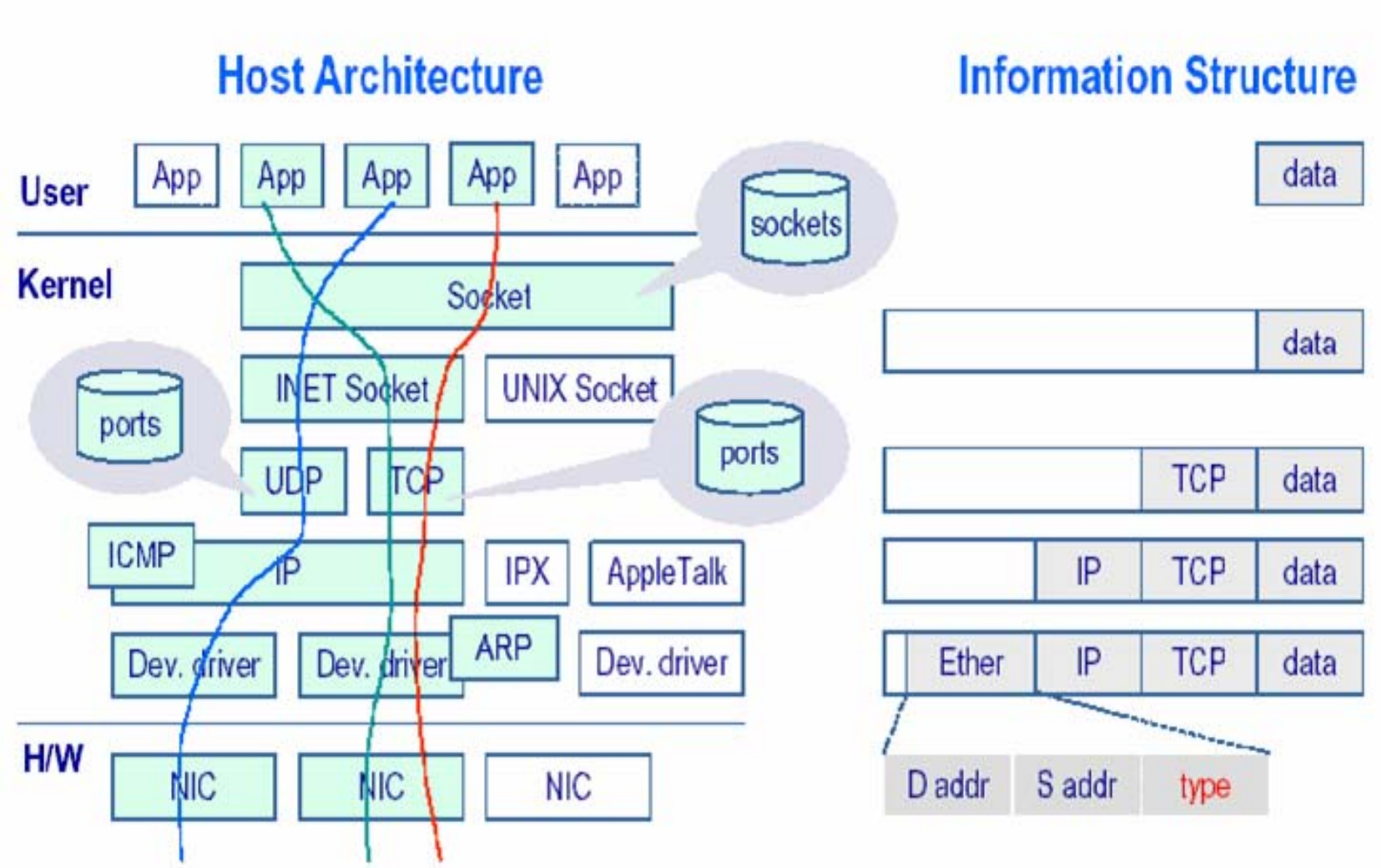


Socket所处的位置

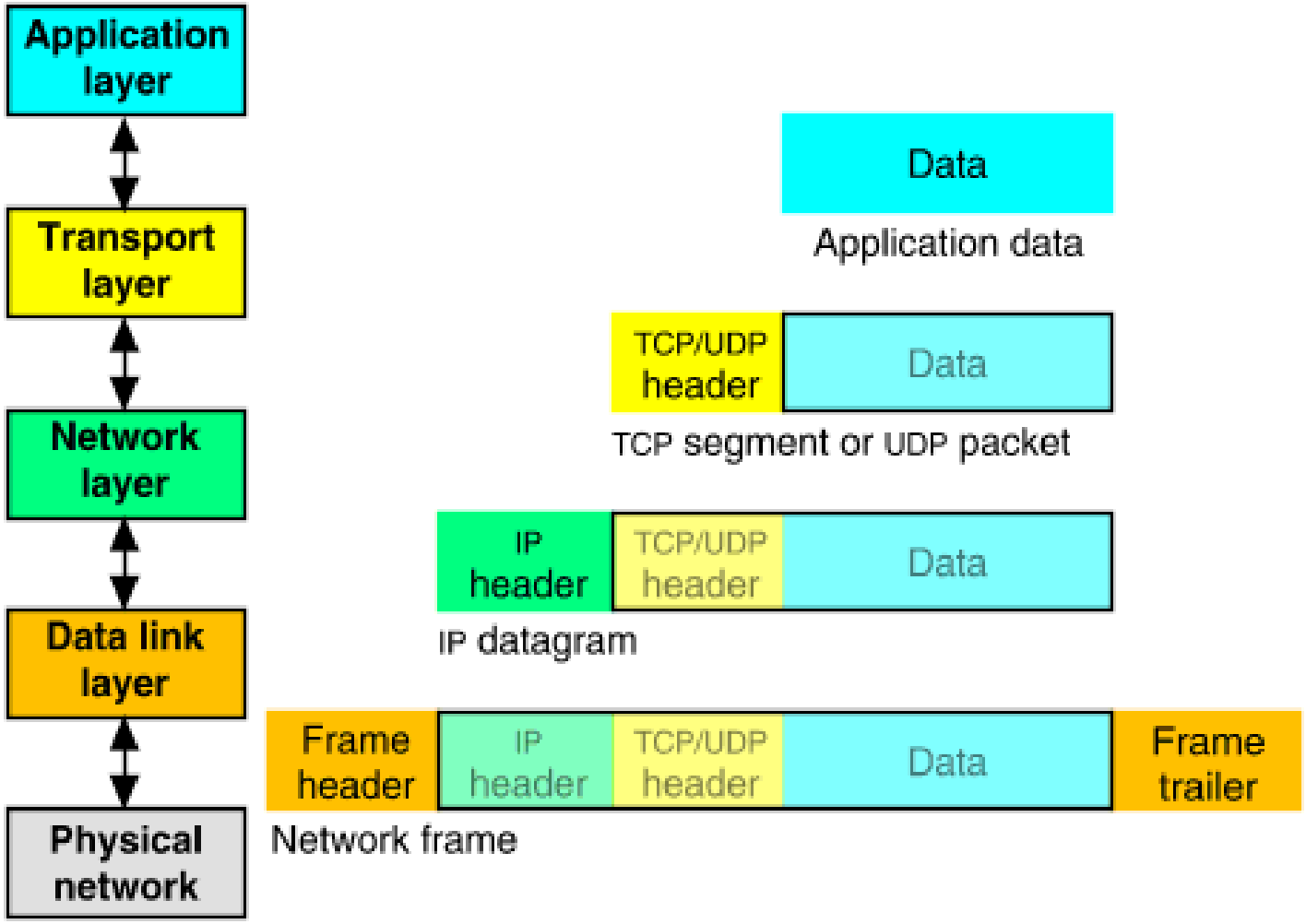
1.2.5 Socket类型

- 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。

The gory details …..



A day in the life of Network Packet



2. 预备知识-主要内容

- IP地址
- 端口号
- 字节序
- 地址结构
- 数据类型
- 协议族与地址族
- 寻址

2.1 IP地址

- IP地址是Internet中主机的标识
  - Internet中的主机要与别的机器通信必须具有一个IP地址
  - 一个IP地址为32位（IPv4），或者128位（IPv6）
  - 每个数据包都必须携带目的IP地址和源IP地址，路由器依靠此信息为数据包选择路由
  - 特殊的IP地址：广播地址、多播地址
- 表示形式：常用点分形式，如202.38.64.10，最后都会转换为一个32位的整数。
- IP地址分级
- 子网掩码

2.1.1 地址表示方法

- IPv4，采用十进制点分法表示
  - 192.168.1.108
  - 有些时候，也采用十六进制表示，这种方法不常见
  - 在程序里，使用二进制表示
- IPv6，采用十六进制，以冒号分割的方式，也有时候采用点分割，点分割的方式不常见
  - fe80::20d:60ff:fecb:86e5
  - fe80:0000:0000:0000:020d:60ff:fecb:86e5
  - 有两种表示方式，长方式和短方式
    - 长方式，为0的位置也要写出来
    - 短方式，连续为0的位置，通过两个连续的冒号表示

2.1.2 地址的二进制格式

- IPv4地址结构

```
/* Internet(IPv4) address, defined in /usr/include/netinet/in.h */
typedef uint32_t in_addr_t;

struct in_addr
{
  in_addr_t s_addr;
};
```

- IPv6地址结构

```
/* IPv6 address, defined in /usr/include/netinet/in.h */
struct in6_addr
{
  union
  {
    uint8_t u6_addr8[16];
    uint16_t u6_addr16[8];
    uint32_t u6_addr32[4];
  } in6_u;
#define s6_addr                 in6_u.u6_addr8
#define s6_addr16               in6_u.u6_addr16
#define s6_addr32               in6_u.u6_addr32
};
```

2.1.3 IP地址转换函数

- 只能处理IPv4地址的函数
  - inet_aton()
  - inet_ntoa()
  - inet_addr()
    - 这个函数有问题，不能正确处理255.255.255.255
  - ……
- 既可以转IPv4，也可以转IPv6的函数
  - inet_ntop()
  - inet_pton()
- 推荐使用inet_pton()/inet_ntop()

2.1.3.1 inet_aton/inet_addr/inet_ntoa

- 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);
```

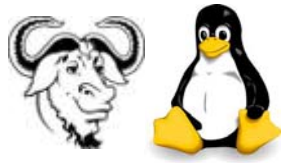
- inet_addr()
  - 功能同上，返回地址。

```
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);
```

2.1.3.1 IP地址转换函数使用



- **unsigned long inet_addr(char *address)**
  - address是以NULL结尾的点分IPv4字符串。该函数返回32位的地址，如果cp字符串包含的不是合法的IP地址，则函数返回。例如：

```
in_addr addr;
addr.s_addr = inet_addr("202.117.50.26");
```

- **char* inet_ntoa(struct in_addr address)**
  - address是IPv4地址结构，函数返回一指向包含点分IP地址的静态存储区字符指针，如果错误则函数返回NULL

```
fprintf(stdout, “Incoming connection from %s:%d\n”,
  inet_ntoa(addr.sin_addr), ntohs(addr.sin_port));
```

2.1.3.1 example code

```
  fprintf(stdout, "------- Convert address from x.x.x.x to binary, using inet_aton() ------\n");

  /* inet_aton() converts the Internet host address cp from the  standard
   * numbers-and-dots  notation  into  binary  data and stores it in the
   * structure that inp points to. inet_aton() returns non-zero if the
   * address is  valid, zero if not.
   */

  struct in_addr addr;

#if 0
  struct in_addr
  {
    in_addr_t s_addr;
  };
#endif

  //int inet_aton(const char *cp, struct in_addr *inp);
  if (inet_aton(argv[1], &addr) != 0)
  {
    // inet_aton() returns non-zero if the address is  valid, zero if not.
    fprintf(stdout, "Valid address: 0x%08x\n", addr.s_addr);
  }
  else
  {
    fprintf(stdout, "Invalid address.\n");
  }
```

2.1.3.1 example code

```c
struct in_addr addr;

addr.s_addr = 0x0900a8c0;

/* The inet_ntoa() function converts the Internet host address  in  given  in
 * network byte order to a string in standard numbers-and-dots notation.  The
 * string is returned in a  statically  allocated  buffer,  which  subsequent
 * calls will overwrite.
 */

//char *inet_ntoa(struct in_addr in);
fprintf(stdout, "Converted by inet_ntoa(), result: %s\n", inet_ntoa(addr));

char ipv4_address[] = "xxx.xxx.xxx.xxx\0";

memset(ipv4_address, 0, sizeof(ipv4_address));

//const char *inet_ntop(int af, const void *src, char *dst, socklen_t cnt);
inet_ntop(AF_INET, (void *) &addr, ipv4_address, sizeof(ipv4_address));

fprintf(stdout, "Converted by inet_ntop(), result: %s\n", ipv4_address);
```

2.1.3.2 inet_pton()/inet_ntop()

- inet_pton()可以将字符串表示的IPv4/IPv6地址转换为二进制

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

int inet_pton(int af, const char *src, void *dst);
```

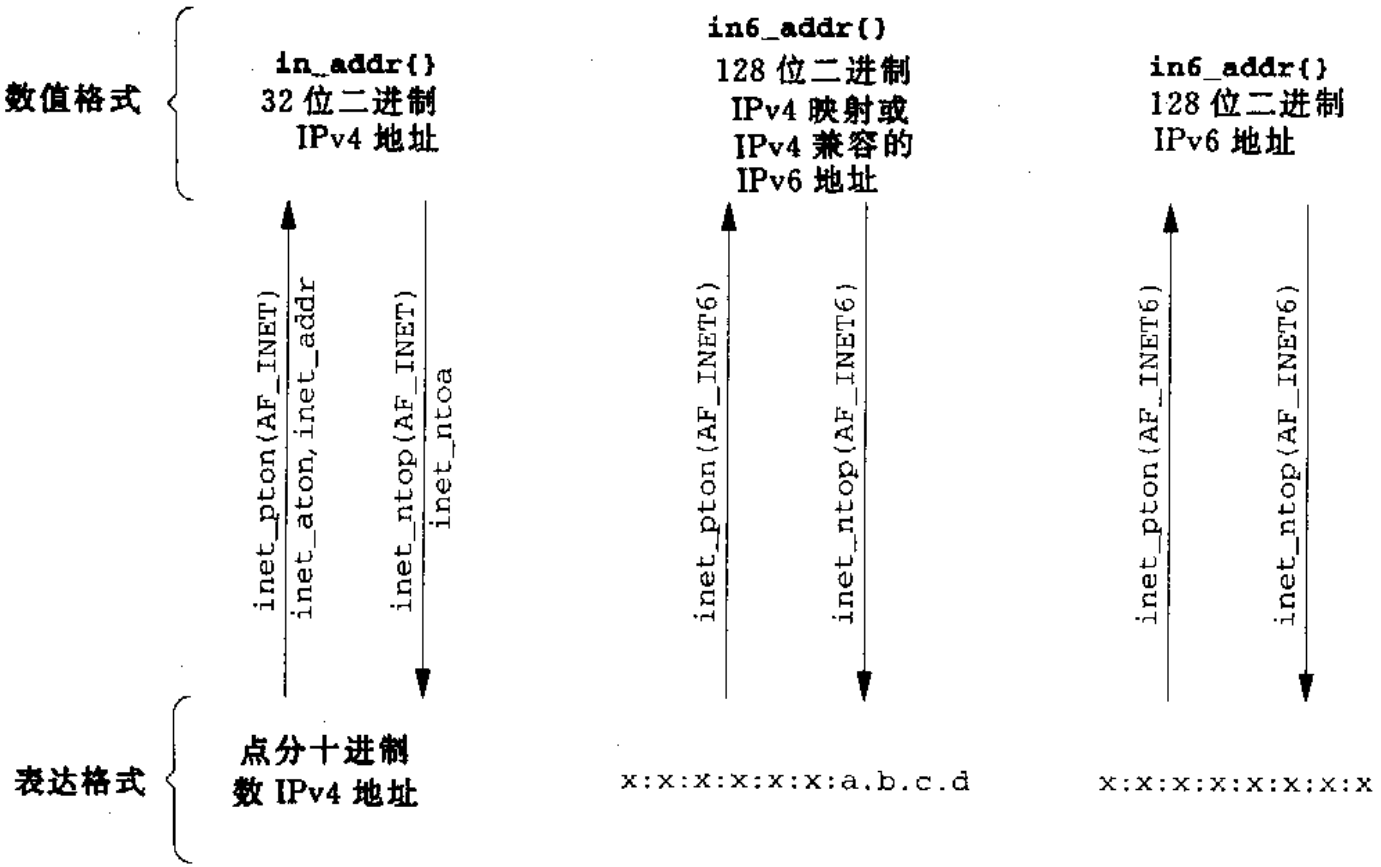
- inet_ntop()可以将二进制表示的IPv4/IPv6地址转换为字符串

```
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>

const char *inet_ntop(int af, const void *src, char *dst, socklen_t cnt);
```

2.1.3.2 example code

```
fprintf(stdout, "------- Convert address from x.x.x.x to binary, using inet_pton() ------\n");

/*
 * This function converts the character string src into a network address structure in the af address family,
 * then copies the network address structure to dst.
 *
 * inet_pton(3) extends the inet_addr(3) function to support multiple address families, inet_addr(3) is now
 * considered to be deprecated in favor of inet_pton(3).
 *
 * The following address families are currently supported:
 *
 * AF_INET
 *
 * src points to a character string containing an IPv4 network address in the dotted-quad format, "ddd.ddd.ddd.ddd".
 * The address is converted to a struct in_addr and copied to dst, which must be sizeof(struct in_addr) bytes long.
 *
 * AF_INET6
 *
 * src points to a character string containing an IPv6 network address in any allowed IPv6 address format.
 * The address is converted to a struct in6_addr and copied to dst, which must be sizeof(struct in6_addr) bytes long.
 */

int result;

//int inet_pton(int af, const char *src, void *dst);
if ((result = inet_pton(AF_INET, argv[1], (void *) &addr)) < 0)
{
  // returns a negative value and sets errno to EAFNOSUPPORT if af does not contain a valid address family.
  fprintf(stderr, "Convert failed: %s\n", strerror(errno));
}
else if (result == 0)
{
  // 0 is returned if src does not contain a character string representing a valid network address in the specified address family.
  fprintf(stderr, "Convert failed: Provided address does not contain a valid address.\n");
}
else
{
  //A positive value is returned if the network address was successfully converted.
  fprintf(stdout, "Result: 0x%08x\n", addr.s_addr);
}
```

2.1.4 地址转换函数对照

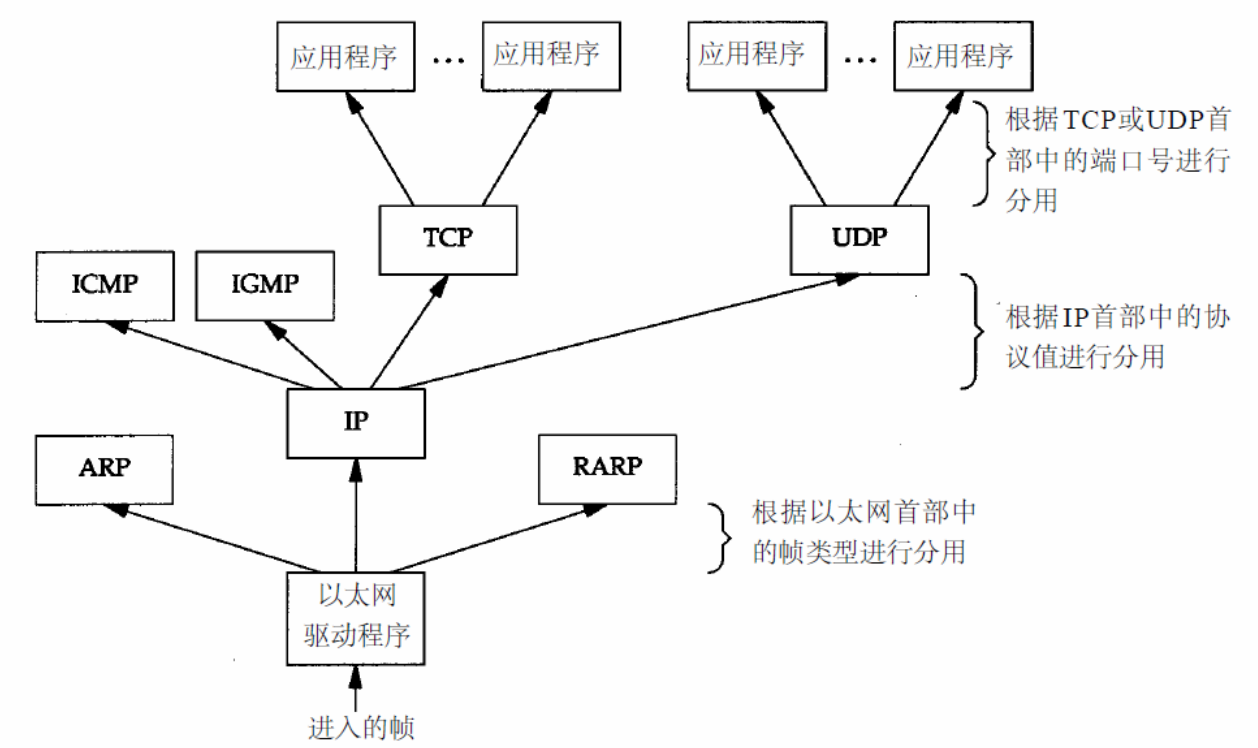


2.2 端口号

- 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
  - 1024以下的端口为特权端口，只能由root用户(euid=0的用户)打开
  - 注册端口：1024~49151
  - 动态或私有端口：49151~65535
- 周知端口，由/etc/services文件定义，可以通过如下函数访问：
  - getservbyname()
  - getservbyport()
- 一般做系统级的网络服务程序（如WEB Server、FTP Server等），接受/etc/services文件的配置是一个好习惯
  - 系统管理员可以通过修改/etc/services文件来配置端口号等

2.2.1 端口号的作用

- 当目的主机收到一个以太网数据帧时，数据就开始从协议栈中由底向上升，同时去掉各层协议加上的报文首部。每层协议盒都要去检查报文首部中的协议标识，以确定接收数据的上层协议。这个过程称作分用（ Demultiplexing），下图显示了该过程是如何发生的。



2.2.2 一个比喻

- 如果把网络数据包的投递过程看成是给远方的一位朋友寄一封信，那么：
  - IP地址就是这位朋友的所在位置，如上海交大XX系，邮局依靠此信息进行信件的投递，网络数据则依靠IP地址信息进行路由
  - 端口号就是这位朋友的名字，传达室依靠这个信息最终把这封信交付给这位收信者，数据包则依靠端口号送达给接收进程
- 也就是说
  - IP地址是做host-host投递的
  - 端口号是做主机内部投递的

2.2.3 端口号使用规则

- 尽可能的避免使用周知端口
  - 如果使用了周知端口，可能会造成一些问题，比如blind attack
  - 避免blink attack，可以在协议中设置一个magic value
- 如果需要使用多个端口号，最好在一个连续的区间使用
  - 该区间够用即可，不要设置过大
  - 方便系统的部署，因为很多时候，网络的防火墙是要限制端口号的，如果端口号随意使用，防火墙需要全部开放端口，不安全，如果端口号在某个段内使用，防火墙可以只打开该段。

2.3 字节序

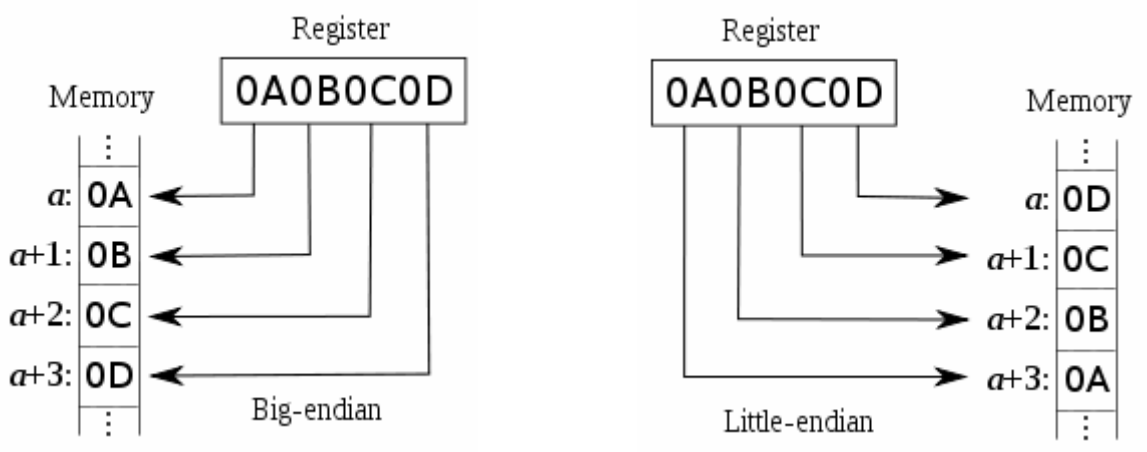
- 不同类型CPU的主机中，内存存储multi-bytes整数序列有两种方法，称为主机字节序(HBO)：
  - 小端序（little-endian） - 低序字节存储在低地址
    - 将低字节存储在起始地址，称为“Little-Endian”字节序，Intel、AMD等采用的是这种方式；
  - 大端序（big-endian）- 高序字节存储在低地址
    - 将高字节存储在起始地址，称为“Big-Endian”字节序，由Macintosh、Motorola等所采用
- 多字节应该理解为多字节表意
  - int通过4个字节表意，所以需要转换
  - 字符串虽然也是多字节，但是是单字节表意，所以不需要转换
- 网络中传输的数据必须按网络字节序
  - 网络字节序就是大端字节序

2.3 字节序转换的必要性

- 凡是传给网络的参数，都需要做字节序转换，将主机字节序转为网络字节序
  - 端口号，要使用htons()进行转换
  - IP地址，也要使用网络字节序的二进制方式，可以使用inet_pton()转换
- 凡是从网络取得的参数，都需要做字节序转换，将网络字节序转为主机字节序
  - 端口号，要使用ntohs()进行转换
  - IP地址，可以使用inet_ntop()转换
- 用户传输的数据，是否要做转换，由用户协议来约定
  - 建议采用网络字节序，这样有更好的扩展能力
  - 即便在系统设计时，参与通讯的主机字节序相同，但是考虑到未来扩展的需要，最好也要统一使用网络字节序

2.3.1 字节序

- 大端(Big-Endian):字节的高位在内存中放在存储单元的起始位置
- 小端(Little-Endian):与大端相反



2.3.2 字节序检测

```c
#include <stdio.h>

int main(int argc, char **argv)
{
  union
  {
    int x;
    char c[4];
  } value;

  value.x = 0x0a0b0c0d;

#if __BYTE_ORDER == __LITTLE_ENDIAN
  fprintf(stdout, "Little endian: ");
#elif __BYTE_ORDER == __BIG_ENDIAN
  fprintf(stdout, "Big endian: ");
#else
#  error        "Please fix <bits/endian.h>"
#endif

  int i;

  for (i = 0; i < 4; i++)
  {
    fprintf(stdout, "0x%02x ", value.c[i]);
  }

  fprintf(stdout, "\n");

  return 0;
}
```

运行结果

- 小端字节序主机(x86)的测试结果

```
[yjs@boss online-class]$ uname -a
Linux boss.oldhand.org 2.6.18-194.el5 #1 SMP Tue Mar 16 21:52:43 EDT 2010 i686 i686 i386 GNU/Linux
[yjs@boss online-class]$ ./byteorder
c[0] = 0x0d c[1] = 0x0c c[2] = 0x0b c[3] = 0x0a
```

- 大端字节序主机(PowerPC)的测试结果

```
[yangjingsong@aix ~]$ uname -a
AIX aix 3 5 00C97AC04C00 powerpc unknown AIX
[yangjingsong@aix ~]$ ./byteorder
c[0] = 0x0a c[1] = 0x0b c[2] = 0x0c c[3] = 0x0d
[yangjingsong@aix ~]$
```

2.3.3 网络字节序和主机字节序

- 网络字节序(NBO-Network Byte Order)
  - 使用统一的字节顺序，避免兼容性问题
- 主机字节序(HBO-Host Byte Order)
  - 不同的机器HBO是不一样的，这与CPU的设计有关
- 处理器体系
  - x86，MOS Technology 6502，Z80，VAX，PDP-11等处理器为Little endian。
  - Motorola 6800，Motorola 68000，PowerPC 970，System/370，SPARC（除V9外）等处理器为Big endian
  - ARM, PowerPC (除PowerPC 970外), DEC Alpha, SPARC V9, MIPS, PA-RISC and IA64的字节序是可配置的。

2.3.4 字节序转换

- 对于长度为8 bits的数据类型，不需要转换
- 对于长度是16 bits的数据类型，需要转换
  - htons()
  - ntohs()
- 对于长度是32 bits的数据类型，需要转换
  - htonl()
  - ntohl()
- 对于长度是64 bits的数据类型，需要转换
  - 系统一般不提供转换的函数，需要自己手动的转换，或者自己定义协议进行约定
  - 有些系统提供htonll()和ntohll()

2.3.4 字节序转换函数

- 把给定系统所采用的字节序称为主机字节序。为了避免不同类别主机之间在数据交换时由于对于字节序解释的不同而导致的差错，引入了网络字节序，即网络传输所采用的字节序。规定网络字节序使用“Big-Endian”方式。
- 有些系统提供了64bit类型的转换，但是Linux没有提供
- 主机到网络
  - `u_long htonl (u_long hostlong);`
  - `u_short htons (u_short short);`
- 网络到主机
  - `u_long ntohl (u_long hostlong);`
  - `u_short ntohs (u_short short);`

Datatype	ILP32 model	LP64 model
char	8	8
short	16	16
int	32	32
long	32	64
pointer	32	64

字节序的处理

- 在编写程序的时候，不需要程序员去关注CPU体系结构来判断字节序，系统提供的头文件都定义好了字节序相关的参数
  - BYTEORDER
  - __BYTE_ORDER，如果程序中需要检查字节序时，使用该宏定义
  - 如果针对不同字节序编码时，可以在程序内判断__BYTE_ORDER的定义：
    - __BIG_ENDIAN，表示大端字节序
    - __LITTLE_ENDIAN，表示小端字节序
- 不要随便拷贝头文件来使用

2.4 地址结构

- 通用地址结构，用于函数原型定义，相当于泛型参数

```
//------------------------------------------------------------------
// Generic address structure, used to define function prototype
// defined in /usr/include/bits/socket.h
//------------------------------------------------------------------
struct sockaddr
{
  sa_family_t sa_family;
  char sa_data[14];
};

//------------------------------------------------------------------
// function prototype, use struct sockaddr as parameter
//------------------------------------------------------------------
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
int accept(int sockfd, struct sockaddr *addr, socklen_t * addrlen);
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
ssize_t recvfrom(int s, void *buf, size_t len, int flags,
               struct sockaddr *from, socklen_t * fromlen);
```

2.4 地址结构

- IPv4地址结构，用户IPv4协议族通讯

```
//------------------------------------------------------------------
// IPv4 address structure, used for IPv4 communication
// defined in /usr/include/netinet/in.h
//------------------------------------------------------------------
struct sockaddr_in ipv4_address_structure;    // IPv4

struct sockaddr_in
{
  sa_family_t sin_family;
  in_port_t sin_port;
  struct in_addr sin_addr;
  unsigned char sin_zero[8];
};
```

2.4 地址结构

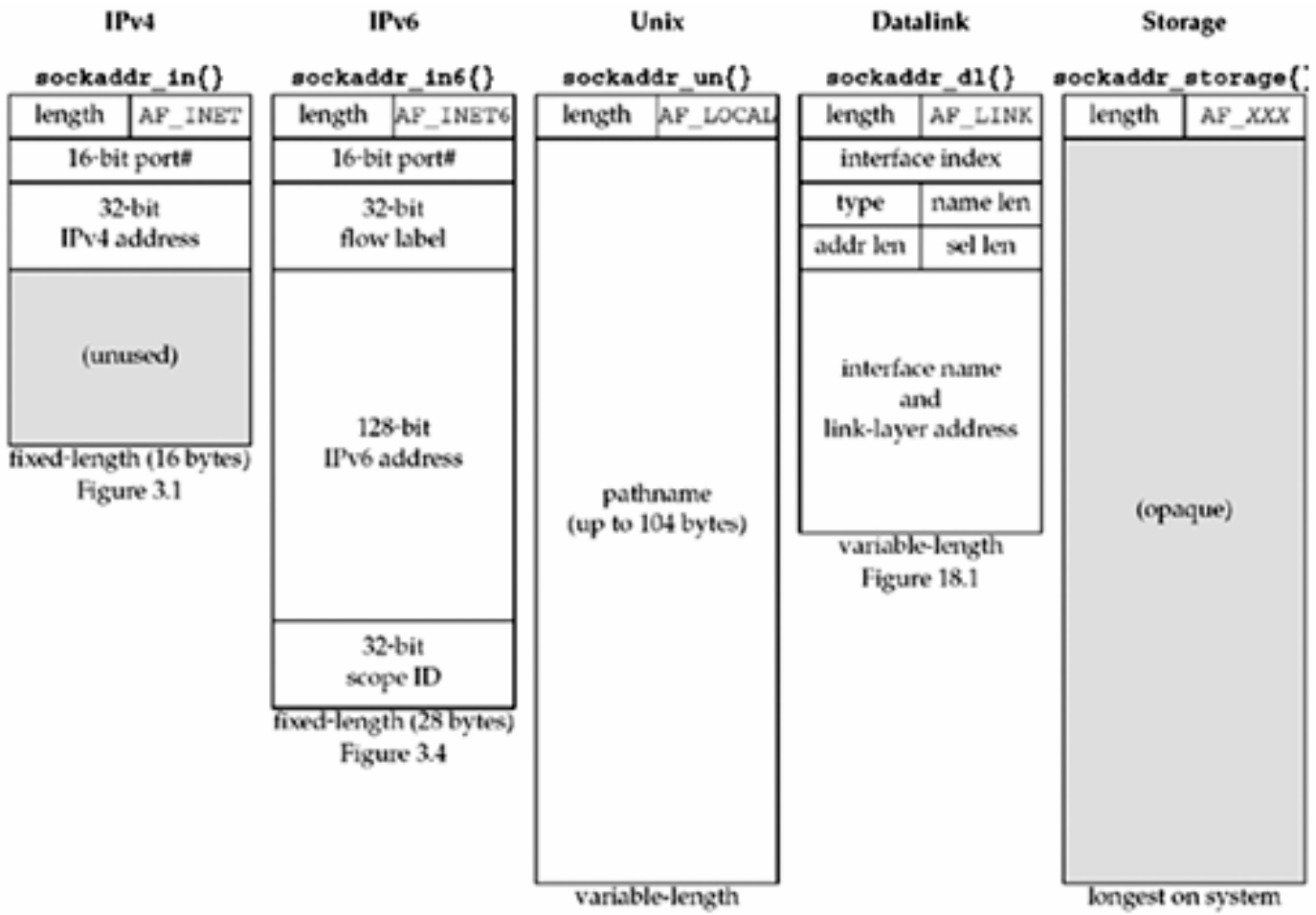
- IPv6地址结构

```
struct sockaddr_in6
{
  sa_family_t sin6_family;
  in_port_t sin6_port;
  uint32_t sin6_flowinfo;
  struct in6_addr sin6_addr;
  uint32_t sin6_scope_id;
}

struct in6_addr
{
  union
  {
    uint8_t u6_addr8[16];
    uint16_t u6_addr16[8];
    uint32_t u6_addr32[4];
  } in6_u;
}
```

- UNIX domain协议地址结构

```
struct sockaddr_un
{
  sa_family_t sun_family;
  char sun_path[108];
}
```

2.4.1 地址结构对照


IPv4
sockaddr_in{}
length
AF_INET
16-bit port#
32-bit
IPv4 address
(unused)
fixed-length (16 bytes)
Figure 3.1
IPv6
sockaddr_in6{}
length
AF_INET6
16-bit port#
32-bit
flow label
128-bit
IPv6 address
32-bit
scope ID
fixed-length (28 bytes)
Figure 3.4
Unix
sockaddr_un{}
length
AF_LOCAL
pathname
(up to 104 bytes)
variable-length
Datalink
sockaddr_dl{}
length
AF_LINK
interface index
type
name len
addr len
sel len
interface name
and
link-layer address
variable-length
Figure 18.1
Storage
sockaddr_storage{
length
AF_XXX
(opaque)
longest on system

2.4.2 地址结构处理

- 第一步，定义一个struct sockaddr_in的结构参数，并将它清零

```
// step 1, memset
memset(&ipv4_address_structure, 0, sizeof(ipv4_address_structure));
```

- 第二步，为这个结构赋值

```
// step 2, set fields
ipv4_address_structure.sin_family = AF_INET;  // IPv4
ipv4_address_structure.sin_port = htons(port);    // Network byte order
inet_pton(AF_INET, argv[1], &ipv4_address_structure.sin_addr);
```

- 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，作为参数传递给bind()/connect()等系统调用

```
// step 3, call functions, as parameter, cast to struct sockaddr
connect(fd, (struct sockaddr *) &ipv4_address_structure,
                sizeof(ipv4_address_structure));
```

2.4.2.1 example code – IPv4

```
  struct sockaddr_in ipv4_address;      // for IPv4 address family use

#if 0
  struct sockaddr_in
  {
    sa_family_t sin_family;
    in_port_t sin_port;
    struct in_addr sin_addr;
    unsigned char sin_zero[8];
  };
#endif

  /* Initialize */
  memset(&ipv4_address, 0, sizeof(ipv4_address));

  /* Set fields */
  ipv4_address.sin_family = PF_INET;    // IPv4 protocol family, you can use AF_INET instead
  ipv4_address.sin_port = htons(port_number);
  //ipv4_address.sin_addr.s_addr = inet_addr(ip_address_string);

  //int inet_aton(const char *cp, struct in_addr *inp);
  //inet_aton(ip_address_string, &ipv4_address.sin_addr);

  //int inet_pton(int af, const char *src, void *dst);
  inet_pton(AF_INET, ip_address_string, &ipv4_address.sin_addr);

  /* Use ipv4_address */
  //int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
  connect(fd, (struct sockaddr *) &ipv4_address, sizeof(ipv4_address));
```

2.4.2.2 example code – IPv6

```
  struct sockaddr_in6 ipv6_address;     // for IPv6 address family use

#if 0
  struct sockaddr_in6
  {
    sa_family_t sin6_family;
    in_port_t sin6_port;
    uint32_t sin6_flowinfo;
    struct in6_addr sin6_addr;
    uint32_t sin6_scope_id;
  };
#endif

  memset(&ipv6_address, 0, sizeof(ipv6_address));
  ipv6_address.sin6_family = PF_INET6;  // for IPv6 address family, you can use AF_INET6 instead
  //ipv6_address.sin6_port = htons(port_number);

  //int inet_pton(int af, const char *src, void *dst);
  //inet_pton(AF_INET6, ip_address_string, &ipv6_address.sin6_addr);

  // set other fields
  connect(fd, (struct sockaddr *) &ipv6_address, sizeof(ipv6_address));
```

2.4.2.3 example code – UNIX Domain

```
  //------------------------------------------------------------------
  // UNIX domain address structure
  //------------------------------------------------------------------
  struct sockaddr_un unixdomain_address;        // for unixdomain family use

#if 0
  struct sockaddr_un
  {
    sa_family_t sun_family;
    char sun_path[108];
  };
#endif

  memset(&unixdomain_address, 0, sizeof(unixdomain_address));
  unixdomain_address.sun_family = PF_UNIX;      // PF_LOCAL, AF_UNIX, AF_LOCAL

  //void *memcpy(void *dest, const void *src, size_t n);
  //memcpy(unixdomain_address.sun_path, file_system_path, strlen(file_system_length));
  connect(fd, (struct sockaddr *) &unixdomain_address, sizeof(unixdomain_address));
```

2.5 数据类型

- 在网络通讯中，最好使用stdint.h定义的整型数据类型

Specific integral type limits

Specifier	Signing	Bits	Bytes	Minimum Value	Maximum Value
int8_t	Signed	8	1	-2^7 which equals −128	$2^7 - 1$ which is equal to 127
uint8_t	Unsigned	8	1	0	$2^8 - 1$ which equals 255
int16_t	Signed	16	2	-2^{15} which equals −32,768	$2^{15} - 1$ which equals 32,767
uint16_t	Unsigned	16	2	0	$2^{16} - 1$ which equals 65,535
int32_t	Signed	32	4	-2^{31} which equals −2,147,483,648	$2^{31} - 1$ which equals 2,147,483,647
uint32_t	Unsigned	32	4	0	$2^{32} - 1$ which equals 4,294,967,295
int64_t	Signed	64	8	-2^{63} which equals −9,223,372,036,854,775,808	$2^{63} - 1$ which equals 9,223,372,036,854,775,807
uint64_t	Unsigned	64	8	0	$2^{64} - 1$ which equals 18,446,744,073,709,551,615

- 如果需要传输浮点型数据，需要特别处理
- 如果需要传输负数，需要特别处理
- 如果需要传输结构体，需要谨慎处理对齐和填充的问题

2.5 传输数据处理

- 整型数(short/long/long long)，采用字节序转换函数，但是64位类型没有提供转换函数
- float/double，如何处理？
  - 转换为整数，在发送时乘以系数，在接收时除以相应的系数,系数为10的整数倍。
- 负值如何处理？
  - 将符号位和值(转为绝对值)单独传输
- 在协议设计时，避免使用char/int/long类型，用stdint.h定义的类型
  - int8_t/uint8_t
  - int16_t/uint16_t
  - int32_t/uint32_t
  - int64_t/uint64_t
- 传输结构体，在协议设计时，主动进行对齐(推荐方式)，或者设置#progma pack(1)
  - 对齐的问题
  - 填充的问题
  - 位域(尽可能不用位域，如果必须用的话，对不同的字节序分别定义，类似IP头、TCP头的定义)

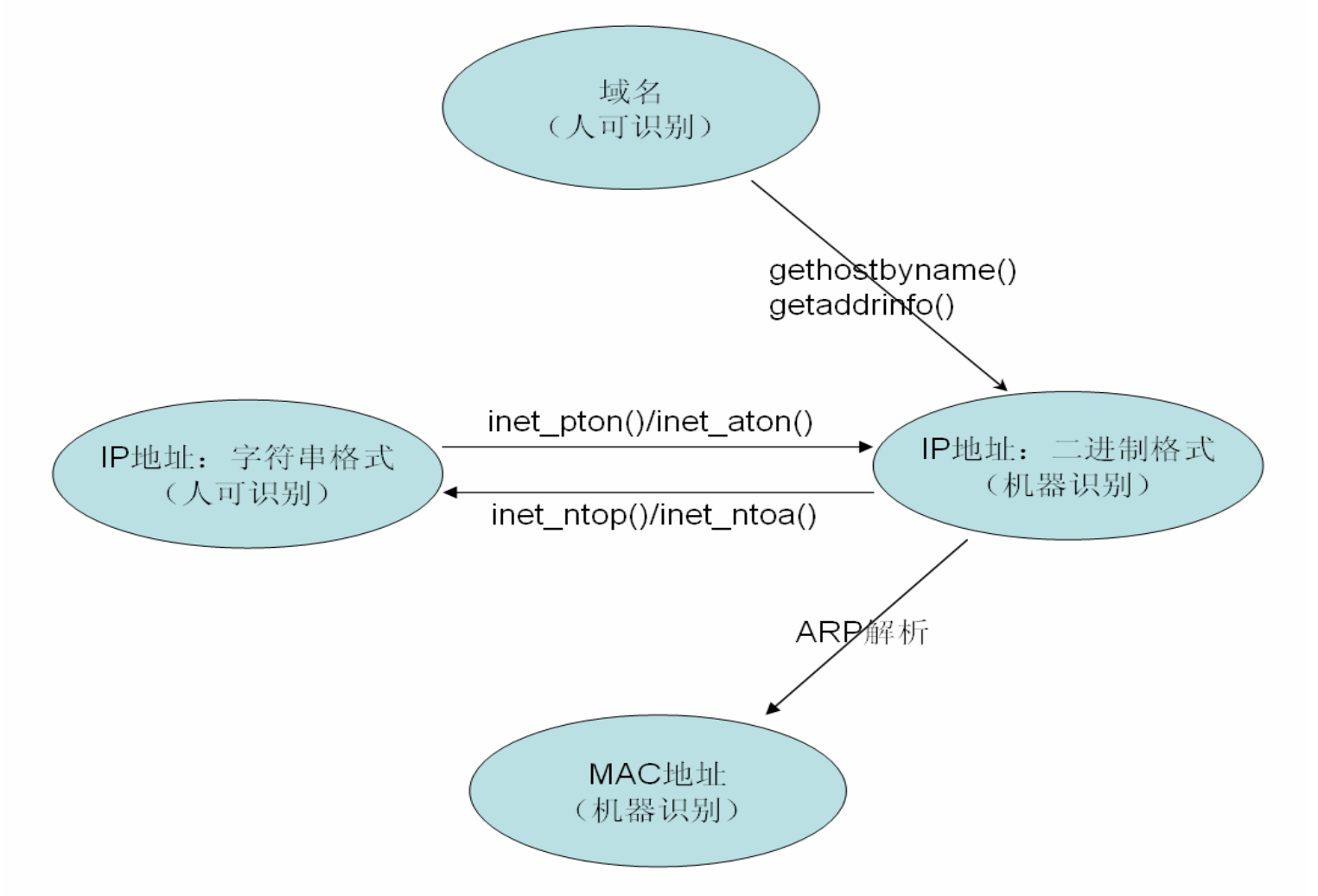
2.6 协议族与地址族

- Address family都是以“ AF_”前缀开头的
- Protocol family都是以“ PF_”前缀开头的
- 从设计上来讲，Protocol family支持多个address family，但是实现上将二者实现为一对一的，也就是说PF_XXX=AF_XXX
  - AF_INET(PF_INET)，对应的是IPv4
  - AF_INET6(PF_INET6)，对应的是IPv6
  - AF_UNIX/AF_LOCAL(PF_UNIX/PF_LOCAL)，对应的是UNIX domain
- 使用的一般规则：
  - 当传递给socket()函数的domain参数时用PF_XXX
  - 当传递给地址相关的处理函数时，用AF_XXX

2.6 协议族/地址族一致

- 在Socket编程中，以下代码中的地址参数应该是一致的：
  - 创建socket时候，调用socket()的domain参数
  - 调用connect()/bind()/accept()/sendto()/recvfrom()的地址参数
  - 地址参数中的family成员

2.7 寻址


域名
（人可识别）
gethostbyname()
getaddrinfo()
IP地址：字符串格式
（人可识别）
inet_pton()/inet_aton()
inet_ntop()/inet_ntoa()
IP地址：二进制格式
（机器识别）
ARP解析
MAC地址
（机器识别）

3. 系统调用-主要内容

- 网络连接相关的系统调用
- 网络信息检索相关的函数
- Socket options相关的系统调用
- 调度相关的系统调用

3.1 网络连接相关的系统调用

- socket()　创建套接字
- bind()　绑定本机端口
- connect()　建立连接
- listen()　监听端口
- accept()　接受连接
- recv(), read(), recvfrom()　数据接收
- send(), write(), sendto()　数据发送
- close(), shutdown()　关闭套接字

3.1.1 socket()

- socket()打开一个网络通讯端口

```
NAME
       socket - create an endpoint for communication

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       int socket(int domain, int type, int protocol);
```

- 返回值
  - 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，并可通过errno取得错误码
- 参数
  - 对于IPv4，domain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参数通常指定为0

3.1.1.1 domain参数

- The domain parameter specifies a communication domain
- This selects the protocol family which will be used for communication.
- These families are defined in <sys/socket.h>

- The currently understood formats include:

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

3.1.1.2 type参数

- The socket has the indicated type, which specifies the communication semantics.
- Currently defined types are:

SOCK_STREAM
Provides sequenced, reliable, two-way, connection-based byte streams. An out-of-band data transmission mechanism may be supported.

SOCK_DGRAM
Supports datagrams (connectionless, unreliable messages of a fixed maximum length).

SOCK_SEQPACKET
Provides a sequenced, reliable, two-way connection-based data transmission path for datagrams of fixed maximum length; a consumer is required to read an entire packet with each read system call.

SOCK_RAW
Provides raw network protocol access.

SOCK_RDM
Provides a reliable datagram layer that does not guarantee ordering.

SOCK_PACKET
Obsolete and should not be used in new programs; see **packet**(7).

Some socket types may not be implemented by all protocol families; for example, **SOCK_SEQPACKET** is not implemented for **AF_INET**.

3.1.1.3 protocol参数

- The protocol specifies a particular protocol to be used with the socket.
- Normally only a single protocol exists to support a particular socket type within a given protocol family, in which case protocol can be specified as 0.
- However, it is possible that many protocols may exist, in which case a particular protocol must be specified in this manner.
  - The protocol number to use is specific to the “communication domain” in which communication is to take place; see protocols(5).
  - See getprotoent(3) on how to map protocol name strings to protocol numbers.

3.1.1.4 example code

```
  // step 1, socket
  int listening_socket;

  //int socket(int domain, int type, int protocol);
  if ((listening_socket = socket(PF_INET, SOCK_STREAM, 0)) < 0)
  {
    fprintf(stderr, "Fatal error: socket() failed: %s\n", strerror(errno));
    // FIXME: how to do? retry?
    exit(1);
  }

#ifdef _DEBUG_
  fprintf(stdout, "Debug: Create a new TCP socket %d.\n", listening_socket);
#endif
```

3.1.2 bind()

- bind给socket分配一个IP地址和一个端口号，也称“命名”(name)

```
NAME
       bind - bind a name to a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

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

- 返回值
  - 成功返回0
  - 失败返回-1，并且设置errno
- 参数
  - sockfd – socket()调用获得的文件描述符
  - my_addr – 本地地址，地址族应与socket()调用所设定的domain相同，填充地址结构应该使用domain所对应的数据结构
  - addrlen – 本地地址长度，my_addr参数所采用的专用地址结构的长度

3.1.2.1 使用说明

- 如果客户端不指定（通常情况），系统根据目的地先选择要使用的接口，然后用该接口的地址作为源地址。
- 对于服务器，一般来说，设为INADDR_ANY，这就意味着由系统挑选一个地址，系统挑选的是发往客户端的接口的那个地址，而发往客户端使用哪个接口则是由客户所连接来时决定的。
- 客户端通常不需要bind()，服务器需要bind以表明要使用的IP地址与端口号

3.1.2.2 自动绑定

- 根据路由表选择自动绑定的IP地址
  - 选择一个可以到达目的主机的网络接口的IP地址
- 端口号
  - 从端口池中选择一个未用的端口号，系统可能有自己的规则，可能从某个端口段选择，不见得是最小的端口号
  - TCP与UDP的端口池彼此独立
- 客户端自动绑定，是因为绑定的地址仅仅用于一次通讯，绑定参数是什么不重要
- 服务器端调用bind()显式绑定一个地址和端口，是因为要使用明确的地址和端口号，这样客户端可以通过IP地址和端口号来连接服务器
- 客户端也可以调用bind()做显式绑定，但是通常没必要做
  - 在系统设计时需要考虑这个问题

3.1.2.3 example code

```c
struct sockaddr_in local_address;

memset(&local_address, 0, sizeof(local_address));

local_address.sin_family = PF_INET;
local_address.sin_addr.s_addr = htonl(INADDR_ANY);
Local_address.sin_port = htons(8080);

if (bind(fd, (struct sockaddr *) &local_address, sizeof(local_address)) < 0)
{
  fprintf(stderr, "bind() failed: %s\n", strerror(errno));
}
else
{
  fprintf(stdout, "bind() successfully.\n");
}
```

3.1.3 connect()

- 在一个socket上初始化一个连接

```
NAME
       connect - initiate a connection on a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

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

- 返回值
  - 成功返回0
  - 失败返回-1，并且设置errno
- 参数
  - sockfd – socket()调用获得的文件描述符
  - serv_addr – 远端地址
    - 地址族应与socket()调用所设定的domain相同，填充地址结构应该使用domain所对应的数据结构
    - 该结构中需要设定远端的IP地址和端口号
  - addrlen – 远端地址长度，serv_addr参数所采用的专用地址结构的长度

3.1.3.1 使用说明

- The file descriptor sockfd must refer to a socket.
- If the socket is of type SOCK_DGRAM then the serv_addr address is the address to which datagrams are sent by default, and the only address from which datagrams are received.
- If the socket is of type SOCK_STREAM or SOCK_SEQPACKET, this call attempts to make a connection to another socket. The other socket is specified by serv_addr, which is an address (of length addrlen) in the communications space of the socket. Each communications space interprets the serv_addr parameter in its own way.
- Generally, connection-based protocol sockets may successfully connect only once; connectionless protocol sockets may use connect multiple times to change their association. Connectionless sockets may dissolve the association by connecting to an address with the sa_family member of sockaddr set to AF_UNSPEC.
- connect()常用于TCP客户端，服务器一般不主动去连接
- connect完成和服务器的三次握手，建立连接
- 需要注意防火墙的影响

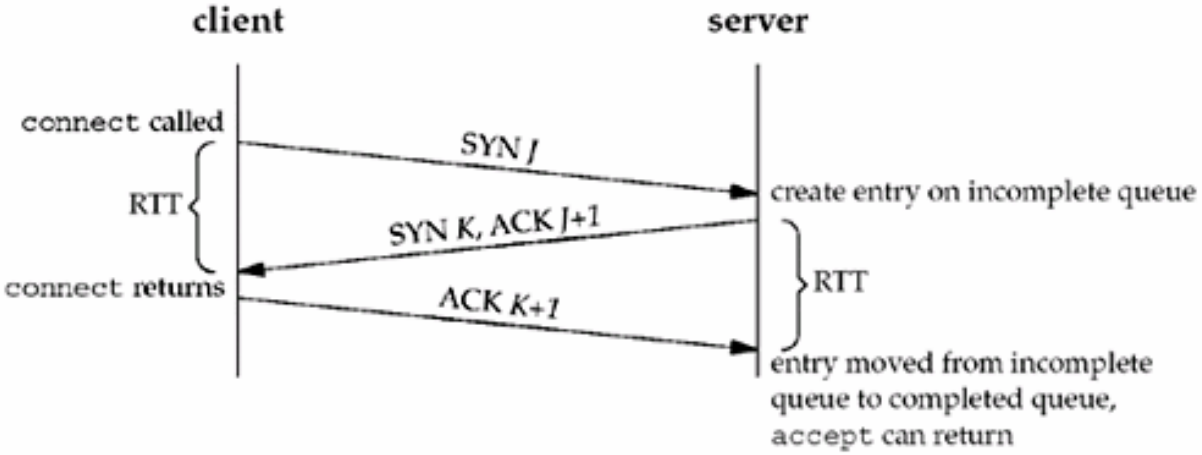
3.1.3.2 TCP socket调用connect

- TCP连接(面向连接的socket)可以成功调用connect()一次，如果连接成功之后被断开，需要关掉当前的socket，重新创建新的socket，再去进行connect()
- TCP的socket在调用connect()时，会发起三次握手，意味着connect()调用可能会阻塞。
- connect()阻塞的问题对于程序员来说，处理比较困难。

3.1.3.3 UDP socket调用connect()

- UDP可以做connect()，但是不是发起与远端主机的连接，只是通知本地的传输层
  - 以后默认发送数据报的地址是connect时的地址因此发送可以使用send()，而不用sendto()
  - 以后只从connect的地址接收数据报，从其他地址来的数据报不接收。
- UDP类型的socket可以调用connect多次，去改变默认的关联地址
- 如果不想跟任何地址关联，将地址结构中的sa_family参数设为AF_UNSPEC，然后调用connect()

3.1.3.4 TCP三次握手




```
04:05:00.886951 IP 172.16.0.2.3628 > 208.67.219.230.80: S 2830161306:2830161306(0) win 5840 <mss
1460,sackOK,timestamp 2269970 0,nop,wscale 2>
04:05:01.114193 IP 208.67.219.230.80 > 172.16.0.2.3628: S 588283986:588283986(0) ack 2830161307 win 65535 <mss
1452,nop,wscale 3,sackOK,timestamp 2066135758 2269970>
04:05:01.114347 IP 172.16.0.2.3628 > 208.67.219.230.80: . ack 1 win 1460 <nop,nop,timestamp 2270027 2066135758>
04:05:12.292049 IP 172.16.0.2.3628 > 208.67.219.230.80: P 1:8(7) ack 1 win 1460 <nop,nop,timestamp 2272821
2066135758>
04:05:12.558631 IP 208.67.219.230.80 > 172.16.0.2.3628: P 1:446(445) ack 8 win 8280 <nop,nop,timestamp 2066147208
2272821>
04:05:12.558832 IP 172.16.0.2.3628 > 208.67.219.230.80: . ack 446 win 1728 <nop,nop,timestamp 2272888 2066147208>
04:05:12.558724 IP 208.67.219.230.80 > 172.16.0.2.3628: F 446:446(0) ack 8 win 8280 <nop,nop,timestamp 2066147208
2272821>
04:05:12.559762 IP 172.16.0.2.3628 > 208.67.219.230.80: F 8:8(0) ack 447 win 1728 <nop,nop,timestamp 2272888
2066147208>
04:05:12.786349 IP 208.67.219.230.80 > 172.16.0.2.3628: . ack 9 win 8279 <nop,nop,timestamp 2066147437 2272888>
```

3.1.3.5 example code

```c
struct sockaddr_in remote_address;

memset(&remote_address, 0, sizeof(remote_address));

remote_address.sin_family = PF_INET;
remote_address.sin_addr.s_addr = htonl("192.168.0.6");
Local_address.sin_port = htons(8080);

//int connect(int socket, const struct sockaddr *address, socklen_t address_len);
if (connect(fd, (struct sockaddr *) &remote_address, sizeof(remote_address)) < 0)
{
  fprintf(stderr, "connect() failed: %s\n", strerror(errno));
}
else
{
  fprintf(stdout, "connect() successfully.\n");
}
```

3.1.4 listen()

- listen()调用使socket在协议地址上进行监听，并为该socket建立一个连接队列，将到达的服务请求保存在此队列中，直到程序处理它们。

```
NAME
       listen - listen for connections on a socket

SYNOPSIS
       #include <sys/socket.h>

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

- 返回值
  - 成功返回0
  - 失败返回-1，并且设置errno
- 参数
  - s – socket()调用获得的文件描述符
  - backlog – 积压值
    - backlog指定在请求队列中允许的最大请求数，进入的连接请求将在队列中等待被accept( )。
    - backlog对队列中等待服务的请求的数目进行了限制，大多数系统缺省值为20。
    - 一个请求到来时，如果输入队列已满，该socket将拒绝连接请求，客户将收到一个出错信息。

3.1.4.1 说明

- To accept connections, a socket is first created with socket(2), a willingness to accept incoming connections and a queue limit for incoming connections are specified with listen, and then the connections are accepted with accept(2).
- The listen call applies only to sockets of type SOCK_STREAM or SOCK_SEQPACKET.
- The backlog parameter defines the maximum length the queue of pending connections may grow to.
  - If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that retries succeed.
- The behaviour of the backlog parameter on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using the tcp_max_syn_backlog sysctl. When syncookies are enabled there is no logical maximum length and this sysctl setting is ignored. See tcp(7) for more information.

3.1.4.2 Backlog

- Backlog 积压值, listen queue
- 在Linux 2.2之前，表示未完成三次握手的队列的最大长度
- 从Linux 2.2开始，表示已经完成三次握手的队列的最大长度
  - 如果需要设置未完成三次握手的队列的最大长度，需要使用sysctl去设置参数tcp_max_syn_backlog
- Backlog值设置为多大合适？
  - 如果设置太大，会造成accept()时间太长，客户端等待时间太长，用户体验很差
  - 如果设置太小，就造成DoS（Deny of Service，拒绝服务）

3.1.4.2 进一步的说明

- backlog指定了正在等待连接的最大队列长度，它的作用在于处理可能同时出现的几个连接请求。
  - 例如，假定backlog参数为2，如果三个客户机同时发出请求，那么前两个会被放在等待处理的队列中，以便服务器程序依次为它们提供服务，而第三个连接的客户则会被拒绝连接。
  - DoS(拒绝服务)攻击即利用了这个原理，非法的连接占用了全部的连接数，造成正常的连接请求被拒绝。
- 调用listen()成功后后，socket变成了监听socket (listening socket)，在TCP状态变迁图中，处于LISTEN状态(可使用*netstat -l*查看)
  - 处于LISTEN状态的socket只能做accept()/close()操作，不能做read()/write()操作。

3.1.4.3 example code

```
// XXX: step 3, listen
// int listen(int sockfd, int backlog);
if (listen(listening_socket, LISTEN_BACKLOG) < 0)
{
  fprintf(stderr, "listen() failed: %s\n", strerror(errno));
  close(listening_socket);
  exit(1);
}

fprintf(stdout, "listen() ok.\n");
```

3.1.5 accept()

- 用listen()建立好输入队列后，服务器可以调用accept()，阻塞式的等待客户的连接请求。

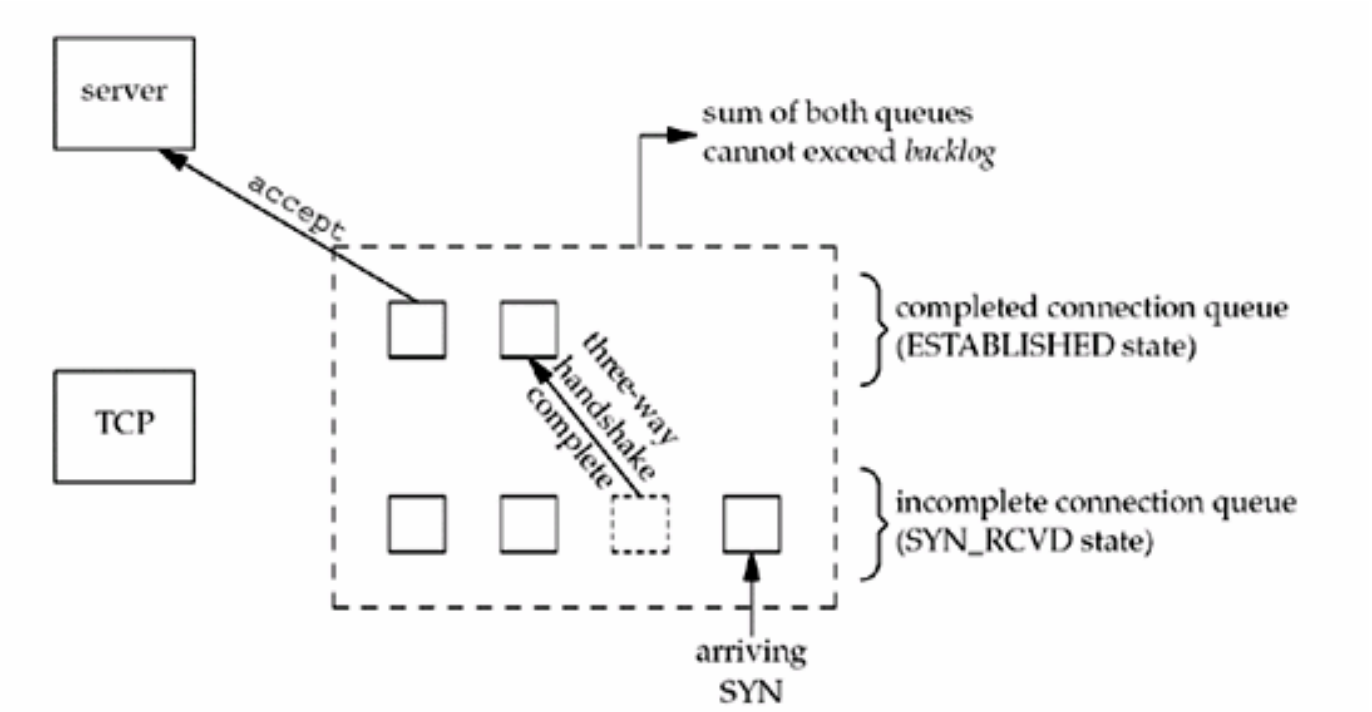
```
NAME
       accept - accept a connection on a socket

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
```

- 返回值
  - 如果成功，返回一个新的已连接的socket，使用这个socket可以和对方进行通信，而原来的监听socket仍然可以接受其他客户的连接
  - 如果失败，返回-1，并且设置errno
- 参数
  - s是接受客户连接的socket，即处于监听状态的socket(listening socket)
  - addr用于接收外来连接的地址信息，如果暂时不关心该地址信息，则可以置为NULL
  - addrlen是addr结构的长度，为值-结果参数

3.1.5.1 说明

- The accept function is used with connection-based socket types (SOCK_STREAM, SOCK_SEQPACKET and SOCK_RDM).
  - It extracts the first connection request on the queue of pending connections, creates a new connected socket with mostly the same properties as s, and allocates a new file descriptor for the socket, which is returned.
  - The newly created socket is no longer in the listening state.
  - The original socket s is unaffected by this call.
  - Note that any per file descriptor flags (everything that can be set with the F_SETFL fcntl, like non blocking or async state) are not inherited across an accept.
- The argument s is a socket that has been created with socket(2), bound to a local address with bind(2), and is listening for connections after a listen(2).
- The argument addr is a pointer to a sockaddr structure.
  - This structure is filled in with the address of the connecting entity, as known to the communications layer.
  - The exact format of the address passed in the addr parameter is determined by the socket's family (see socket(2) and the respective protocol man pages).
  - The addrlen argument is a value-result parameter:
    - it should initially contain the size of the structure pointed to by addr;
    - on return it will contain the actual length (in bytes) of the address returned.
  - When addr is NULL nothing is filled in.
- If no pending connections are present on the queue, and the socket is not marked as non-blocking, accept blocks the caller until a connection is present. If the socket is marked non-blocking and no pending connections are present on the queue, accept returns EAGAIN.
- In order to be notified of incoming connections on a socket, you can use select(2) or poll(2).
  - A readable event will be delivered when a new connection is attempted and you may then call accept to get a socket for that connection.
  - Alternatively, you can set the socket to deliver SIGIO when activity occurs on a socket; see socket(7) for details.

3.1.5.2 listen box



3.1.5.3 example code

```
struct sockaddr_in peer_address;
socklen_t peer_address_length;
int new_accepted_socket;

peer_address_length = sizeof(peer_address);

//int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
if ((new_accepted_socket =
     accept(listening_socket, (struct sockaddr *) &peer_address, &peer_address_length)) < 0)
{
  fprintf(stderr, "accept() failed: %s\n", strerror(errno));
  // FIXME: how to do?
}
else
{
  fprintf(stdout, "Accepted a new connection from %s:%d\n",
          inet_ntoa(peer_address.sin_addr), ntohs(peer_address.sin_port));

  // add new_accepted_socket to rset, wset
  //FD_SET(new_accepted_socket, &global_read_set);
  //FD_SET(new_accepted_socket, &global_write_set);
  //FD_SET(new_accepted_socket, &global_except_set);
  net_register_read(new_accepted_socket);
}
```

3.1.6 read()/write()

```
#include <unistd.h>

ssize t read(int fd, void *buf, size t count);
ssize_t write(int fd, const void *buf, size_t count);
```

- 在Unix/Linux中，socket被看成文件描述符，所以常用来进行文件读写的read()和write()同样可以读写socket连接。
- read()试图从socket_fd读出n个字节，并将之放到buf里去。如果成功，返回被读出的字节数目（如果返回0，表明读到文件结束，对于socket来说，返回0表示对端关闭了连接），读文件的位置自动向前。如果读出的数目小于n，则可能是接近文件结尾、从终端读、或者被信号中断等原因。如果失败，返回-1，有可能还没开始读，就被信号中断，此时返回-1，并置errno为EINTR。
- write()试图往socketfd写n个字节，写的内容从buf里取。如果成功，返回写了的字节数目，如果是0，表明没有写，如果返回-1，表明遇到错误或被信号中断，如果还没开始写，就被信号中断，则返回-1，并置errno为EINTR。

3.1.7 recv()/send()

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

ssize_t recv(int s, void *buf, size_t len, int flags);
ssize_t send(int s, const void *buf, size_t len, int flags);
```

- recv()和send()系统调用提供了和read()和write()差不多的功能，不过它们提供了第四个参数来控制读写操作，更加灵活
- 第四个参数可以是0或者是以下的组合。

flags	Description	recv	send
MSG_DONTROUTE	Bypass routing table lookup		•
MSG_DONTWAIT	Only this operation is nonblocking	•	•
MSG_OOB	Send or receive out-of-band data	•	•
MSG_PEEK	Peek at incoming message	•	
MSG_WAITALL	Wait for all the data	•	

3.1.8 recvfrom()/sendto()

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

ssize_t recvfrom(int s, void *buf, size_t len, int flags, struct sockaddr *from, socklen_t *fromlen);
ssize_t sendto(int s, const void *buf, size_t len, int flags, const struct sockaddr *to, socklen_t tolen);
```

- recvfrom()可以用在UDP和TCP两种情况下
  - recv()等同于在recvfrom()里把后两个参数置为NULL
  - 系统收到一个消息后，如果参数from指针非空，而且socket不是面向连接类型(如TCP)的，消息的源地址被填入from所指的地方，fromlen所指的地方也被填为相应的值。
  - 返回值：收到的字节数，出现错误返回-1。
- sendto()可以用在UDP和TCP两种情况下。
  - 该函数比send()函数多了两个参数，to表示目地机的IP地址和端口号信息。
  - sendto()函数返回实际发送的数据字节长度，出现发送错误时返回-1

3.1.9.1 close()

```
#include <unistd.h>

int close(int fd);
```

- 用于TCP时，如果发送队列非空，发送之，然后终止TCP连接。
- close掉一个fd后，这个fd值就不再代表什么连接了。不能再把这个fd作为read或write的参数了。
- 返回值：成功为0，出现错误为-1。
- 人们通常忽略对close返回值的检查，但是最好进行检查，如write成功返回往往不代表确实写成功，如果写出错，close的返回值为-1。

3.1.9.2 shutdown()

```
NAME
       shutdown - shut down part of a full-duplex connection

SYNOPSIS
       #include <sys/socket.h>

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

- TCP连接是双向的(是可读写的)，当我们使用close()时，会把读写通道都关闭，有时侯我们希望只关闭一个方向，这个时候我们可以使用shutdown()。
- 针对不同的howto，系统回采取不同的关闭方式。
  - howto=0
    - 这个时候系统会关闭读通道，但是可以继续往接字描述符写。
  - howto=1
    - 关闭写通道，和上面相反，着时候就只可以读了。
  - howto=2
    - 关闭读写通道，和close()一样

3.2 网络信息检索函数

- gethostname()　获得主机名
- getpeername()　获得与套接口相连的远程协议地址
- getsockname()　获得套接口本地协议地址
- gethostbyname()　根据主机名取得主机信息
- gethostbyaddr()　根据主机地址取得主机信息
- getprotobyname()　根据协议名取得主机协议信息
- getprotobynumber()　根据协议号取得主机协议信息
- getservbyname()　根据服务名取得相关服务信息
- getservbyport()　根据端口号取得相关服务信息

3.3 Socket options相关的系统调用

- ioctl()/fcntl() 设置套接口的工作方式
- getsockopt()/setsockopt() 获取/设置一个套接口选项

Socket控制及选项

- fcntl()对文件描述符进行控制，比如设置为非阻塞方式工作等等，提供的是通用的命令，对socket这种特殊描述符没有特殊处理，就是把socket当成描述符来对待的
- ioctl()是对设备进行控制，有针对性，对不同的设备有不同的控制指令和参数，对socket来说，可以对路由表操作，可以对接口操作
- 有些配置既可以通过fcntl()实现，也可以通过ioctl()实现，比如设置非阻塞方式工作
- setsockopt()/getsockopt()对socket进行设置，包括设置工作方式，调整参数等，实际上是对内核里的协议栈进行操作的。
  - 工作方式：广播
  - 调整参数：调整缓冲区大小，超时时间等等
  - 实现某些功能：实现多播功能

3.3.1 ioctl()/fcntl()

- 设置socket为非阻塞模式

```
#include <unistd.h>
#include <fcntl.h>

int save_flags;

//int fcntl(intfd, int cmd);
save_flags = fcntl(socket_fd, F_GETFL);

save_flags |= O_NONBLOCK;

fcntl(socket_fd, F_SETFL, save_flags);
```

- 对于read()/recv()/recvfrom()，
  - 如果没有可用的消息，返回-1，errno会被设置为EAGAIN(=EWOULDBLOCK)，如果有可用的，就返回可用的部分，会小于等于指定的字节数。
- 对于write()/send()/sendto()
  - 如果消息无法放入发送buffer，返回-1并且将errno置为EAGAIN。
- 对于accept()
  - 如果不行，返回-1，errno = EAGAIN
- 对于connect()
  - 如果不行，返回-1，errno = EINPROGRESS

3.3.2 getsockopt()/setsockopt()

```
NAME
       getsockopt, setsockopt - get and set options on sockets

SYNOPSIS
       #include <sys/types.h>
       #include <sys/socket.h>

       int getsockopt(int s, int level, int optname, void *optval, socklen_t *optlen);

       int setsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);
```

- getsockopt()和setsockopt()处理和一个socket相关的选项，这些选项会影响到多个层次的协议；如果处理的是socket层相关的选项，level设为SOL_SOCKET, IP层：IPPROTO_IP, TCP层：IPPROTO_TCP。
- 对于setsockopt()，参数optval（1使能，0使之无效）和optlen用来写相关的选项的值
- 对于getsockopt()，optval是一个用来存放返回值的buffer（通常是int型），optlen是一个value-result参数，调用的时候填buffer的长度，调用完毕，该值被改为实际返回值的长度。
- optname指定了选项名。
- 使用socklen_t这个类型是POSIX定义的。

3.3.2 Socket的层(level)

- SOL_SOCKET(通用选项)
- IPPROTO_IP(IP层选项)
- IPPROTO_ICMPV6(ICMPv6选项)
- IPPROTO_IPV6(IPv6选项)
- IPPROTO_TCP(TCP选项)

3.3.2 参数

- setsockopt()/getsockopt()参数有三种情况：
  - 开关：0表示关闭，非0表示打开
    - SO_BROADCAST
    - SO_REUSEADDR/SO_REUSEPORT
  - 值：将option设置为该值大小
    - SO_RCVBUF/SO_SNDBUF
    - SO_RCVLOWAT/SO_SNDLOWAT
  - 结构体，不同的option可能定义不同的结构体，其成员的意义分别说明
    - SO_LINGER
    - SO_RCVTIMEO/SO_SNDTIMEO

3.3.2 选项举例

- SO_RCVBUF – 接收缓冲区大小
- SO_SNDBUF – 发送缓冲区大小
- SO_RCVLOWAT – 接收低潮值
- SO_SNDLOWAT – 发送低潮值
- SO_REUSEADDR – 地址重用
- SO_BROADCAST – 广播
- IP_ADD_MEMBERSHIP – 加入多播组
- IP_DROP_MEMBERSHIP – 离开多播组

3.3.2 example code

```c
  int value;

  value = 1;
  // int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);
  if (setsockopt(listening_socket, SOL_SOCKET, SO_REUSEADDR, &value, sizeof(value)) < 0)
  {
    fprintf(stdout, "Warning: setsockopt() failed: %s\n", strerror(errno));
  }

#ifdef _DEBUG_
  fprintf(stdout, "Debug: setsockopt(SO_REUSEADDR) successfully.\n");
#endif

#ifdef SO_REUSEPORT
  value = 1;
  // int setsockopt(int socket, int level, int option_name, const void *option_value, socklen_t option_len);
  if (setsockopt(listening_socket, SOL_SOCKET, SO_REUSEPORT, &value, sizeof(value)) < 0)
  {
    fprintf(stdout, "Warning: setsockopt() failed: %s\n", strerror(errno));
  }

#  ifdef _DEBUG_
  fprintf(stdout, "Debug: setsockopt(SO_REUSEPORT) successfully.\n");
#  endif
#endif
```

3.4 调度相关

- select()
- poll()
- epoll()

I/O多路复用

- I/O多路复用的三个调用
  - select(),来源于BSD
  - poll()，来源于System V
  - epoll()，Linux特有的
- 相同点
  - 作用是一样的，通知该函数我们希望的事情，该函数会告诉我们结果
- 不同点
  - 参数不相同
    - select()使用的是fdset
    - poll()用的是pollfd，一个结构体
    - epoll()用的是struct epoll_event，对于用户进程来说，数据结构不是很重要。
  - 性能，select()/poll()是O(n)，epoll()是O(1)
    - select()/poll()返回的结果没有独立出来，用户进程需要遍历查找已经就绪的来进行处理
    - epoll()将已经就绪的通过独立的集合返回，这样用户进程可以直接访问该结果集，命中率是100%
  - epoll()能够自动的清除已经关闭的描述符，select()/poll()要由程序来清除已经关闭的描述符。
  - select()最麻烦，poll()较为简单，epoll()最简单
  - select()所管理的描述符数较少，poll()较大，epoll()很大。
  - select()检查的状态比较少，poll()支持的状态检查数更多，epoll()支持两种触发方式(ET/LT)

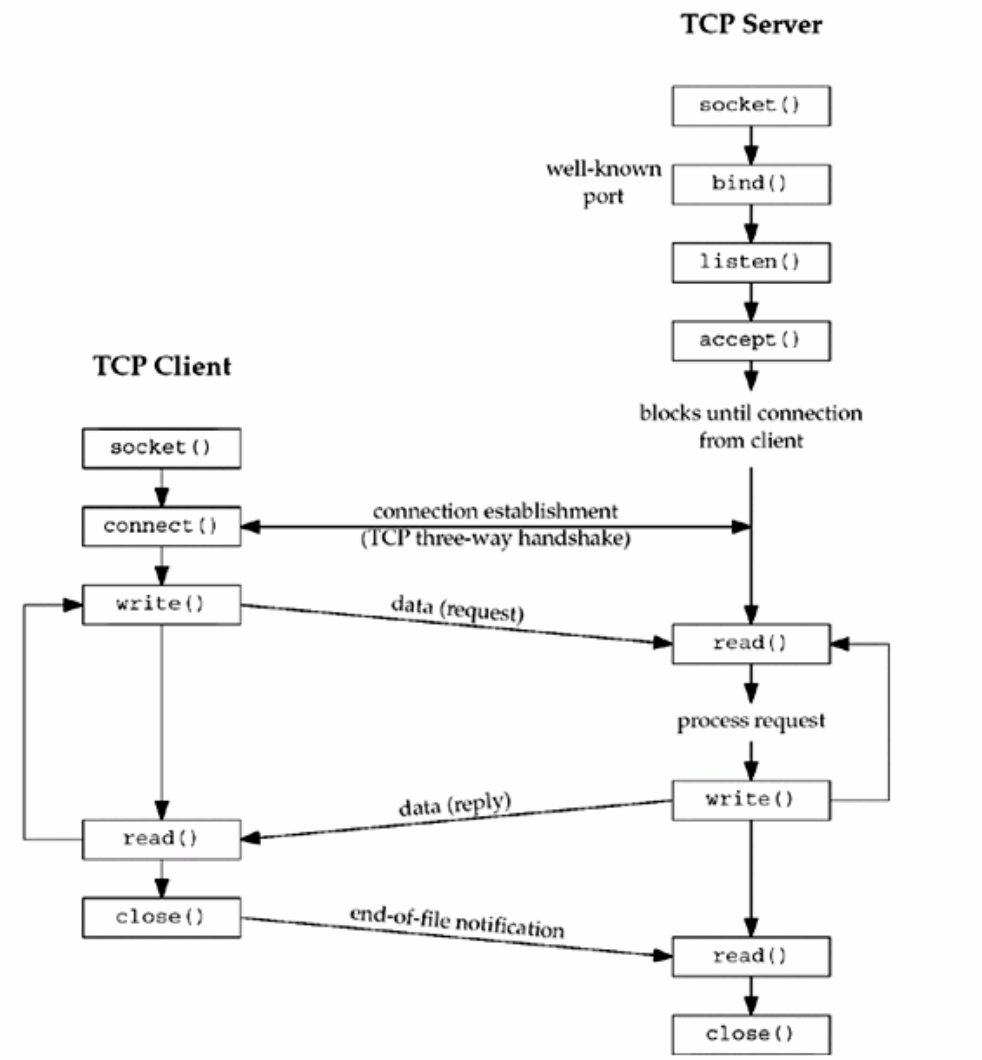
3.6 系统I/O模型

- 系统I/O与服务器模型
- 阻塞I/O模型
- 非阻塞I/O模型
- 多路复用I/O模型
- 信号驱动I/O模型
- 异步I/O模型
- I/O模型比较

4. 编程

- TCP编程
- UDP编程
- Unix domain编程

4.1.1 TCP编程流程


TCP Server
socket()
well-known port
bind()
listen()
accept()
blocks until connection from client
TCP Client
socket()
connect()
connection establishment (TCP three-way handshake)
write()
data (request)
read()
process request
write()
data (reply)
read()
close()
end-of-file notification
read()
close()

4.1.2 TCP编程注意事项

- 套接字只有处于ESTABLISHED状态，才可以进行数据收发
  - 客户端调用connect()成功之后，进入ESTABLISHED状态
  - 服务器端调用accept()成功之后得到的新的描述符已经处于ESTABLISHED状态
- 数据发送和接收必须检查返回值
  - 发送时，实际发送成功的字节数与期望发送的字节数不一定相等，也就是说，数据可能一次发送不完，要循环发送多次，直到数据全部发送完毕
  - 接收时，不能假定期望接收的数据一次接收完成，可能需要循环来确保接收到了期望长度的数据
- 连接断开的检查
  - 在读数据时，如果read()/recv()返回0表示连接的对方关闭了连接
  - 在写数据时，需要捕获SIGPIPE信号来检测连接的对方关闭了读端，此时write()/send()调用返回-1，errno设置为EPIPE
- 哪端主动断开？
  - 一般来说，服务器端应该控制断开的主动权，不能依赖客户端断开之后自己再被动断开，否则如果客户端程序设计有问题，或者客户端恶意的不断开连接，服务器端就会造成DoS。
- 服务器端需要设置SO_REUSEADDR option来解决2MSL的问题
  - Linux不支持SO_REUSEPORT，但是其他系统可能需要设置该option
- 保持连接的问题
  - 可以设置SO_KEEPALIVE option来保持连接
  - 最好应用程序自己做keep alive，这样能够更稳妥

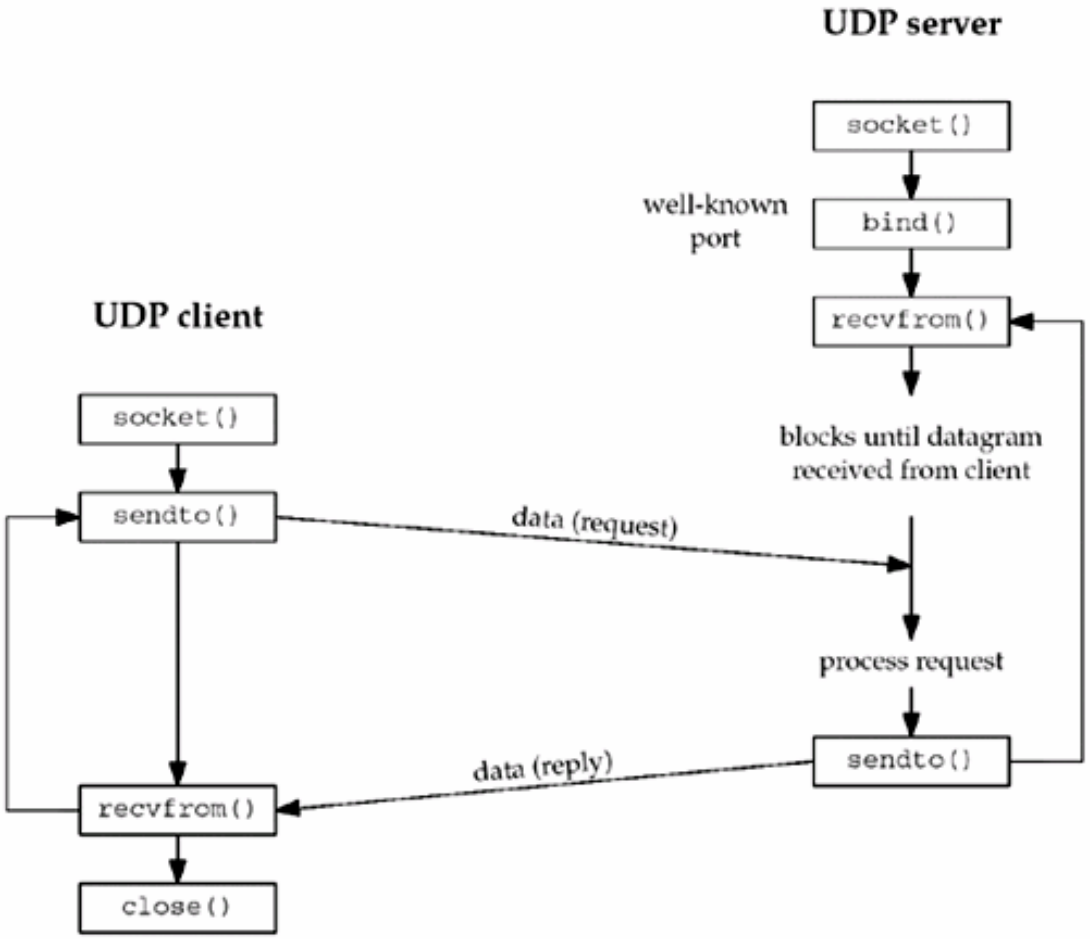
4.1.3 多进程并发服务器编程要点

- fork()的时机
  - 在accept()调用成功之后做fork()
  - 产生的子进程处理新建立的连接（在该连接上进行数据读写）
  - 父进程继续进行accept()，接受新的连接
- 子进程要做的事情
  - 关闭listening_socket，参考APUE Page 175
  - 处理完毕之后，关闭new_connected_socket
  - 处理完之后要显式的终止，调用exit()
- 父进程要做的事情
  - 关闭new_connected_socket，参考APUE Page 175
  - 通过SIGCHLD信号触发方式为子进程收尸

4.1.4 多线程并发服务器编程要点

- 创建线程的时机
  - 在accept()成功之后创建工作线程
  - 新线程（工作线程）一般创建为分离属性的线程
- 新线程需要做的事情
  - 新线程需要在完成数据收发之后关闭描述符
  - 如果使用了线程取消机制，则工作线程必须在cleanup handler 里边设置关闭描述符
- 需要注意的其他问题
  - 创建线程时，传递描述符给start_routine时要注意参数传递的安全问题
  - 工作线程不要对listening_socket做任何的处理

4.2.1 UDP编程流程


UDP server
socket()
well-known port
bind()
recvfrom()
blocks until datagram received from client
process request
sendto()
UDP client
socket()
sendto()
data (request)
data (reply)
recvfrom()
close()

4.2.2 UDP编程注意事项

- 如果不希望通讯的双方被第三方打扰，可以进行connect()，这样只有参与通讯的双方之间能够交换数据
- sendto()成功不表示数据一定发送出去了，sendto()成功只表示应用程序的数据已经递交给了传输层
- UDP允许发送0长度的数据，因此recvfrom()返回0不是错误，是正常的
- 使用UDP通讯时，一定要注意MTU的影响，如果包大小超过路径MTU，则可能被路由器丢弃数据，进而造成通讯失败
  - 必要时要做主动的拆包操作，比如传输视频时
  - 在有些情况下，也需要做拼包处理，比如音频采集通讯时，每个音频采集的数据都很小，只有几个字节，如果不拼包直接发送，则传输的有效载荷太低
  - 在有些系统中，如果传输大量的小包，有可能会被IDS认为是flood攻击，IDS可能会触发防火墙可能会动作，造成传输失败
- 在输出数据量很大时，需要调整socket的发送缓冲区和接收缓冲区大小
  - SO_RCVBUF/SO_SNDBUF

4.2.3 广播编程

- 接收端不需要做特殊处理，只需要正常绑定IP地址和端口号
  - 一般广播的接收端程序需要绑定所有的IP地址(INADDR_ANY)
- 发送端需要设置SO_BROADCAST选项
  - 在未这是SO_BROADCAST选项的情况下，会报permission denied错误
- 发送端需要向广播地址发送数据
  - 可以是绝对广播地址(也称为受限广播地址)
  - 可以是子网广播地址

4.2.4 多播编程

- 发送端通常来说不做特殊处理，使用普通的UDP发送程序就可以了
  - 如果希望本机发出的多播包也可以被本机的进程收到，需要设置IP_MULTICAST_LOOP选项
  - 该选项Linux系统默认设置为enable状态，发送端要显式的设置IP_MULTCAST_LOOP选项，确保移植性
- 接收端进程需要做特殊处理
  - 加入多播组，需要设置IP_ADD_MEMEBERSHIP
  - 离开多播组，需要设置IP_DROP_MEMBERSHIP
- 控制多播数据可以走多远
  - 可以设置IP_MULTICAST_TTL选项
- 外出多播数据的接口
  - 可以通过IP_MULTICAST_IF选项设置
  - 在主机没有配置缺省路由的情况下，如果不指定外出接口，会报错
- 参考资料：
  - http://tldp.org/HOWTO/Multicast-HOWTO-6.html

4.3 UNIX domain编程

- 接口是socket
  - domain(protocol faimly)=PF_UNIX/PF_LOCAL
  - address family = AF_UNIX/AF_LOCAL
  - 地址结构：struct sockaddr_un
  - 类型：流式(SOCK_STREAM)，类似于TCP/数据报(SOCK_DGRAM)，类似于UDP
- 内部的实现机制类似于管道，相当于一个全双工的管道。
- 优势：本地通讯速度快
  - 如果本地通讯使用网络127.0.0.1，数据包要从应用程序传输给传输层，再传给网络层，网络层发现数据包的目的地址是本机，则传给传输层，再传给应用层
  - 使用UNIX domain方式通讯，相当于两个进程之间建立管道，数据类似于直接拷贝。
- 应用：
  - X-window，本地通讯可以使用unixdomain
  - MySQL，本地数据库连接，使用unixdomain
  - Syslog，本地通讯使用unixdomain

4.3 UNIX domain通讯中path的问题

- 使用unix domain协议通讯的客户端和服务器之间，通过一个文件系统的路径名来找到对方
  - 该文件系统的路径名可以是绝对路径或者相对路径
  - 实质上是通过一个inode找到对方。
  - 需要注意的是，区分的是文件系统的路径名而不是字符串
- 客户端与服务器之间通讯不通过文件，因此socket文件的内容大小都是0。
- 实现unixdomain通讯的流式客户端和服务器端程序
  - 服务器端需要做bind
  - 客户端不需要做bind
- 实现unixdomain通讯的数据报客户端和服务器端程序
  - 客户端必须做bind()，否则服务器数据无法返回
  - 客户端运行时需要指定两个socket文件，一个是自己需要绑定的socket文件，另一个是服务器程序绑定的socket文件（用于sendto调用的目标地址），两个文件名不能相同

4.4 DNS编程

- gethostbyname()
- getaddrinfo()

4.5 嵌入式环境网络编程

- 网络通信常常是在不同的操作系统之间运行，所以网络应用程序必须考虑不同体系结构以及OS之间的差异:
  - 字节顺序
    - 对双字节与4字节数据进行字节序转换
  - 字的长度
    - 不同的OS对于相同的数据类型可能有不同的表示长度。
  - 字节边界问题
    - 不同的OS对相同的数据类型打包方式不相同，因为它们的定界限制不一样。
      - 例如：结构struct{char a;int b}
- 解决方法
  - 方法一：对于具有相同字节顺序的OS，通信双方均以单字节定界；对于具有不同字节顺序的OS，显示地定义格式（位数、字节顺序类型）
  - 方法二：将需要发送的消息的结构在发送前变换成一种统一的格式，到达接收方后再执行相反的过程。

5. 作业: 实现简单的WEB服务器

实现一个简单的 Web 服务器 myhttpd。服务器程序启动时要读取配置文件 /etc/myhttpd.conf，其中需要指定服务器监听的端口号和服务目录，例如：

```
Port=80
Directory=/var/www
```

注意，1024 以下的端口号需要超级用户才能开启服务。如果你的系统中已经安装了某种 Web 服务器（例如 Apache），应该为 myhttpd 选择一个不同的端口号。当浏览器向服务器请求文件时，服务器就从服务目录（例如/var/www）中找出这个文件，加上 HTTP 协议头一起发给浏览器。但是，如果浏览器请求的文件是可执行的则称为 CGI 程序，服务器并不是将这个文件发给浏览器，而是在服务器端执行这个程序，将它的标准输出发给浏览器，服务器不发送完整的 HTTP 协议头，CGI 程序自己负责输出一部分 HTTP 协议头。

Any questions?

Contact:

- yjs@oldhand.org
- 133 0122 6268