第3章　网络编程

本章主要介绍网络编程的一些主要方法，并配有编程实验。一些实验只提出要求，程序编写过程需要读者自行完成。

如果应用程序涉及本地与远程之间的通信，就需要采用网络编程。网络编程最主要的工作就是在发送端把信息通过规定好的协议进行包的组装，在接收端按照规定好的协议把包进行解析并提取出对应的信息，从而达到通信的目的。中间最主要的就是数据包的组装、过滤、捕获和分析以及其他处理。

通过使用套接字达到进程间通信目的的编程就是网络编程。套接字即 Socket，应用程序通常通过套接字向网络发出请求或者应答网络请求，实际上是网络应用程序接口（API）。套接字是由传输层提供的应用程序（进程）和网络之间的接入点，如图 3-1 所示。应用程序（进程）可以通过套接字访问网络，套接字利用主机的网络层地址和端口号为两个进程建立逻辑连接。

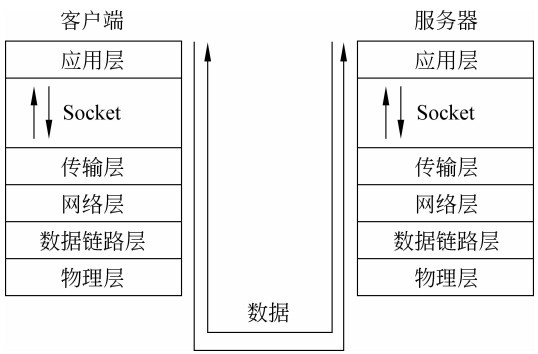

图 3-1　Socket 是应用层与传输层之间的桥梁

套接字可以用于多种协议，包括 TCP/IP 协议。常用的端口号如表 3-1 所示。

表 3-1　TCP/IP 常用端口号

协议	NNTP	FTP(数据)	FTP(控制)	Telnet	STMP	HTTP	POP3
端口号	19	20	21	23	25	80	110

为了方便网络编程，20 世纪 90 年代初，Microsoft 联合其他几家公司共同制定了一套 Windows 下的网络编程接口，即 Windows Sockets 规范。它不是一种网络协议，而是一套开放的、支持多种协议的 Windows 下的网络编程接口。现在的 Winsock 已经基本上实现了与协议无关，可以使用 Winsock 调用多种协议的功能，但较常使用的是 TCP/IP 协议。Socket 实际上是在计算机中提供了一个通信端口，可以通过这个端口与任何一个具有 Socket 接口的计算机通信。应用程序在网络上传输，接收的信息都通过这个 Socket 接口实现。

Socket 是 TCP/IP 网络的 API，Socket 接口提供了很多的函数，可以用于开发网络应用程序。Socket 数据传输是一种特殊的 I/O，同时 Socket 也是一种文件描述符。Socket 的使用主要有 Socket 建立、配置、建立连接、数据传输和结束传输等过程。

3.1 利用套接字建立逻辑信道

一般发起通信请求的程序被称为客户端，用户一般是通过客户端软件访问某种服务。客户端应用程序通过与服务器建立连接和发送请求，然后接收服务器返回的内容。服务器则一般是等待并处理客户端请求的应用程序。服务器通常由系统执行，在系统生存期间一直存在和等待客户端的请求，并且在接收到客户端的请求后，根据请求向客户端返回合适的内容。

通信的一方（被动方，称为服务器）监听某个端口；通信的另一方（主动方，称为客户端）如果知道服务器的 IP 地址和它所监听的端口，便可以试图发送请求建立连接。该连接请求包含：服务器 IP 地址、服务器端口号、客户 IP 地址、客户端口号。由于客户端口号由客户端的系统（TCP 进程）自动选取一个当前未用的端口，该四元组便可以在因特网中唯一标识一个逻辑连接。服务器收到客户端发来的连接请求后，便发出响应建立该连接，这样就建立了一条逻辑信道。

客户和服务器通过请求响应方式可以进行双向数据传输。当结束数据传输时，需要关闭该连接。这种工作模式是有连接的客户端/服务器模式（Client/Server）。

根据连接启动的方式以及本地套接字要连接的目标，套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认。

（1）服务器监听：是指服务器端套接字并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。

（2）客户端请求：是指由客户端的套接字提出连接请求，要连接的目标是服务器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后向服务器端套接字提出连接请求。

（3）连接确认：是指当服务器端套接字监听到或者接收到客户端套接字的连接请求，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

这三个步骤类似于三次握手，如图 3-2 所示。

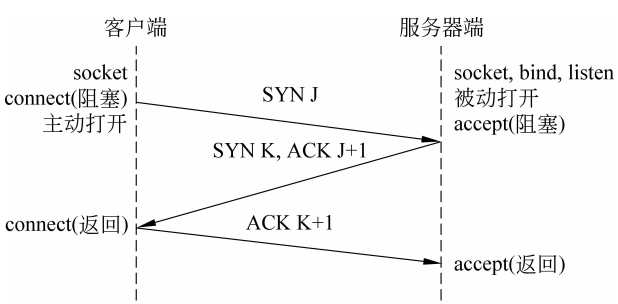

图 3-2 客户端与服务器端的三次握手

3.2 Client/Server 工作模式分类

Client/Server 工作模式一般按下列分类：

(1) 有状态和无状态：服务器是否记录客户端的当前状态。

(2) 有连接(TCP)和无连接(UDP)：客户端和服务器之间是否先建立连接再传输数据。

(3) 循环和并发：服务器对多客户端请求的服务是采用循环方法还是并发程序方法。

TCP 协议面向连接，使用可靠的字节流传送服务；而 UDP 协议面向非连接，使用非可靠的数据报服务。TCP 协议提供高可靠性的传输，UDP 协议提供高效的传输。它们在实际应用中有其各自所适应的场合。

3.3 面向连接的 Client/Server 模式

在面向连接的 Client/Server 结构中，操作过程采取的是主动请求方式：服务器首先启动，并根据请求提供相应服务。

通过调用 socket()建立一个套接口，然后调用 bind()将该套接口和本地网络地址联系在一起，再调用 listen()使套接口做好侦听的准备，并规定它的请求队列的长度，之后调用 accept()接收连接。客户端在建立套接口后就可调用 connect()和服务器建立连接。连接一旦建立，客户机和服务器之间就可以供对方读取或者读取对方数据。最后在数据传送结束后，双方调用 close()关闭套接口。

3.3.1 面向连接的服务器工作流程

面向连接的服务器工作流程包括以下几个环节。

1. 创建套接字

Socket 建立是通过调用 Socket 函数实现的，该函数定义如下：

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

其中参数：

domain：指明使用的协议族，如果取值 AF_INET，用于网络通信；如果取值 AF_UNIX，用于单一 UNIX 系统中进程间通信。

type：指明 socket 类型，如果取值 SOCK_STREAM，表示是流式，面向连接的比特流，顺序、可靠、双向，用于 TCP 通信；如果取值 SOCK_DGRAM，表示数据报式、无连接、定长、不可靠，用于 UDP 通信。

protocol：由于指定了 type，一般用 0。

函数返回：一个整型的 socket 描述符，供后面使用。如果调用失败，返回一个 INVALID_SOCKET 值，错误信息可以通过 WSAGetLastError 函数返回。

例如，一个 socket 可如下建立：

```
int sockfd=socket(AF_INET,SOCK_STREAM,0)
```

2. 将本地 IP 地址和端口号绑定到套接字

Socket 的建立实际上是为 socket 数据结构分配了一个名字空间并返回指针，接着要对数据结构提供数据。bind()将一本地地址与一套接口捆绑，它适用于未连接的数据报或流类套接口，在 connect()或 listen()调用前使用。bind()函数通过给一个未命名套接口分配一个本地名字为套接口建立本地捆绑(主机地址/端口号)。

bind()定义如下：

```
int bind(SOCKET socket, struct sockaddr * address, int addr_len)
```

其中参数：

sockfd：由 socket()调用返回的套接口文件描述符。

sockaddr：数据结构 sockaddr 中包括了关于本地地址、端口和 IP 地址的信息。

addr_len：地址长度，可以设置成 sizeof(structsockaddr)。

通常服务器在启动时都会绑定一个众所周知的地址(如 IP 地址＋端口号)，用于提供服务，客户端可以通过它连接服务器；而客户端不用指定，有的系统会自动分配一个端口号和自身的 IP 地址组合。这就是为什么通常服务器端在 listen()之前会调用 bind()，而客户端就不会调用，而是在 connect()时由系统随机生成一个。

函数返回：如无错误发生，则 bind()返回 0；否则返回 SOCKET_ERROR，应用程序可通过 WSAGetLastError()获取相应错误代码。

3. 服务端使用 listen()开启监听

listen()在套接字函数中表示让一个套接字处于监听到来的连接请求的状态。从客户端发来的连接请求将首先进入该等待队列，等待本进程的处理。listen()定义如下：

```
int listen(SOCKET socket, int backlog)
```

其中参数：

socket：一个已绑定未被连接的套接字描述符。

backlog：进入队列中允许的连接的个数。进入的连接请求在使用系统调用 accept()应答之前要在进入队列中等待。该值是队列中最多可以拥有的请求的个数，大多数系统的默认设置为 20。

函数返回：无错误返回 0；否则返回 SOCKET ERROR，可以调用函数 WSAGetLastError 取得错误代码。

例如，listen(s,1)表示连接请求队列长度为 1，即只允许有一个请求，若有多个请求，则出现错误，给出错误代码 WSAECONNREFUSED。

4. 接受从客户端发来的请求

accept()是网络编程的重要函数，其作用是在一个套接口接受一个连接，其头文件对于 Windows 系统是在＃include＜winsock.h＞中，而 Linux 系统则在＃include ＜sys/socket.h＞中。

accept()从端口的请求连接的等待连接队列中抽取第一个连接，创建一个与此同类的新的套接口并返回句柄。如果队列中无等待连接，且套接口为阻塞方式，则 accept()阻塞调用进程直至新的连接出现。如果套接口为非阻塞方式且队列中无等待连接，则 accept()返回一错误代码。已接受连接的套接口不能用于接受新的连接，原套接口仍保持开放。accept()定义如下：

```
SOCKET  accept(SOCKET socket, struct sockaddr * address, int addr_len)
```

其中参数：

Socket：正在监听端口的套接口文件描述符。

Address：客户端的 socket 地址。

addr_len：socket 地址的长度。

函数返回：如果没有错误产生，则 accept()返回一个描述所接收包的 SOCKET 类型的值；否则返回 INVALID_SOCKET 错误，应用程序可通过调用 WSAGetLastError()获得特定的错误代码。

5. 发送和接收数据

建立连接后，客户端和服务器端就可以进行数据传输了，通过使用 send()发送数据，使用 recv()接收数据。

```
int send(SOCKET  socket, char * message, int msg_len,int flags)
```

其中参数：

socket：发送数据的套接口文件描述符。它可以通过 socket()系统调用返回，也可以通过 accept()系统调用得到。

message：指向要发送的数据的指针。

msg_len：要发送数据的字节长度。

flags：标志，一般设置为 0。

函数返回：无错时返回实际发送的字节数，否则返回 SOCKET_ERROR。

```
int recv(SOCKET  socket, char * message, int msg_len,int flags)
```

其中参数：

socket：要读取的套接口文件描述符。

message：保存读入信息的缓冲区起始地址。

msg_len：缓冲区的最大长度。

flags：标志，一般设置为 0。

函数返回：无错时返回实际接收的字节数，否则返回 SOCKET_ERROR。

6. 关闭连接套接字

使用 close()调用关闭连接的套接口文件描述符：

```
int closesocket(SOCKET  socket);
```

之后就不能再对此套接口做任何的读/写操作。

7. 转 4 或结束

3.3.2 面向连接的客户端工作流程

1. 创建套接字

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

2. 发出连接请求

connect()用于建立与指定 socket 的连接。对于流类套接口(SOCK_STREAM 类型),利用名字与一个远程主机建立连接,一旦套接口调用成功返回,它就能收发数据了。对于数据报类套接口(SOCK_DGRAM 类型),则设置成一个默认的目的地址,并用它进行后续的 send()与 recv()调用。

```
int connect(SOCKET socket, struct sockaddr * address, int addr_len)
```

其中参数:

Socket:由系统调用 socket()返回的套接口文件描述符。

Address:指向数据结构 sockaddr 的指针,其中包括目的(即服务器)端口和 IP 地址。

addr_len:地址长度,可以使用 sizeof(struct sockaddr)获得。

函数返回:若无错误发生,则 connect()返回 0;否则返回 SOCKET_ERROR 错误,可通过 WSAGetLastError()获取相应错误代码。

3. 发送和接收数据

4. 关闭此连接的套接字

其工作过程如图 3-3 所示。

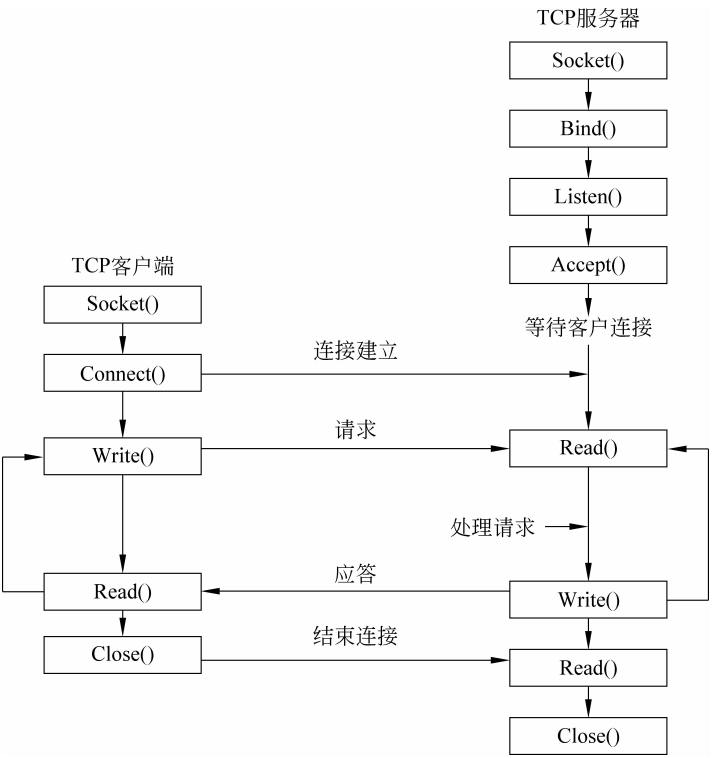

图 3-3 基本 TCP 客户-服务器工作过程

3.4 无连接的 Client/Server 模式

在无连接的 Client/Server 结构中，服务器使用 socket()和 bind()函数调用建立和连接 Socket。由于此时的 Socket 是无连接的，服务器使用 recvfrom()函数从 Socket 接收数据。客户端也只调用 bind()函数而不调用 connect()函数。注意：无连接的协议不在两个端口之间建立点对点的连接，因此 sendto()函数要求程序在一个参数中指明目的地址。recvfrom()函数不需要建立连接，它对到达相连协议端口的任何数据作出响应。当 recvfrom()函数从 Socket 收到一个数据报时，它将保存发送此数据包的进程的网络地址以及数据包本身。程序(服务器和客户)用保存的地址去确定发送(客户)进程。在必要的条件下，服务器将其应答数据报送到从 recvfrom()函数调用中所得到的网络地址中去。其工作过程如图 3-4 所示。

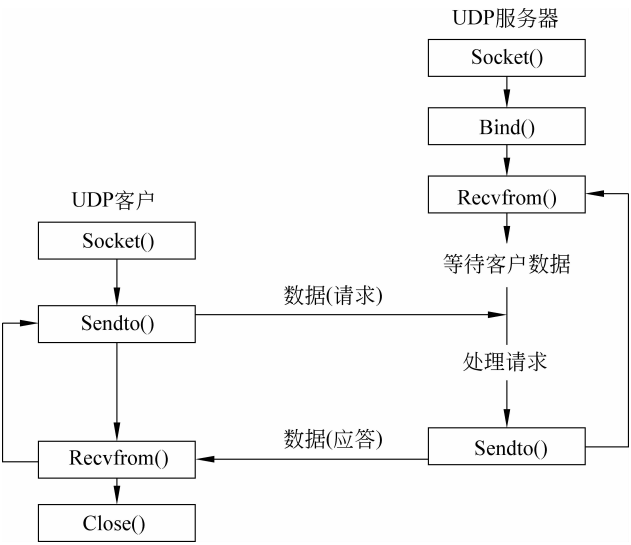

图 3-4 基本 UDP 客户-服务器工作过程

一般而言，大多数 TCP 服务器是并发的，而大多数 UDP 服务器是迭代的。多数 TCP 服务器是与调用 fork 处理每个客户连接的服务器并发执行的。迭代服务器没有对 fork 的调用，所以单一服务器进程就处理了所有客户。

3.5 编 程 实 验

在 Visual C++ 中进行 Winsock 的 API 编程开发时，需要在项目中使用下面 3 个文件，否则会出现编译错误。

(1) winsock.h：Winsock API 的头文件，需要包含在项目中。

(2) wsock32.lib：Winsock API 连接库文件。在使用中，一定要把它作为项目的非默认的连接库包含到项目文件中。

(3) winsock.dll：Winsock 的动态连接库，位于 Windows 的安装目录下。

【例 3-1】 下面是一个有连接的编程实例。程序分两部分：服务器端程序和客户端程序。

(1) 服务器端程序。

```
//TCPdtd_server.cpp -main, TCPdaytimed
#include<stdlib.h>
#include<stdio.h>
#include<winsock2.h>
#include<time.h>

void errexit(const char *,…);
void TCPdaytimed(SOCKET);
SOCKET passiveTCP(const char *, int);

#define QLEN    5
#define  WSVERS    MAKEWORD(2, 0)

/*------------------------------------------------------------------
 * main -Iterative TCP server for DAYTIME service
 *------------------------------------------------------------------
 */
void
main(int argc, char *argv[])
/* argc: 命令行参数个数, 例如:C:\>TCPdaytimed 8080,  argc=2 argv[0]="TCPdaytimed",
argv[1]="8080" */
{
    struct    sockaddr_in fsin;        /* the from address of a client */
    char *service="daytime";           /* service name or port number */
    SOCKET msock, ssock;               /* master & slave sockets */
    int  alen;                         /* from-address length */
    WSADATA wsadata;                   //Socket 的版本信息

    switch(argc) {
    case 1:                            //命令行仅文件名(即命令动词)
        break;
    case 2:                            //命令行仅一个参数
        service=argv[1];
        break;
    default:                           //其他情况
        errexit("usage: TCPdaytimed [port]\n");
    }

    if (WSAStartup(WSVERS, &wsadata) !=0)
//Winsock 服务初始化。首参数指明程序请求使用的 Socket 版本(高位字节指明副版本,低位字
//节指明主版本);次参数返回请求的 Socket 的版本信息
```

```
        errexit("WSAStartup failed\n");

    msock=passiveTCP(service, QLEN);  //创建被动的套接字,调用 passiveTCP 实现。第
//一个是字符串:服务的名字或者端口号;第二个是传入连接的请求队列所需的长度
    while (1) {
        alen=sizeof(struct sockaddr);
        ssock=accept(msock, (struct sockaddr *)&fsin, &alen);
        if (ssock==INVALID_SOCKET)
            errexit("accept failed: error number %d\n",
                GetLastError());
        TCPdaytimed(ssock);
        (void) closesocket(ssock);  //调用 close 是从容关闭:TCP 保证所有的数据可靠交
                                    //付给客户(连接终止前收到确认)
    }
//在循环中,使用 accept 从主套接字得到一个连接(accept 完成三次握手过程)
//对于新的连接服务器调用过程 TCPdaytimed 进行处理
//处理完毕继续循环,再次调用 accept 阻塞
//循环实现就足够了,服务器忙时,其他的请求可以排队
}//main()

/*--------------------------------------------------------------------
* TCPdaytimed -do TCP DAYTIME protocol
*--------------------------------------------------------------------
*/
//从其他机器上获得另一个系统当前的日期和时间
Void  TCPdaytimed(SOCKET fd)
{
    char *pts;                          /* pointer to time string*/
    time_t  now;                        /* current time*/

    (void) time(&now);
    pts=ctime(&now);
    (void) send(fd, pts, strlen(pts), 0);
    printf("%s", pts);
}
```

(2) 客户端程序。

```
/* TCPdtc_client.cpp -main, TCPdaytime */

#include<stdlib.h>
#include<stdio.h>
#include<winsock2.h>

void TCPdaytime(const char *, const char *);
void errexit(const char *, …);
```

```
SOCKET connectTCP(const char * , const char * );

#define  LINELEN        128
#define WSVERS        MAKEWORD(2, 0)

/*------------------------------------------------------------------
 * main - TCP client for DAYTIME service
 *------------------------------------------------------------------
 */
int
main(int argc, char * argv[])          /* usage: TCPdaytime [host [port]]\n */
{                                      /* host 可以为服务器的 IP 地址或域名,port 可
                                          以为服务器监听的端口号或服务名 */
    char * host="localhost";           /* host to use if none supplied */
    char * service="daytime";          /* default service port */
    WSADATA wsadata;

    switch(argc) {
    case 1:
        host="localhost";
        break;
    case 3:
        service=argv[2];               /* FALL THROUGH */
    case 2:
        host=argv[1];
        break;
    default:
        fprintf(stderr, "usage: TCPdaytime [host [port]]\n");
        exit(1);
    }

    if (WSAStartup(WSVERS, &wsadata)!=0)    /* 启动某版本的 DLL */
        errexit("WSAStartup failed\n");
    TCPdaytime(host, service);              /* 建立连接并传输数据 */
    WSACleanup();                           /* 卸载某版本的 DLL */
    printf("按任意键继续…");
    getchar();
    return 0;                               /* exit */
}

/*------------------------------------------------------------------
 * TCPdaytime - invoke Daytime on specified host and print results
 *------------------------------------------------------------------
 */
void
```

```
TCPdaytime(const char * host, const char * service)
{
    char buf[LINELEN+1];                    /* buffer for one line of text */
    SOCKET s;                               /* socket descriptor */
    int  cc;                                /* recv character count */

    s=connectTCP(host, service);            /* 建立连接 */

    cc=recv(s, buf, LINELEN, 0);
    while(cc!=SOCKET_ERROR && cc >0) {
        buf[cc]='\0';                       /* ensure null-termination */
        (void) fputs(buf, stdout);
        cc=recv(s, buf, LINELEN, 0);
    }
    closesocket(s);
}
```

(3) 程序运行。

为了运行程序,须先编译程序:在 Visual C++ 下建立 dsw 文件,然后生成并运行(按 Ctrl+F5 键)。使用套接字函数的程序链接时需要加载 ws2_2.lib,即在工程设置/Link/General/对象/库模块中加入 ws2_2.lib。

运行程序时要先运行服务器程序:在命令窗口中输入 TCPdtc_server server_ip [server_port]\n;在另一台计算机上运行 TCPdtc_client server_ip [server_port]\n。如果程序没有回应,可能是防火墙引起服务器的监听端口关闭。在程序运行时可以启动 Wireshark 捕获数据包,通过分析可以展示 C/S 的连接过程。

【例 3-2】 将网卡的工作模式设置为混杂模式。网卡的混杂模式是网络分析的需要。Linux 已经提供了设置网卡为混杂模式的命令,实际上也可自行编写程序实现。虽然只是一个本地应用,因涉及网络接口,也要使用 socket 编程。

(1) 程序清单。

```
/*-----------------------------------------------------------------
* 程序:makprom
* 功能:设置网卡为混杂模式
*-----------------------------------------------------------------
*/
#include<cstdio.h>
#include<sys/ioctl.h>
#include<cstdlib.h>
#include<sys/socket.h>
#include<cstring.h>
#include<linux/in.h>
#include<linux/if_ether.h>
#include<unistd.h>
#include<net/if.h>
```

```
int main(int argc, char **argv) {
  int sock, n;
  struct ifreq ethreq;  //网络接口结构
  if ((sock=socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)))<0) {
    perror("socket");
    exit(1);
  }
/* Set the network card in promiscuos mode */
  strncpy(ethreq.ifr_name,"eth0",IFNAMSIZ);     //把网络设备的名字填充到 ifr 结构中
  if (ioctl(sock,SIOCGIFFLAGS,&ethreq)==-1) {  //获取接口标志
    perror("ioctl");
    close(sock);
    exit(1);
  }
  ethreq.ifr_flags |=IFF_PROMISC; //获取接口标志后将其设置成混杂模式。用|是因为必须
                                  //在保留原来设置的情况下,在标志位中加入"混杂"模式
  if (ioctl(sock,SIOCSIFFLAGS,&ethreq)==-1) {  //将标志位设置写入
    perror("ioctl");
    close(sock);
    exit(1);
  }
  printf("Success' to set eth0 to promiscuos mode…\n");
  return 0;
}
```

(2) 运行程序。

先编译程序,命令如下:

```
gcc makprom.c -o makprom
```

执行命令:

```
sudo ./makprom
```

可通过 ifconfig 命令查看当前网卡的工作模式是否被设置为 promiscuos。

实验 3-1　TCP 通信程序设计

【实验名称】

基于 TCP 的聊天程序设计。

【实验原理】

在使用 Winsock 控件时,首先需要考虑使用什么协议。可以使用的协议包括 TCP 和 UDP。两种协议之间的重要区别在于它们的连接状态。

TCP 协议是一种有连接协议,可以将其与电话系统相比。在开始数据传输之前,通信双方必须先建立连接。

UDP 协议是一种无连接协议,两台计算机之间的传输类似于传递邮件:消息从一台计算机发送到另一台计算机,但是两者之间没有明确的连接。另外,单次传输的最大数据量取

决于具体的网络环境。

(1) 实现思想。

Internet 上的聊天程序一般都是以服务器提供服务端连接响应。使用者通过客户端程序登录到服务器,就可以与登录在同一服务器上的用户交谈。这是一个面向连接的通信过程。因此程序要在 TCP/IP 环境下实现服务器端和客户端两部分程序。

(2) 服务器端工作流程。

服务器端通过 socket()系统调用创建一个 Socket 数组后(即设定了接受连接客户的最大数目),与指定的本地端口绑定 bind(),就可以在端口进行侦听 listen()。如果有客户端连接请求,则在数组中选择一个空 Socket,将客户端地址赋给该 Socket,然后登录成功的客户就可以在服务器上聊天了。

(3) 客户端工作流程。

客户端程序相对简单,只需要建立一个 Socket 与服务器端连接,成功后即可通过该 Socket 发送和接收数据。

【实验目的】

选择一个操作系统环境(Linux 或者 Windows),编制 TCP/IP 通信程序,完成一定的通信功能。了解 TCP/IP 通信协议原理及 Winsock 控件的使用方法;使用 TCP/IP 协议进行相互直接的发送和接收等。

【实验要求】

(1) 登录功能。客户端登录到聊天服务器,服务器管理所有登录的客户,并将客户列表发送给各个客户显示。

(2) 客户可以通过服务器转发,实现一对一和多对多聊天。

(3) 实现呼叫功能。当客户端程序连接服务器时,通过服务器搜索所要呼叫的客户,如果检测到此用户且该用户正处于联网状态,则服务器通知此用户的客户端程序响应主叫方客户端程序,然后在主叫方和被叫方建立连接后,双方就可以聊天或进行其他的通信。

(4) 客户端程序应该可以实时显示目前其他用户的状态(例如好友信息上、下线)。

(5) 界面友好,能够建立至少 1 个群组,能够加入群实现群聊,要求有留言功能且能进行文件传输。

(6) 实验时,请运行 Wireshark 软件,对通信时的数据包进行跟踪分析。

实验 3-2　UDP 通信程序设计

【实验名称】

基于 UDP 丢包统计程序设计。

【实验目的】

选择一个操作系统环境(Linux 或者 Windows),编制 UDP/IP 通信程序,完成一定的通信功能。

【实验要求】

在发送 UDP 数据包时做一个循环,连续发送 100 个数据包;在接收端统计丢失的数据包。

实验时,请运行 Wireshark 软件,对通信时的数据包进行跟踪分析。

【实验思考】

(1) 说明在实验过程中遇到的问题和解决方法。

(2) 给出程序详细的流程图和对程序关键函数的详细说明。

(3) 使用 Socket API 开发通信程序中的客户端程序和服务器程序时,各需要哪些不同的函数?

(4) 解释 connect()、bind()等函数中 struct sockaddr * addr 参数各个部分的含义,并用具体的数据举例说明。

(5) 说明面向连接的客户端和面向非连接的客户端在建立 Socket 时有什么区别。

(6) 说明面向连接的客户端和面向非连接的客户端在收发数据时有什么区别。面向非连接的客户端又是如何判断数据发送结束的?

(7) 比较面向连接的通信和无连接通信,它们各有什么优点和缺点?适合在哪种场合下使用?

(8) 实验过程中使用 Socket 时是工作在阻塞方式还是非阻塞方式?通过网络检索阐述这两种操作方式的不同。

实验 3-3 网络嗅探器设计

【实验名称】

网络嗅探器设计。

【实验环境】

详细说明运行的操作系统和网络平台。

【实验目的】

用原始套接字捕获所有经过本地网卡的数据包,并从中分析出协议、IP 源地址、IP 目的地址、TCP 源端口号、TCP 目的端口号以及数据包长度等信息。

【实验原理】

嗅探器作为一种网络通信程序,是通过对网卡的编程实现网络通信的,对网卡的编程是使用通常的套接字(Socket)方式进行。但是,通常的套接字程序只能响应与自己硬件地址相匹配的或是以广播形式发出的数据帧,对于其他形式的数据帧(如已到达网络接口却非发给此地址的数据帧),网络接口在验证到投递的地址并非自身地址之后将不引起响应,即应用程序无法收取到达的数据包。而网络嗅探器的目的恰恰在于从网卡接收所有经过它的数据包,这些数据包既可以是发给它的也可以是发往别处的。显然,要达到此目的就不能再让网卡按通常的正常模式工作,而必须将其设置为混杂模式。在混杂模式下,网卡能够接收一切通过它的数据,而不管该数据是否是传给它的。

编程实现时,这种对网卡混杂模式的设置是通过原始套接字实现的,这也有别于通常使用的数据流套接字和数据报套接字。设置混杂模式后就可以开始对网络数据包进行嗅探了,其中对数据包的获取仍像流式套接字或数据报套接字那样通过 recv()函数完成。但是与其他两种套接字不同的是,原始套接字此时捕获到的数据包并不仅仅是单纯的数据信息,而是包含有 IP 首部、TCP 首部等信息包的最原始的数据信息,这些信息保留了它在网络传输时的原貌。通过对这些在低层传输的原始信息的分析可以得到有关网络的一些信息。由于这些数据经过了网络层和传输层的封闭,因此需要根据其附加的帧头对数据包进行分析。

数据包的总体结构如图 3-5 所示。

数据包		
IP首部	TCP首部(或其他信息首部)	数据

图 3-5　数据包的总体结构

数据在从应用层到达传输层时，将添加 TCP 首部或是 UDP 首部。其中 UDP 首部比较简单，由一个 8B 的首部和数据部分组成，具体格式如图 3-6 所示。

16位	16位
源端口	目的端口
UDP长度	UDP校验和

图 3-6　UDP 数据包段头结构

而 TCP 首部则比较复杂，以 20 个固定字节开始，在固定首部后面还可以有一些长度不固定的可选项，图 3-7 给出 TCP 首部的格式组成。

<table>
<tr><td colspan="8">16位</td><td>16位</td></tr>
<tr><td colspan="8">源端口</td><td>目的端口</td></tr>
<tr><td colspan="9">顺序号</td></tr>
<tr><td colspan="9">确认号</td></tr>
<tr><td>TCP首部长度</td><td>(保留)7位</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td>窗口大小</td></tr>
<tr><td colspan="8">校验和</td><td>紧急指针</td></tr>
<tr><td colspan="9">可选项(0或更多的32位字)</td></tr>
<tr><td colspan="9">数据(可选项)</td></tr>
</table>

图 3-7　TCP 数据段头结构

在网络层，还要给 TCP 数据包添加一个 IP 首部以组成 IP 数据包。IP 首部以大端点机次序传送，从左到右，版本字段的高位字节优先传输(SPARC 是大端点机；Pentium 是小端点机)。如果是小端点机，就要在发送和接收时先行转换然后才能进行传输。IP 数据段头格式如图 3-8 所示。

<table>
<tr><td colspan="4">16位</td><td colspan="4">16位</td></tr>
<tr><td>版本</td><td>IHL</td><td colspan="2">服务类型</td><td colspan="4">总长</td></tr>
<tr><td colspan="4">标识</td><td colspan="3">标志</td><td>分段偏移</td></tr>
<tr><td colspan="3">生存时间</td><td>协议</td><td colspan="4">头校验和</td></tr>
<tr><td colspan="8">源地址</td></tr>
<tr><td colspan="8">目的地址</td></tr>
<tr><td colspan="8">选项(0或更多)</td></tr>
</table>

图 3-8　IP 数据段报头结构

在明确了以上几个数据段头的组成结构后，就可以对捕获到的数据包进行分析了。

【实验内容及步骤】

包括主要程序流程和函数说明,程序分工说明。

【实验结果】

请自行填写。

【实验中的问题及心得】

请自行填写。

【实验思考】

在本实验的基础上,实现一个 Sniffer 程序捕捉用户名和密码,若捕获到密码则在屏幕上面显示,同时输出源计算机和目的计算机的 IP 地址,遇到其他信息则进行简单的抛弃而不做任何处理。

实验 3-4　停等协议通信

【实验名称】

利用停止等待协议传输数据文件。

【实验环境】

详细说明运行的操作系统、网络平台、机器的 IP 地址。

【实验目的】

深入理解停止等待协议的主要特点和工作过程。

【实验原理】

停等 ARQ 是最简单且最基本的数据链路层协议。假定传送的数据不会产生差错,但接收方接收数据的速率不一定能够跟上发送方发送数据的速率。为使接收方的接收缓冲区在任何情况下都不会溢出,最简单的情况就是发送方每发送一帧就暂停下来,接收方收到数据帧后就交给主机,然后发送一个确认信息给对方,表示接收的任务已完成。这时,发送方才发送下一个数据帧。在这种情况下,接收方的接收缓冲区的大小只要能够装下一个数据帧即可。显然,用这样的方法收、发双方可以同步得很好。由接收方控制发送方的数据流量是计算机网络中流量控制的一个重要方法。

假定数据帧在主机 A 与主机 B 的传输过程中出现了差错。由于通常都在数据帧中加上了循环冗余校验 CRC,所以主机 B 很容易检验出收到的数据帧是否会有差错。当发现差错时,主机 B 就向主机 A 发送一个否认帧 NAK,以表示主机 A 应当重发出现差错的数据帧。如多次出现差错,就要多次重发数据帧,直至收到主机 B 发来的确认帧 ACK 为止。为此,在发送端必须暂时保存已发送过的数据帧的副本。当通信质量太差时,主机 A 在重发一定的次数后即不再进行重发,而是将此情况向上一层报告。

主机 B 收不到主机 A 发来的数据帧,这种情况称为帧丢失。发送帧丢失时主机 B 不会向主机 A 发送任何应答帧。如果主机 A 要等到主机 B 的应答信息后再发送下一个数据帧,那么将永远等待下去,于是就出现了死锁现象。同理,若主机 B 发过来的应答帧丢失,也会出现这种死锁现象。要解决死锁问题,可在主机 A 发送完一个数据帧时就启动一个超时定时器。若在超时定时器所设置的定时时间 t 结束后仍收不到主机 B 的任何应答帧,则主机 A 就重传前面所发送的这一数据帧。超时定时器设置值时,若定时时间选得太短,则还没有收到应答帧就重发了数据帧;若定时时间选得太长,则要白白浪费许多时间。一般可将定

时时间选为略大于从发完数据帧到收到应答帧所需的平均时间。

如果丢失的是应答帧，超时重发将使主机 B 收到两个相同的数据帧。由于主机 B 无法识别重复的数据帧，因而在主机 B 收到的数据中出现了另一种差错：重复帧。要解决重复帧的问题，必须使每个数据帧带上不同的发送序号。每发送一个新的数据帧就把它的发送序号加 1。若主机 B 收到发送序号相同的数据帧，就表明出现了重复帧。主机 B 应当丢弃此重复帧，并向主机 A 发送一个确认帧 ACK，因为主机 B 已经知道主机 A 还没有收到上一次发过去的确认帧 ACK(有可能此确认帧在传输过程中出错)。

【实验内容及步骤】

包括主要程序流程和函数说明，程序分工说明。

【实验结果】

请自行填写。

【实验中的问题及心得】

请自行填写。

实验 3-5　GBN 协议编程

【实验原理】

GBN 即 Go Back N protocol(或称连续 ARQ、Go-BACK-N ARQ、退回 N)，意思是当出现差错必须重传时，要向回走 N 个帧，然后再开始重传。

当接收方检测出失序的信息帧后，要求发送方重发最后一个正确接收的信息帧之后的所有未被确认的帧；或者当发送方发送了 N 个帧后，若发现该 N 帧的前一个帧在计时器超时后仍未返回其确认信息，则该帧被判为出错或丢失，此时发送方就不得不重新发送出错帧及其后的 N 帧。对接收方而言，由于这一帧出错，就不能以正常的序号向它的高层递交数据，对其后发送来的 N 帧也可能都不能接收而丢弃。GO-BACK-N 操作过程如图 3-9 所示。图 3-9 中假定发送完 8 号帧后，发现 2 号帧的确认返回信号在计时器超时后还未收到，则发送方只能退回到 2 号帧开始重发之后所有已发的数据。

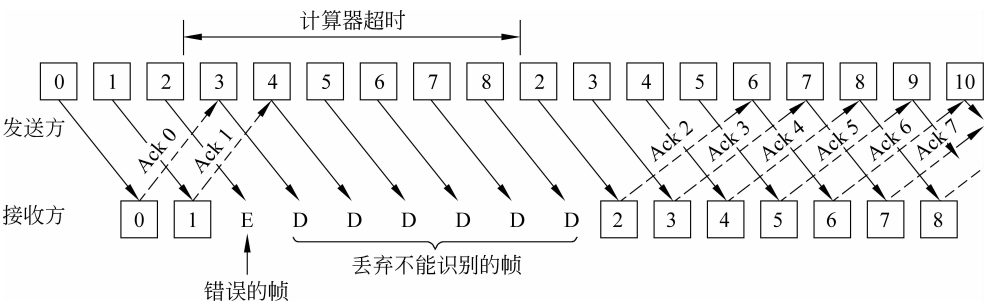

图 3-9　GBN 工作原理

还有另一种情况：2 号数据帧丢失，3～5 号数据帧虽然正确传送到节点 B，但也不得不被丢弃。在节点 A 发送 5 号数据帧的过程中，超时定时器设定的超时时间到。因此，在 5 号数据帧发送完毕后，就回到 2 号数据帧进行重传。

【实验要求】

(1) 体现链路层和网络层的控制流作用。例如在网络层出现突发数据时,链路层窗口会填满;网络层流量稀少时,链路层窗口数据较少。

(2) 已发送未确认的 frame,允许发送 frame,在发送窗口中用不同的颜色表示。

(3) 要求体现发送方发送 frame,frame 在信道上传播,frame 到达接收方,接收方发送 ACK,接收方滑动窗口。

(4) 要求体现发送方接收 ACK,发送方滑动窗口。

(5) 要求体现 frame 丢失或者损坏的情况。

(6) 要求体现 ACK 丢失或者损坏的情况。

(7) 要求可以配置 rtt 时间,即从发送 frame x 出去到接收到 frame x 的确认的时间,要有默认值。

(8) 要求可以调整发送方超时的时间,要有默认值。

(9) 要求可以配置发送方窗口大小和序列号大小,能够体现使用全部序列号空间时产生的问题,能够体现避免该问题的配置。

(10) 界面直观,简单易懂,演示过程中可以随时停止。

【实验内容及步骤】

包括主要程序流程和函数说明,程序分工说明。

【实验结果】

请自行填写。

【实验中的问题及心得】

请自行填写。

实验 3-6 IPv4 组播通信

【实验原理】

(1) 组播地址。

IP 组播技术是一种允许一台或多台主机(组播源)发送单一数据包到多台主机的 TCP/IP 网络技术。当需要将一个节点的信号传送到多个节点时,无论是采用重复点对点通信方式还是采用广播方式,都会严重浪费网络带宽,因此组播技术应运而生。组播能使一个或多个组播源只把数据包发送给特定的组播组,而只有加入该组播组的主机才能接收到数据包。

使用同一个 IP 组播地址接收组播数据包的所有主机构成了一个主机组,也称为组播组。一个组播组的成员是随时变动的,一台主机可以随时加入或离开组播组,组播组成员的数目和所在的地理位置也不受限制,一台主机也可以属于几个组播组。此外,不属于某一个组播组的主机也可以向该组播组发送数据包。

IP 组播通信必须依赖于组播地址,组播地址都是一些特殊的 IP 地址,组播地址的最高 4 位都是 1110,它的范围是 11100000～11101111,即 224～239。因此组播地址的范围为 224.0.0.0～239.255.255.255,属于 IP 分类中的 D 类地址,并被划分为局部链接组播地址、预留组播地址和管理权限组播地址三类。

在这些地址中,并不是所有地址都是用户可用的,它们中有一些属于保留地址。

① 局部链接组播地址,范围是 224.0.0.0～224.0.0.255,这部分地址一般是分配给特

定协议作为它的组播信息的分发。例如 224.0.0.1 是组播中所有成员，224.0.0.2 是组播中所有路由器，224.0.0.9 是 RIPv2 组播更新的地址，而 224.0.0.5 和 224.0.0.6 用作 OSPF 状态更新，224.0.0.10 是思科私有协议 EIGRP 用于作为组播路由信息更新的地址，224.0.0.13 是表示所有 PIM(Protocol Independent Multicast，协议无关组播)的路由器。

② 预留组播地址，范围是 224.0.1.0～238.255.255.255，可用于全球范围(如 Internet)或网络协议。

③ 本地管理组地址，范围是 239.0.0.0～239.255.255.255，这个地址范围用于作为私人组播领域的管理权限地址，类似于私有 IP 地址，不能用于 Internet，可限制组播范围。

上述都是由第三层地址表示的，但是到了数据链路层还需要第二层地址进行传输。因此组播寻址有专门的物理地址用于组播，对以太网而言，开头是 01-00-5E。

(2) IP 组播地址与以太网硬件组播地址的映射方法。

以太网硬件地址是 48 位，而 IP 地址是 32 位，有效 IP 组播地址是 28 位，以太网支持 IP 组播地址到以太网组播地址的映射，它们之间的映射主要规则如下。

将 IP 组播地址的低 23 位简单地代替特定的以太网地址 01.00.5e.00.00.00(十六进制)中的低 23 位，然后把这 23 位前面的一位置 0，MAC 地址的前 24 位必须为 01-00-5E。

例如：IP 组播地址 224.205.155.110 对应的二进制为 11100000.11001101.10011011.01101110，这一串二进制数的后 23 位的十六进制是 4D 9B 6E，组播 MAC 标识为 01-00-5E，于是最终映射到以太网的地址为 01-00-5E-4D-9B-6E。

按此规则，IP 组播地址范围为 224.0.0.0～239.255.255.255，映射到以太网组播地址为 01.00.5E.00.00.00～01.00.5E.7F.FF.FF。

这种映射地址的方法有三个优点：第一，方法简单，便于计算和实现；第二，可以包括绝大部分组播地址；第三，IP 组播地址映射后仅使用以太网地址的固定部分，有利于排错和查找，不易与其他使用以太网的协议发生冲突和干扰。

在 IP 组播地址映射到以太网地址的过程中，有可能出现多个不同的 IP 组播地址映射到了相同的以太网地址。例如，IP 组播地址 225.118.100.100 和 226.246.100.100 映射到以太网的地址均为 01.00.94.118.100.100，这主要是由于 IP 组播地址的有效位为 28 位，而映射到以太网时仅取低 23 位(第 9～31 位)，高 5 位(实际是 IP 组播地址的第 4～8 位)的地址信息在映射过程中实际上没有使用且被丢掉了，这样如果低 23 位地址信息一样，则不管高 5 位地址的值是多少，其映射的以太网地址都是一样的，如图 3-10 所示。

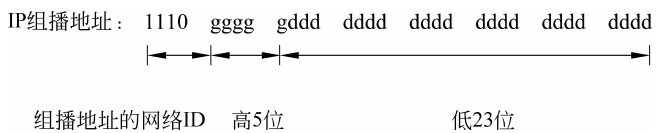

图 3-10 IP 组播地址映射到以太网地址

高 9 位映射时信息均被丢掉，映射时直接代替特定以太网地址的低 23 位。但实际上，任意两个 IP 组播地址映射到以太网地址时，其地址相同的概率很低。

(3) 组播的工作过程。

在局域网内，源主机的网络接口将到目的主机的数据包发送到高层，这些数据包中的目的地址是物理接口地址或广播地址。如果主机已经加入到一个组播组中，主机的网络接口

就会识别出发送到该组成员的数据包。因此，如果主机接口的物理地址为 80-C0-F6-A0-4A-B1，其加入的组播组为 224.0.1.10，则发送给主机的数据包中的目的地址一定是下面三种类型之一。

接口地址：80-C0-F6-A0-4A-B1

广播地址：FF-FF-FF-FF-FF-FF

组播地址：01-00-5E-00-01-0A

广域网中，路由器必须支持组播路由。当主机中运行的进程加入到某个组播组中时，主机向子网中的所有组播路由器发送 IGMP(Internet 分组管理协议)报文，告诉路由器凡是发送到该组播组的组播报文都必须发送到本地的子网中，这样主机的进程就可以接收到报文了。子网中的路由器再通知其他的路由器，这些路由器就知道该将组播报文转发到哪些子网中去。子网中的路由器也向 224.0.0.1 发送一个 IGMP 报文(224.0.0.1 代表组中的全部主机)，要求组中的主机提供组的相关信息。组中的主机收到这个报文后，都各自将计数器的值设为随机值，当计数器递减为 0 时再向路由器发送应答。这样就防止了组中所有的主机同时向路由器发送应答而造成网络拥塞。主机向组播地址发送一个报文作为对路由器的应答，组中的其他主机一旦收到这个应答报文，就不再发送应答报文了。因为组中的主机向路由器提供的都是相同的信息，所以子网路由器只需得到组中一个主机提供的信息即可。

如果组中的主机都退出了，路由器就收不到应答，因此路由器认为该组目前没有主机加入，于是停止到该子网报文的路由。IGMPv2 的解决方案是：组中的主机在退出时向 224.0.0.2 发送报文通知组播路由器。

(4) 应用编程接口(API)。

对组播选项所进行的操作只需 5 个新的套接字操作。函数 setsockopt()及 getsockopt()用于建立和读取这 5 个选项的值。表 3-2 中列出了组播的可选项，并列出其数据类型和描述。

表 3-2　组播可选项列表

IPv4 选项	数据类型	描　　述
IP_ADD_MEMBERSHIP	struct ip_mreq	加入到组播组中，用 setsockopt()函数发送该选项。该选项类型是 ip_mreq 结构，它的第一个字段 imr_multiaddr 指定了组播组的地址，第二个字段 imr_interface 指定了接口的 IPv4 地址
IP_ROP_MEMBERSHIP	struct ip_mreq	从组播组中退出，数据结构 ip_mreq 的使用方法与上面相同
IP_MULTICAST_IF	struct ip_mreq	指定提交组播报文的接口，该选项可以修改网络接口，在结构 ip_mreq 中定义新的接口
IP_MULTICAST_TTL	u_char	设置组播报文的数据包的 TTL(生存时间)。默认值是 1，表示数据包只能在本地的子网中传送
IP_MULTICAST_LOOP	u_char	使组播报文环路有效或无效。组播组中的成员自己也会收到它向本组发送的报文。该选项用于选择是否激活这种状态