实验二十七 SOCKET 网络程序设计

【实验目的】

1、理解进程通信的原理及通信过程；

2、掌握基本的网络编程方法。

【实验学时】

4 学时

【实验内容】

1、进一步学习 UDP 及 TCP 协议的工作原理；

2、学习 SOCKET 编程的基本方法；

3、学习应用 C 语言与 WinSock2 进行简单的无连接的网络程序设计，实现网络数据传输；

4、学习应用 C 语言与 WinSock2 进行简单的面向连接的网络程序设计，实现网络数据传输。

【实验原理】

1、关于使用套接字编程的一些基本概念

- 半相关

网络中用一个三元组可以在全局唯一标示一个进程：（协议，本地地址，本地端口号）。这样一个三元组，叫做一个半相关（half-association），它指定连接的每半部分。

- 全相关

一个完整的网间进程通信需要由两个进程组成，并且只能使用同一种高层协议。也就是说，不可能通信的一端用 TCP 协议，而另一端用 UDP 协议。因此一个完整的网间通信需要一个五元组来标识：协议，本地地址，本地端口号，远地地址，远地端口号。这样一个五元组，叫做一个全相关（association），即两个协议相同的半相关才能组合成一个全相关。

- TCP/IP 协议的地址结构为：

```
struct sockaddr_in
{
short sin_family;       /*协议的地址族，IP 协议是 AF_INET*/
u_short sin_port;       /*16 位端口号，网络字节顺序*/
struct in addr sin addr;       /*32 位 IP 地址，网络字节顺序*/
char sin_zero[8];       /*填充*/
}
```

- 套接字类型

TCP/IP 的 socket 提供下列二种类型套接字：

①、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、无重复地发送，且按发送顺序接收。内设流量控制，避免数据流超限；数据被看作是字节流，无长度限制。文件传送协议（FTP ）即使用流式套接字。

②、数据报套接字（SOCK_DGRAM）

提供了一个无连接服务。数据包以独立包形式被发送，不提供无错保证，数据可能丢失或重复，并且接收顺序混乱。网络文件系统（NFS）使用数据报式套接字。

③原始套接字（SOCK_RAW）

该接口允许对较低层协议，如 IP、ICMP 直接访问。常用于检验新的协议实现或访问现有服务中配置的新设备。

- 基本套接字系统调用

为了更好地说明套接字编程原理，下面给出几个基本套接字系统调用说明：

①、创建套接字——Socket()

应用程序在使用套接字前，首先必须拥有一个套接字，系统调用 socket()向应用程序提供创建套接字的手段，其调用格式如下：

SOCKET socket(int af,int type,int protoCol);

该调用要接收三个参数：af、type、protocol 参数 af 指定通信发生的区域，UNIX 系统支持的地址族有：AF UNIX、AF INET、AF NS 等，而 DOS、WINDOWS 中仅支持 AF_INET，因此，地址族与协议族相同。参数 type 描述要建立的套接字的类型。参数 protocol 说明该套接字使用的特定协议，如果调用者不希望特别指定使用的协议，则置为 0，使用默认的连接模式。根据这 几个参数建立一个套接字，并将相应的资源分配给它，同时返回一个整型套接字号。因此，socket()系统调用实际上指定了相关五元组中的“协议”这一元。

②、指定本地地址——bind()

当一个套接字用 socket()创建后，存在一个名字空间（地址族），但它没有被命名。bind()将套接字地址（包括本地主机地址与本地端口地址）与所创建的套接字号联系起来，即将名字赋予套接字，以指定本地半相关。其调用格式如下：

int bind(SOCKET s,const struct sockaddr FAR*name,int namelen);

参数 s 是由 socket()调用返回的并且未作连接的套接字描述符(套接字号)。参数 name 是赋给套接字 s 的本地地址(名字)，其长度可变，结构随通信域的不同而不同。 namelen 表明了 name 的长度。如果没有错误发生，bind()返回 0。否则返回值 SOCKET ERROR。地址在建立套接字通信过程中起着重要作用，作为一个网络应用程序设计者对套接字地址结构必须有明确认识。

③、建立套接字连接——connect()与 accept()

这两个系统调用用于完成一个完整相关的建立，其 connect()用于建立连接。无连接的套接字进程也可以调用 connect()，但这时在进程之间没有实际的报文交换，调用将从本地操作系统直接返回。这样做的优点是程序员不必为每一数据指定目的地址，而且如果收到的一个数据报，其目的端口未与任何套接字建立“连接”，便能判断该端口不可操作。而 accept()用于使服务器等待来自某客户进程的实际连接。

connect()的调用格式如下：

int connect(SOCKET s,const struct sockaddr FAR*name,int namelen);

参数 s 是欲建立连接的本地套接字描述符。参数 name 指出说明对方套接字地址结构的

指针。对方套按字地址长度由 namelen 说明。

如果没有错误发生，connect()返回 0。否则返回值 SOCKET_ERROR。在面向连接的协议中，该调用导致本地系统与外部系统之间连接实际建立。

由于地址族总被包含在套接字地址结构的前两个字节中，并通过 Socket()调用与某个协议族相关。因此 bind()和 connect()无须协议作为参数。

accept()的调用格式如下：

SOCKET accept(SOCKET s,struct sockaddr FAR*addr,int FAR*addrlen);

参数 s 为本地套接字描述符，在用做 accept()调用的参数前应该先调用过 listen()。addr 指向客户方套接字地址结构的指针，用来接收连接实体的地址。 addr 的确切格式由套接字创建时建立的地址族决定。addrlen 为客户方套接字地址的长度(字节数)。如果没有错误发生，accept()返回一个 SOCKET 类型的值，表示接收到的套接字的描述符。否则返回值 INVALID_SOCKET。

accept()用于面向迎接服务器。参数 addr 和 addrlen 存放客户方的地址信息。调用前，参数 addr 指向一个初始值为主的地址结构，而 addrlen 的初始值为 0；调用 accept()后，服务器等待从编号为 s 的套接字上接受客户连接请求，而连接请求是由客户方的 connect()调用发出的。当有连接请求到达时，accept()调用将请求连接队列上的第一个客户方套接字地址及长度放入 addr 和 addrlen，并创建一个与 s 有相同特性的新套接字号。新的套接字可用于处理服务器开发请求。

四个套接字系统调用，socket()、bind()、connect()、accept()，可以完成一个完全五元相关的建立。socket()指定五元组中的协议元，它的用法与是否为客户或服务器、是否面向连接无关。bind()指定五元组中的本地二元，即本地主机地址和端口号，其用法与是否面向连接有关：在服务器方，无论是否面向连接，均要调用 bind()；在客户方，若采用面向连接，则可以不调用 bind()，而通过 connect()自动完成。若采用无连接，客户方必须使用 bind()以获得一个唯一的地址。

以上讨论仅对客户/服务器模式而言，实际上套接字的使用是非常灵活的，唯一需遵循的原则是进程通信之前，必须建立完整的相关。

④、监听连接——listen()

此调用用于面向连接服务器，表明它愿意接收连接。listen()需在 accept()之前调用，其调用格式如—下：

int listen(SOCKET s,int backlog):参数 s 标识一个本地已建立、尚未连接的套接字号，服务器愿意从它上面接收请求。

backlog 表示请求连接队列的最大长度，用于限制排队请求的个数，目前允许的最大值为 5。

如果没有错误发生，listen()返回 0。否则它返回 SOCKET_ERROR。

listen()在执行调用过程中可为没有调用过 bind()的套接字 s 完成所必须的连接，并建立长度为 backlog 的请求迎接队列。

调用 listen()是服务器接收一个连接请求的四个步骤中的第二步。它在调用 socket()分配一个流套接字，且调用 bind()给 s 赋于一个名字之后调用，而且一定要在 accept()之前调用。

⑤、数据传输——send()与 recv()

当一个连接建立以后，就可以传输数据了。常用的系统调用有 send()和 recv()。send()调用用于在参数 s 指定的已连接的数据报或流套接字上发送输出数据，格式如下：

int send(SOCKET s,const char FAR*buf,int len,int flags):

参数 s 为已连接的本地套接字描述符。buf 指向存有发送数据的缓冲区的指针，其长度由 len 指定 oflags 指定传输控制方式，如是否发送带外数据等。如果没有错误发生，send()返回总共发送的字节数。否则它返回 SOCKET_ERROR。

recv()调用用于在参数 s 指定的已连接的数据报或流套接字上接收输入数据，格式如下：

int recv(SOCKET s,char FAR*buf,iht len,int flags):参数 s 为已连接的套接字描述符。buf 指向接收输入数据缓冲区的指针，其长度由 len 指定。flags 指定传输控制方式，如是否接收带外数据等。如果没有错误发生，recv，()返回总共接收的字节数。如果连接被关闭，返回 0。否则它返回 SOCKET ERROR。

⑥、输入/输出多路复用——select()

select()调用用来检测一个或多个套接字的状态。对每一个套接字来说，这个调用可以请求读、写或错误状态方面的信息。请求给定状态的套接字集合由一个 fd_set 结构指示。

在返回时，此结构被更新，以反映那些满足特定条件的套接字的子集，同时，select()调用返回满足条件的套接字的数目，其调用格式如下

int select(int nfds,fd set FAR*readfds,fd set FAR*writefds,fd_set FAR:k

exceptfds,const struct timeval FAR*timeout):参数 nfds 指明被检查的套接字描述符的值域，此变量一般被忽略。参数 readfds 指向要做读检测的套接字描述符集合的指针，调用者希望从中读取数据。参数 writefds 指向要做写检测的套接字描述符集合的指针。exceptfds 指向要检测是否出错的套接字描述符集合的指针。timeout 指向 select()函数等待的最大时间，如果设为 NULL 则为阻塞操作。select()返回包含在 fd set 结构中已准备好的套接字描述符的总数目，或者是发生错误则返回 SOCKET_ERROR。

⑦、关闭套接字——closesocket()

closesocket()关闭套按字 s，并释放分配给该套接字的资源：如果 s 涉及一个打开的 TCP 连接，则该连接被释放。closesocket()的调用格式如下：

BOOL closesocket(SOCKET S)；参数 s 待关闭的套接字描述符。如果没有错误发生，closesocket()返回 0。否则返回值 SOCKET ERROR。

2、用于无连接协议（如 UDP）的 SOCKET 系统调用流程框图

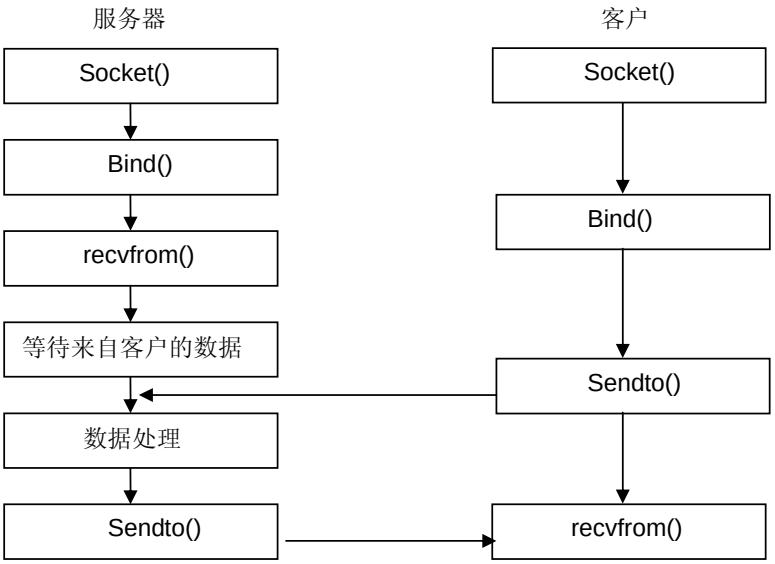

图 9-21 用于无连接的 Socket 系统调用流程图

3、用于面向连接协议（如 TCP）的 SOCKET 系统调用流程框图

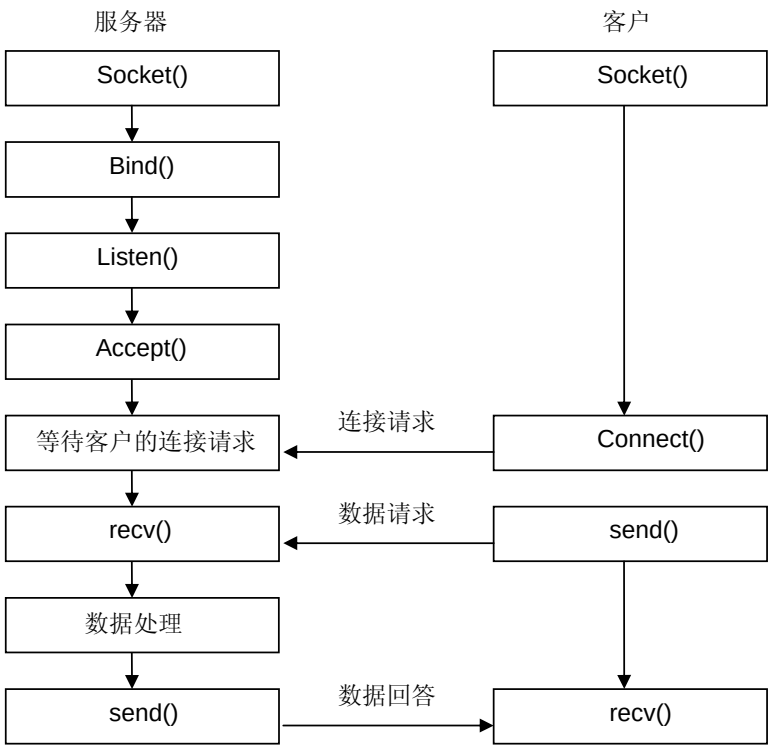

图 9-22 用于面向连接的 Socket 系统调用流程图

【实验步骤】

步骤一：使用 UDP 协议的无连接客户-服务器程序设计

根据实验原理中介绍的内容，设计一个无连接的客户-服务器系统，实现二者之间的数据传递。下面是一个简单的 UDP 客户-服务器程序的实例，作为参考。

说明：下述服务器程序已经在服务器端运行，在协议数据发生器端仅需编写相应的客户程序并运行。

服务器端程序：

```
#include<stdio.h>
#include<winsock.h>
#pragma comment(1ib,"wS2_32.1ib")
#define   SERV UDP PORT 6000      /*服务器进程端口号,视具体情况而定*/
#define   SERV_HOST_ADDR"10.60.46.58"      /*服务器地址,视具体情况而定*/
/*宏定义用来打印错误消息*/
#define PRINTERROR(s)
fprintf(stderr,"\n%:%d\n",s,WSAGetLastError())
/*数据报通信的服务器端子程序*/
void DatagramServer(ShOrt nPort)
{
SOCKET theSocket;
/*创建一个数据报类型的 socket*/
theSocket=socket(AF_INET,      //地址族
SOCK DGRAM,   //socket 类型
IPPROTO UDP);//协议类型:UDP
/*错误处理*/
if(theSoCket==INVALID_SOCKET)
{
PRINTERROR("socket()"):
return;
}
/*填写服务器地址结构*/
SOCKADDR_IN saServer;
saServer.Sin_family=AF_INET;
saServer,sin addr.s addr=INADDR ANY;//由 WinSock 指定地址
saServer.sin=(nport);   //服务器进程端口号
/*将服务器地址与已创建的 socket 绑定*/
int   nRet:

nRet=bind(theSocket,      //Socket 描述符
(LPSOCKADDR)&saServer,   //服务器地址
sizeof(struct SOCkaddt)//地址长度
);
```

```
/*错误处理*/
if(neet==SOCKET_ERROR)
{
PRINTERROR("bind()"):
Closesocket(theSocket);
return;
}
/*等待来自客户端的数据*/
SOCKADDR_IN saClient;
char szBUf[1024];
int nLen=sizeof(saclient);
while(1)
{
/*准备接收数据*/
memset(szBuf,0,sizeof(szBuf));
nRet=recvfrom(theSocket,//已绑定定的 socket
szBuf,//接收缓冲区
sizeof(szBuf), //缓冲区大小
0, //Flags
(LPSOCKADDR)&saclient,   //接收客户端地址的缓冲区
&nLen);      //地址缓冲区的长度
if(nRet>0)
{
/*打印接收到的信息*/
printf("\nData received:%s",szBuf);
/*发送数据给客户端*/
strcpy(szBuf,"From the Server");
sendto(theSocket,      //已绑定的 socket
szBuf,//发送缓冲区
strlen(szBuf),      //发送数据的长度
0,      //Flags
(LPSOCKADDR)&saclient, nLen): //目的地址，地址长度
closesocket(theSocket);
return;
}
/*数据报服务器端主程序*/
void main()
{
WORD wVersionRequested=MAKEWORD(1,1):
WSADATA wsaData;
int nRet;
short nPort;
nPort=SERV_UDP_PORT;
/*初始化 Winsock*/
```

```
nRet=WSAStartUp(wVersionRequested,&wsaData);
if(wsaData.wVersion!=wVersionRequested)
{
fprintf(Stderr,"/n Wrong version\n");
return;
}
/*调用数据服务器子程序*/
DatagramServer(nPort);
/*结束 WinSock*/
WSACleanup();
}
```

上述服务器程序已经在服务器端运行，请学生认真阅读分析后，然后根据实验原理二中介绍的内容，在协议数据发生器端编写相应的无连接的客户端程序并运行。从而实现客户和服务器间的数据传输。在协议数据发生器端一端运行客户端进程，在网络协议分析仪端捕获数据并进行分析。

步骤二：使用 TCP 协议的面向连接的客户-服务器程序设计

根据实验原理中介绍的内容，设计一个面向连接的客户-服务器系统，实现二者之间的数据传递。下面是一个简单的 TCP 客户-服务器程序的服务器程序：

面向连接的服务器程序：

```
ginclude<PROCESS.H>
ginclude<windows.h>
ginclude<winsock.h>
ginclude<sys/types.h>

#include<fcntl.h>
#include<wsipx,h>
#include<wsnwlink.h>
#include<stdio.h>
#defineSERV TCP PORT 6000       /*服务器进程端口号,视具体情况而定*/
#define SERV HOST ADDR"10.60.46.40"       /*服务器 IP,视具体情况而定*/
int sockfd;
#pragma comment(lib,"wS2_32.lib")
```

//线程用来处理客户端的请求，服务器主进程每与某客户端建立一个连接之后，便启动一个新的线程来处理接下客户端的请求，参数为服务器与该客户端的连接点:socket//

```
DWORD ClientThread(void*pVoid)
{
int nRet;
char szBUf[1024];
memset(szBuf,0,   sizeof(szBuf));
/*接收来自客户端的数据信息*/
nRet=recv((SOCKET)pVoid,      //与客户端连接的 socket
szBuf,      //接收缓冲区
sizeof(szBuf),      //缓冲区长度
```

```
0);      //Flags
/*错误处理*/
if(nRet==INVALID SOCKET)
{
printf("recv()");
closesocket(sockfd);
closesocket((SOCKET)pVoid);
return 0;
}
/*显示接收到的数据*/
printf("\nData received:%s\n",SzBuf);
/*发送数据给客户端*/
strcpy(szBuf,"From the Server");      //发送内容
nRet=send((SOCKET)pVoid,      //与客户端连接的 socket
szBuf,      //数据缓冲区
strlen(szBuf),      //数据长度

0);     //Flags
/*结束连接,释放 socket*/
closesocket((SOCKET)pVOid);
return 0;
}
```

//服务器主程序:在一个众所周知的断端口上等待客户的连接请求，有请求到来时建立与客户端的连接，并启动一个线程处理该请求//

```
int main()
{
int    clilen;
int pHandle=-1;
struct sockaddr_in    serv_addr:
SOCKET socketClient;
DWORD    ThreadAddr:
HANDLE dwClientThread;
SOCKADDR_IN SockAddr;
/*初始化 Winsock API,即连接 Winsock 库*/
WORD wVersionRequested=MAKEWORD(1,1);
WSADATA wsaData;
if(WSAStartup(wVersionRequested,&wsaData)){
printf("WSAStartup failed%s\n",WSAGetLasterror());
return-l;
}
/*打开一个 TCP SOCKET*/
if((sockfd=socket(AF INET,SOCK_STREAM,0))<0)
printf("server:can't open stream sockef\n");
/*绑定本地地址,以便客户端连接*/
```

```
memset((char*)&serv addr,0,sizeof(struct sockaddr_in));
serv_addr.Sin_family=AF_INET;
serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
serv_addr.sin_port=htons(SERV_TCP_PORT);
if(bind(sockfd.(struct sockaddr*)&serv_addr,sizeof(serv_addr))<0)
printf("server:can't bind local address"):
/*设置服务器的最大连接数为 15*/
listen(sockfd,5);

/*循环等待来自客户端的连接请求*/
while(1)
{
/*阻塞等待一个请求的到来*/
clilen=sizeof(SOCKADDR IN);
socketClient=accept(sockfd,
(LPSOCKADDR)&SockAddr,
&clilen);
/*出错处理*/
if   (socketClient==INVALID   SOCKET)
{
printf("accept failed!\n");
break:
}
/*打印已建立的连接信息*/
printf("Connection accepted on socket:%d from:%s\n",
socketClient,
inet ntoa(SockAddr.sin_addr));
/*启动一个新线程处理该请求*/
dwClientThread   =CreateThread(NULL,
0,
(LPTHRAD START ROUTINE)&ClientThread,
(void   *)socketClient,
0,
&ThreadAddr);
/*错误处理*/
if(!dwClientThread)
printf("Cannot start client thread...");
/*线程建立以后,主程序里不再使用线程 handle,将其关闭,但线程继续运行*/
CloseHandle((HANDLE)dwClientThread);
}
/*结束  windows sockets API*/
WSACleanUp();
return   0;
}
```

上述服务器程序已经在服务器端运行，请学生认真阅读，然后根据实验原理二中介绍的内容，设计面向连接的客户端程序，实现客户与服务器间的数据传输。在协议数据发生器一端运行客户端进程，在网络协议分析仪端捕获数据并分析。

步骤三：使用 TCP 协议进行复杂的客户-服务器程序设计

上述的例子程序比较简单，有一些进程中往往同时存在几条连接，这样的进程在有报文到来时，可以往它处理的任何 socket 上执行 recv 调用，但它不知道哪个 socket 上已有报文，哪个上没有，可以使用 select()系统调用来解决这样的问题。有兴趣的同学可自行编写这样较复杂的客户-服务器程序。

【思考问题】

结合实验过程中的实验结果，问答下列问题：

l、根据编程练习实验中记录的客户和服务器程序的端口号并结合程序，说明：在客户/服务器模型当中，客户进程的端口号和服务器进程的端口号都是由程序给出说明的吗？为什么？

2、在 TCP/IP 网络中，当客户与服务器进程建立了一条 TCP 连接以后，是否属于该连接的所有包都是经过同一路径（即一条虚电路）传递的？为什么？