Socket编程介绍

Socket

Socket网络编程

- 按照操作系统
  - Windows的socket编程
  - *nix的socket编程
- 按照编程语言
  - 使用C++、Java的socket编程
  - 使用脚本语言的socket编程
- ……

Socket的一些历史

- Sockets本来是UNIX操作系统下流行的一种网络编程接口（API），在4.2 BSD中被首先引入的，被称为“Berkeley Socket API”

- Windows网络应用程序编程接口Windows Sockets API就是在1991年根据4.3 BSD操作系统的“Berkeley Socket API”制定的

- Windows sockets，为windows环境下使用的一套网络编程机制（或规范），简称为Winsock。在Windows操作系统下得到广泛应用的、开放的、支持多种协议的网络编程接口

- 目前版本Winsock 2，由动态链接库WSOCK32.DLL提供支持

- Windows环境下网络编程事实上的标准

- 在Winsock规范中把Winsock API函数集分为与BSD Socket（用在UNIX中）相兼容的基本函数、网络数据信息检索函数和Windows专用扩展函数三类

- 可以看出，Winsock来源于BSD Socket API，但它又根据Windows操作系统的特点进行了扩充。可认为是一种BSD Sockets的“方言”

- Winsock规范的核心内容是符合Berkeley Socket风格的库函数，为了使程序员能充分地利用Windows消息驱动机制进行编程，也定义开发了一组针对Windows的扩展库函数。

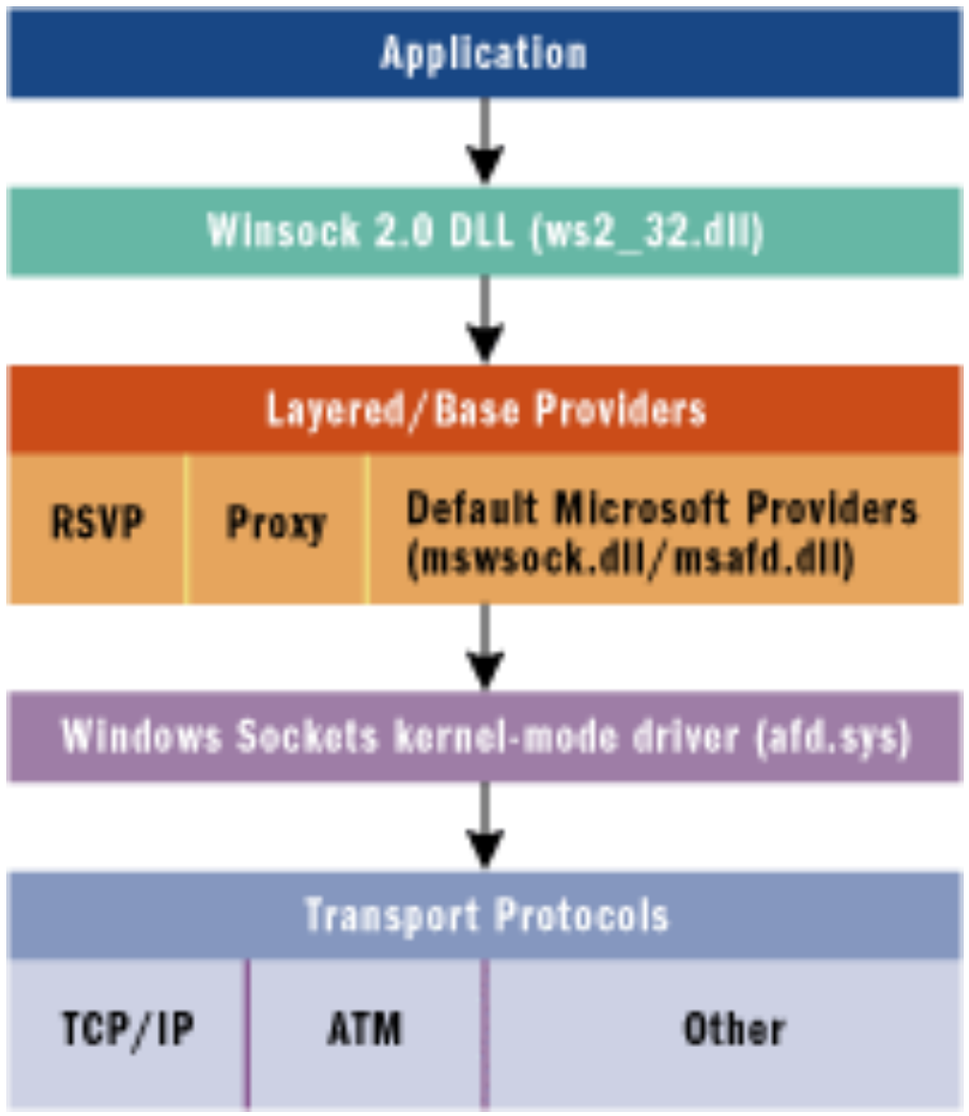
Socket的概念与工作原理

- Socket可以看成是两个网络应用程序进行通信时，各自通信连接中的一个端点，是一个逻辑上的概念

- 通信时其中的一个网络应用程序将要传输的一段信息写入它所在主机的Socket中，该Socket通过与网络接口卡（Network Interface Cards, NIC）相连的传输介质将这段信息发送到另外一台主机的Socket中，使这段信息能传送到其他程序中

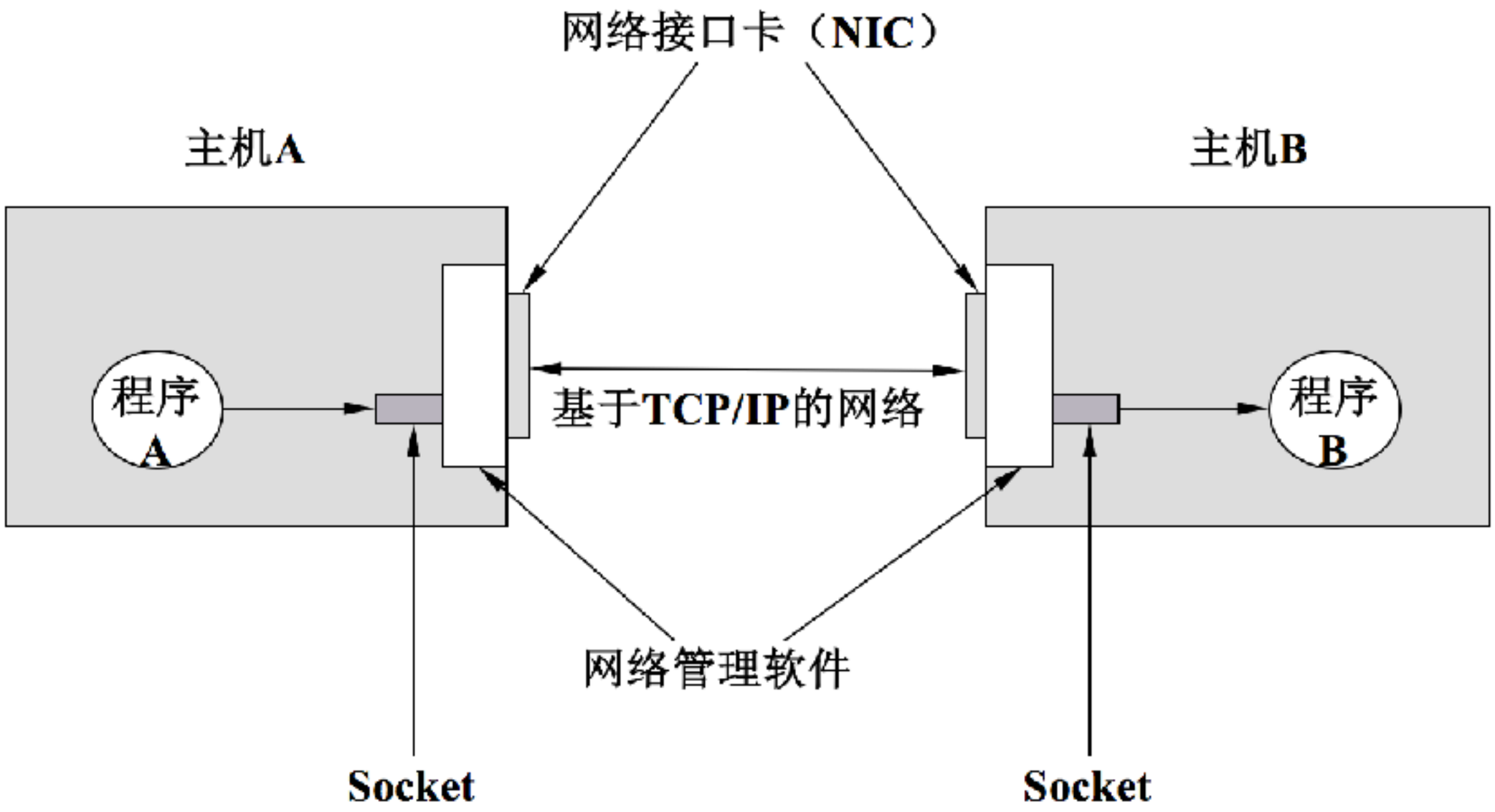
- 是否类似*nix中的文件系统?

- 按TCP/IP机制，网络分层管理
  - 硬件层通信：主要由设备厂家完成
  - 网络IP层、传输层：操作系统提供软件服务
    - 应用层软件：由各服务厂商提供，如浏览器、QQ等
- 用户自编的通信应用程序，工作在传输层和应用层之间。应用通信程序需要和工作在传输层之上的socket协议栈交换数据



- Winsock的一个工作原理图

- Socket的基本工作原理
  - 通过socket协议栈交换数据：
    - 通过API函数调用和操作，如send/write, receive/read等
    - 用户每注册一个socket，socket协议栈自动为用户每一个socket分配两个缓冲区：接收缓冲区、发送缓冲区


网络接口卡（NIC）
主机A
主机B
程序A
基于TCP/IP的网络
程序B
网络管理软件
Socket
Socket

构建一个Socket：以接电话为例

- 类比
  - 电话——进行语音数据交换
  - 网络——进行数字数据交换

- 接听一个电话（Socket）的第一步：
  - 安装电话机（初始化Socket）

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol)
```

```
import socket
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
```

socket()

- int socket(int domain, int type, int protocol)
  - domain: domain family 协议族
    - PF_INET
    - PF_INET6
    - PF_LOCAL
    - PF_PACKET
    - PF_IPX

socket()

- int socket(int domain, int type, int protocol)
  - type: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- protocol: 最终采取的协议:
  - IPPROTO_TCP
  - IPPROTO_UDP

- Winsock的启动与socket的建立
  - 第一步：Winsock头文件

```
#include <winsock2.h>

#pragma comment(lib,”WS2_32”)
```

- Winsock的启动与socket的建立
  - 第二步：先检测系统中有没有winsock的实现
    - WSAStartup()
  - 本函数必须是应用程序或DLL调用的第一个Windows Sockets函数，指明Windows Sockets API的版本号及获得特定Windows Sockets实现的细节
  - 应用程序或DLL只能在一次成功的WSAStartup()调用之后，才能进一步调用其它的Winsock API函数(如开始建立socket)

- WSAStartup格式
  - int WSAStartup(

WORD wVersionRequested,

LPWSADATA lpWSAData

);

- typedef struct WSAData{

  WORD wVersion;

  WORD wHighVersion;

  char szDescription[WSADESCRIPTION_LEN+1];

  char szSystemStatus[WSASYS_STATUS_LEN+1];

  unsigned short iMaxSockets;

  unsigned short iMaxUdpDg;

  char FAR * lpVendorInfo;

} WSADATA, FAR *LPWSADATA;

- WSAStartup()的返回值
    - WSASYSNOTREADY：在Winsock的头文件Winsock 2.h中，该错误代码定义的数值为10091，它表明加载的Winsock DLL不存在或底层的网络子系统无法使用
    - WSAVERNOTSUPPORTED：该代码的数值为10092，所需的Windows Sockets API的版本未由特定的Windows Sockets实现提供。如果由wVersion返回的版本用户不能接受，则要调用WSACleanup()函数清除对Winsock的加载。
    - WSAEINVAL：该代码的数值为10022，说明应用程序指出的Windows Sockets版本不能被该Winsock DLL的实现所支持。
    - WSAEINPROGRESS：该代码的数值为10036，说明一个阻塞的Winsock调用正在进行中。
    - WSAEPROCLIM：该代码的数值为10067，说明已经达到了Windows Sockets实现所支持的任务数量的极限。
    - WSAEFAULT：该代码数值为10014，说明lpWSAData参数是一个无效的指针。

- Winsock的启动与socket的建立
  - 第一步：Winsock头文件

    #include <winsock2.h>

    pragma comment(lib,”WS2_32”)
  - 第二步：先检测系统中有没有winsock的实现
    - WSAStartup()
  - 第三步：socket()或WSASocket()，分别对应winsock1和2的实现

- Winsock2提供的扩展格式
  - SOCKET WSASocket(

    int af,

    int type,

    int protocol,

    LPWSAPROTOCOL_INFO lpProtocolInfo,

    Group g,

    int iFlags

);

三种不同类型（type）的Socket

- 流套接字（SOCK_STREAM）

  - 提供了一种可靠的、面向连接的双向数据传输服务。被传输的数据看作是无记录边界的字节流

  - 在TCP/IP协议族中，使用TCP协议来实现字节流的传输

  - 大批量的数据，或者对数据的传输有较高的要求时使用

- 数据报套接字（SOCK_DGRAM）
  - 无连接、不可靠的双向数据传输
  - 数据包以独立的包形式被发送，保留记录边界，不提供可靠性保证。
  - 使用UDP协议等（也支持其它的协议）
  - 可用于出现差错可能性较小的网络，或广播通信

- 原始套接字（SOCK_RAW）
  - 原始套接字可以读写内核没有处理的IP数据包
  - （因为流套接字只能读取TCP协议的数据，数据包套接字只能读取UDP协议的数据）故若要访问其他协议发送数据必须使用原始套接字

构建一个Socket：以接电话为例

- 接听一个电话（Socket）的第二步：
  - 分配一个电话号码（绑定Socket与地址）

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

```
import socket
HOST='?.?.?.?'
PORT=…
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind(HOST,PORT)
```

bind()

- int bind(int sockfd, struct sockaddr *myaddr, socklen_t addrlen)
- 用本地地址为套接字命名
- 成功返回0，失败返回-1
- 什么情况需要调用bind()?

构建一个Socket：以接电话为例

- 接听一个电话（Socket）的第三步：
  - 等候接听电话（Socket的listen方法）

```
int listen(int sockfd, int backlog)
```

```
import socket
…
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.bind((HOST,PORT))
s.listen(1)
```

listen()

- int listen(int sockfd, int backlog)
  - 成功返回0，失败返回-1
    - backlog：用于在TCP层接收链接的缓冲池的最大个数，当客户链接请求大于这个数，则其它的未进入链接缓冲池的客户会自动重新链接，直到超时
- Winsock2中对应函数为listen()

构建一个Socket：以接电话为例

- 接听一个电话（Socket）的第四步：
  - 接听（Socket的accept方法）

```
int accept(int sockfd, struct sockaddr *myaddr,
socklen_t addrlen)
```

```
import socket
…
s.listen(1)
conn,address=s.accept()
```

accept()

- int accept(int sockfd, struct sockaddr *myaddr, socklen_t* addrlen)
  - 成功时返回（客户端socket的）文件描述符
  - 失败是返回-1
  - [py]返回（conn, address），分别为一个新的socket对象，即对方（客户端）的socket对象，和对方的地址
- Winsock2中对应函数为WSAAccept()

构建一个Socket：以接电话为例

- 接听一个电话（Socket）的第五步：
  - 挂电话

```
int close(int sockfd)
```

```
import socket
…
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
…
conn.close()
```

构建一个Socket：以接电话为例总结

- 上面我们构建了以一个服务器端（接电话）的Socket
- 一般过程为：
  - 创建socket
  - bind socket
  - listen to socket
  - accept (data from) socket

构建一个Socket：以拨电话为例

- 拨打一个电话（Socket）：
  - 初始化Socket（建立Socket）
  - 拨打电话

```
int connect(int sockfd, struct sockaddr *myaddr,
socklen_t addrlen)
```

```
c=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
c.connect((HOST,PORT))
```

一个例程

Hello world
in socket programming

hello_server.c

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  void error_handling(char *message);
8.
```

```
9.  int main(int argc, char *argv[])
10. {
11.     int serv_sock;
12.     int clnt_sock;
13.
14.     struct sockaddr_in serv_addr;
15.     struct sockaddr_in clnt_addr;
16.     socklen_t clnt_addr_size;
17.
18.     char message[]="Hello World!";
19.
20.     if(argc!=2)
21.     {
22.         printf("Usage : %s <port>\n", argv[0]);
23.         exit(1);
24.     }
25.
26.     serv_sock=socket(PF_INET, SOCK_STREAM, 0);
27.     if(serv_sock == -1)
28.         error_handling("socket() error");
29.
```

```
30.     memset(&serv_addr, 0, sizeof(serv_addr));
31.     serv_addr.sin_family=AF_INET;
32.     serv_addr.sin_addr.s_addr=htonl(INADDR_ANY);
33.     serv_addr.sin_port=htons(atoi(argv[1]));
34.
35.     if(bind(serv_sock, (struct sockaddr*) &serv_addr, sizeof(serv_addr))==-1)
36.         error_handling("bind() error");
37.
38.     if(listen(serv_sock, 5)==-1)
39.         error_handling("listen() error");
```

```
40.
41.     clnt_addr_size=sizeof(clnt_addr);
42.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
43.     if(clnt_sock==-1)
44.         error_handling("accept() error");
45.
46.     write(clnt_sock, message, sizeof(message));
47.     close(clnt_sock);
48.     close(serv_sock);
49.     return 0;
50. }
```

❖ hello_client.c

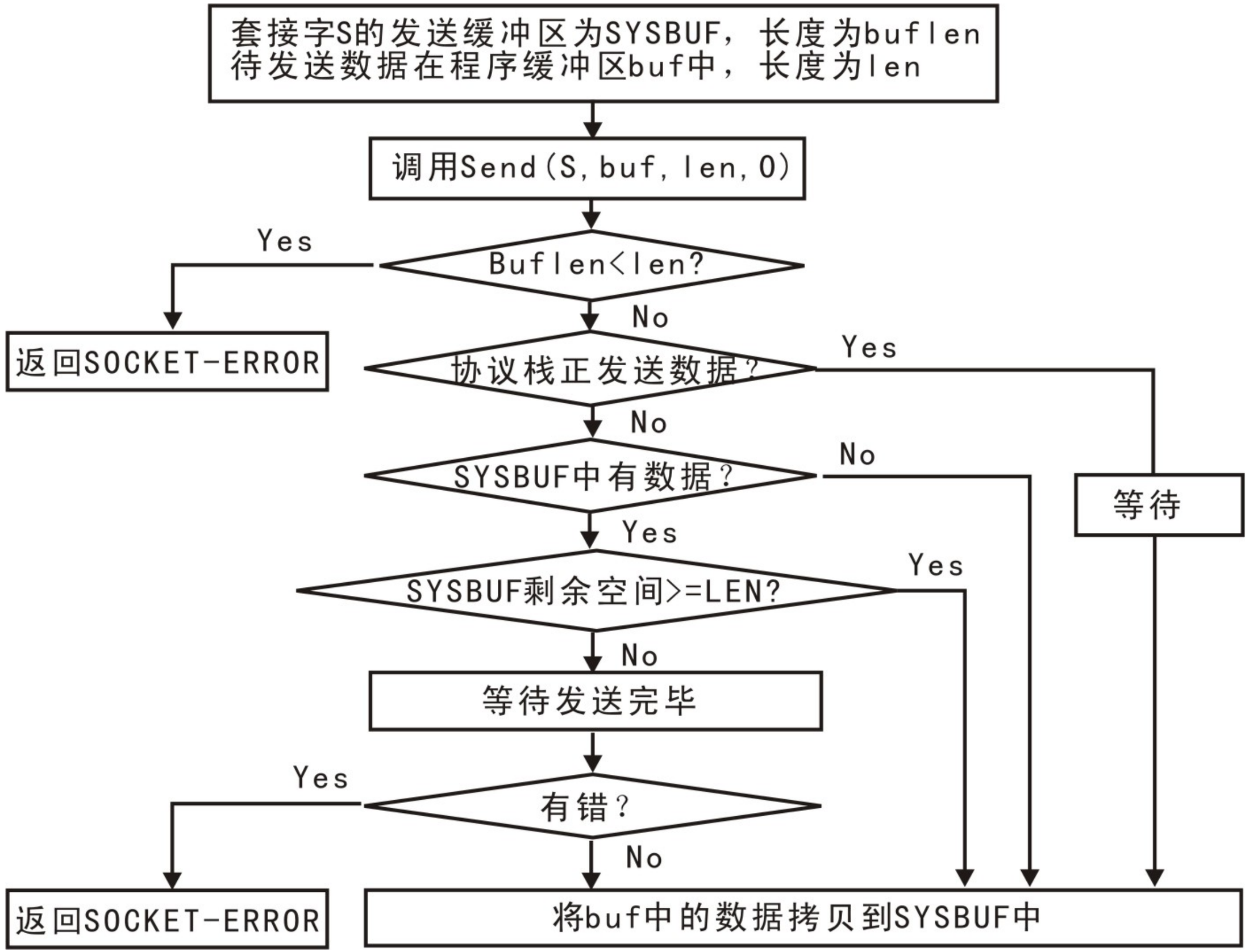
```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  #include <string.h>
4.  #include <unistd.h>
5.  #include <arpa/inet.h>
6.  #include <sys/socket.h>
7.  void error_handling(char *message);
8.
9.  int main(int argc, char* argv[])
10. {
11.     int sock;
12.     struct sockaddr_in serv_addr;
13.     char message[30];
14.     int str_len;
15.
16.     if(argc!=3)
17.     {
18.         printf("Usage : %s <IP> <port>\n", argv[0]);
19.         exit(1);
20.     }
```

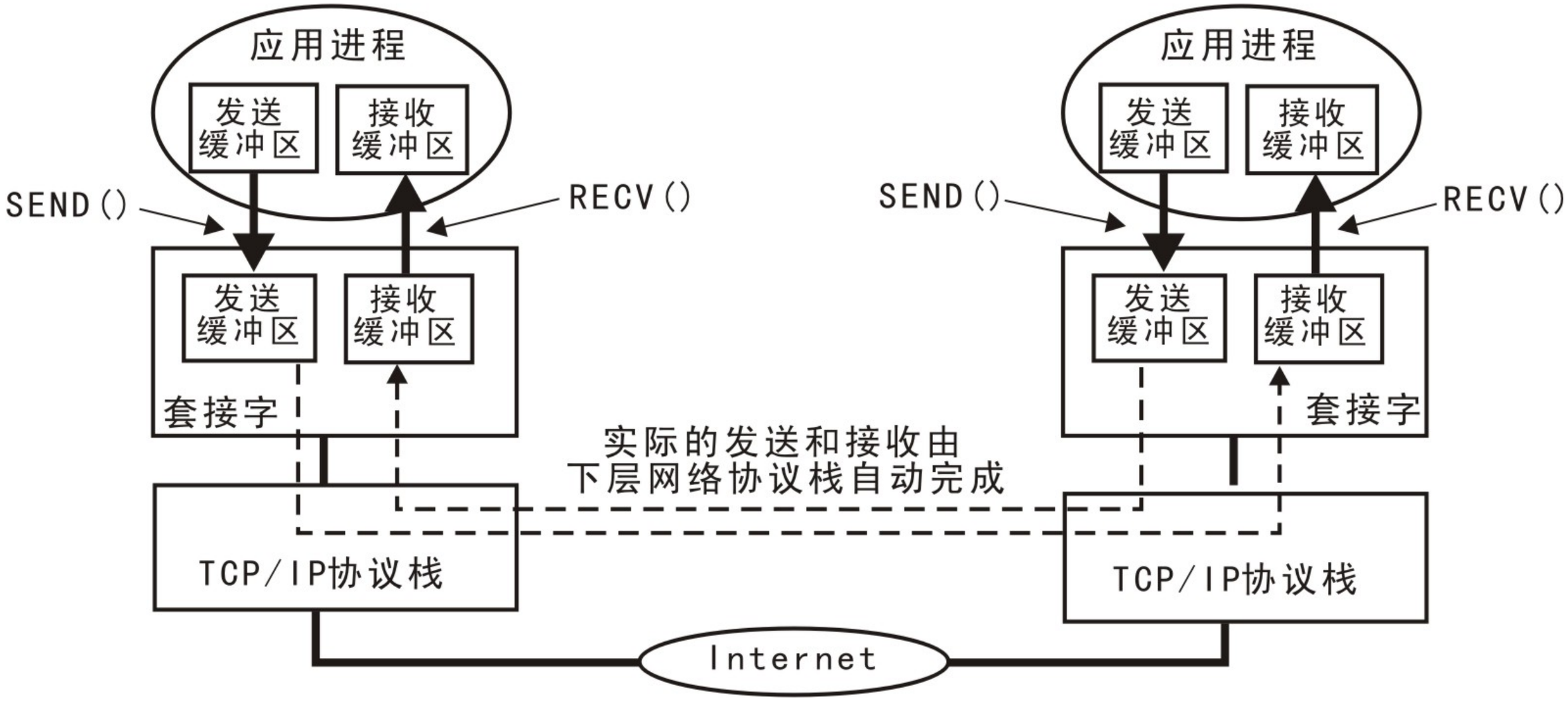
```
26.     memset(&serv_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family=AF_INET;
28.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_addr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
32.         error_handling("connect() error!");
33.
34.     str_len=read(sock, message, sizeof(message)-1);
35.     if(str_len==-1)
36.         error_handling("read() error!");
37.
38.     printf("Message from server : %s \n", message);
39.     close(sock);
40.     return 0;
```

Socket编程函数介绍（补充）

- Winsock中和read/write相对应函数
- recv() / WSARecv()
  - int recv(SOCKET s, char FAR* buf, int len, int flags)
- recvfrom() / WSARecvFrom()

- send() / WSASend()
  - int send(SOCKET s, const char FAR* buf, int len, int flags)
- sendto() / WSASendto()


套接字S的发送缓冲区为SYSBUF，长度为buflen
待发送数据在程序缓冲区buf中，长度为len
调用Send(S,buf,len,0)
Buflen<len?
Yes
返回SOCKET-ERROR
No
协议栈正发送数据?
Yes
等待
No
SYSBUF中有数据?
No
Yes
SYSBUF剩余空间>=LEN?
Yes
No
等待发送完毕
有错?
Yes
No
返回SOCKET-ERROR
将buf中的数据拷贝到SYSBUF中


应用进程
发送
缓冲区
接收
缓冲区
SEND()
RECV()
发送
缓冲区
接收
缓冲区
套接字
实际的发送和接收由
下层网络协议栈自动完成
TCP/IP协议栈
TCP/IP协议栈
Internet

```
40.
41.     clnt_addr_size=sizeof(clnt_addr);
42.     clnt_sock=accept(serv_sock, (struct sockaddr*)&clnt_addr, &clnt_addr_size);
43.     if(clnt_sock==-1)
44.         error_handling("accept() error");
45.
46.     write(clnt_sock, message, sizeof(message));
47.     close(clnt_sock);
48.     close(serv_sock);
49.     return 0;
50. }
```

Server

```
26.     memset(&serv_addr, 0, sizeof(serv_addr));
27.     serv_addr.sin_family=AF_INET;
28.     serv_addr.sin_addr.s_addr=inet_addr(argv[1]);
29.     serv_addr.sin_port=htons(atoi(argv[2]));
30.
31.     if(connect(sock, (struct sockaddr*)&serv_addr, sizeof(serv_addr))==-1)
32.         error_handling("connect() error!");
33.
34.     str_len=read(sock, message, sizeof(message)-1);
35.     if(str_len==-1)
36.         error_handling("read() error!");
37.
38.     printf("Message from server : %s \n", message);
39.     close(sock);
40.     return 0;
```

Client

Socket编程函数介绍（补充）

- 字节序转换
- htonl()
  - 将主机的无符号长整型数本机顺序转换为网络字节顺序 (Host to Network Long)，用于IP地址
  - u_long PASCAL FAR htonl(u_long hostlong)
  - hostlong是主机字节顺序表达的32位数。htonl()返回一个网络字节顺序的值

- htons()
  - 将主机的无符号短整型数转换成网络字节顺序(Host to Network Short)，用于端口号。
    - u_short PASCAL FAR htons(u_short hostshort);
    - hostshort：主机字节顺序表达的16位数。htons()返回一个网络字节顺序的值

- ntohl()
  - 将一个无符号长整型数从网络字节顺序转换为主机字节顺序。(Network to Host Long)，用于IP地址
    - u_long PASCAL FAR ntohl(u_long netlong);
    - netlong是一个以网络字节顺序表达的32位数，ntohl()返回一个以主机字节顺序表达的数

- ntohs()
  - 将一个无符号短整型数从网络字节顺序转换为主机字节顺序。(Network to Host Sort)，用于端口号
  - u_short PASCAL FAR ntohs(u_short netshort);
  - netshort是一个以网络字节顺序表达的16位数。ntohs()返回一个以主机字节顺序表达的数

Socket编程函数介绍（补充）

- 格式转换
- inet_addr()
  - 将一个点间隔地址转换成一个in_addr
  - unsigned long PASCAL FAR inet_addr(const struct FAR* cp)
  - cp：一个以Internet标准“.”间隔的字符串

- inet_ntoa()
  - 将网络地址转换成“.”点隔的字符串格式
  - char FAR* PASCAL FAR inet_ntoa(struct in_addr in)
  - in：一个表示Internet主机地址的结构

- 获取与套接口相连的端地址getpeername()
  - int getpeername(SOCKET s, struct sockaddr * name, int * namelen);
- 获取一个套接口的本地名字getsockname()
  - int getsockname(SOCKET s, struct sockaddr * name, int * namelen);

总结：流套接字通用编程模型

- 服务端：
  - 套接字的创建和关闭
  - 绑定套接字到指定的IP地址和端口号
  - 设置套接字进入监听状态
  - 接收连接请求
  - 收发数据
- 客户端：
  - 套接字创建和关闭
  - 申请建立连接
  - 收发数据
  - 断开连接，关闭

服务器端

Socket()建立套接字，返回套接字号s

bind()，套接字s与本地接口绑定

listen()，通知TCP服务器准备好接收连接

accept()，接收连接，等待客户端的请求

建立连接，accept()返回，得到新的套接字

recv()/send()，接收/发送数据

closesocket()，关闭套接字

客户端

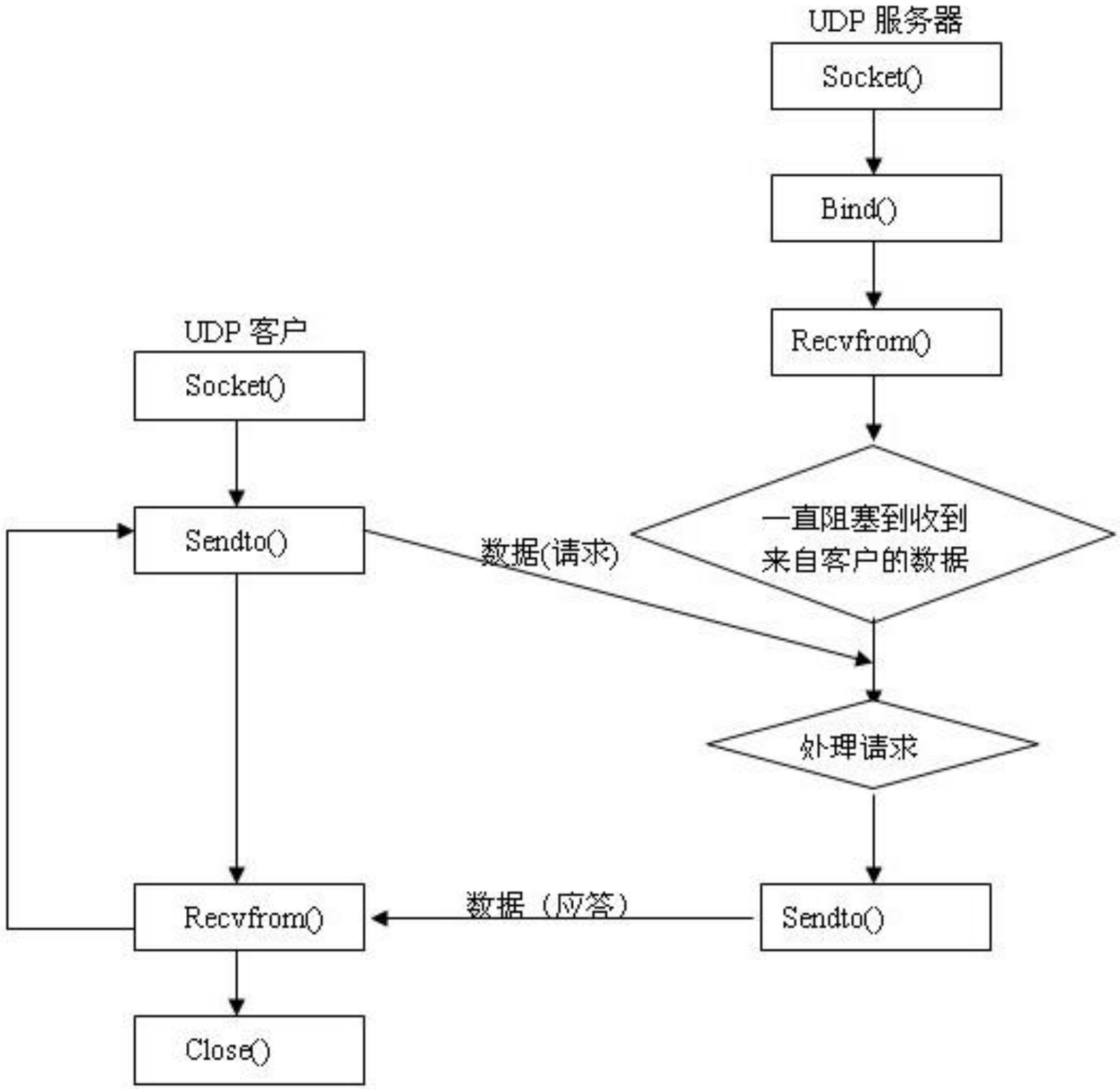
Socket()建立套接字，返回套接字号s

Connect()，主动发出请求，连接服务器

send()/recv()，发送/接收数据

总结：数据报套接字通用编程模型

- 服务端：
  - 创建套接字
  - 绑定IP地址和端口
  - 收发数据
  - 关闭套接字
- 客户端：
  - 创建套接字
  - 收发数据
  - 关闭套接字


UDP 服务器
Socket()
Bind()
Recvfrom()
一直阻塞到收到
来自客户的数据
处理请求
Sendto()
UDP 客户
Socket()
Sendto()
数据(请求)
Recvfrom()
数据（应答）
Close()

总结：常用的socket函数

主要函数	
socket()	创建一个套接字，并返回套接字的标识符
bind()	把套接字绑定到特定的网络地址上
listen()	启动指定的套接字，监听到来的连接请求
accept()*	接收一个连接请求，并新建一个套接字，原来的套接字返回监听状态
connect()*	请求讲本地套接字连接到一个指定的远方套接字上
send()*	向一个已经与对方建立连接的套接字发送数据
sendto()*	向一个未与对方建立连接的套接字发送数据，并指定对方网络地址
recv()*	从一个已经与对方建立连接的套接字接收数据
recvfrom()*	从一个未与对方建立连接的套接字接收数据，并返回对方网络地址
shutdown()	有选择的关闭套接字的全双工连接
closesocket()*	关闭套接字，释放相应的资源

辅助函数	
htonl()	**把32位无符号数从主机字节序转换为网络字节序**
htons()	**把16位无符号数从主机字节序转换为网络字节序**
ntohl()	**把32位无符号数从网络字节序转换为主机字节序**
ntohs()	**把16位无符号数从网络字节序转换为主机字节序**
Inet_addr()	**把标准的点分十进制的IP转换成长整形地址数据**
Inet_ntoa()	**把长整形的IP地址数据转换成点分十进制的字符串**
getpeername()	**获得套接字连接上对方的网络地址**
getsocketname()	**获得指定套接字的网络地址**
控制函数	
getsocketopt()	**获得指定套接字的属性选项**
setsocketopt()	**设置与指定套接字相关的属性选项**
ioctlsocket()	**为套接字提供控制**
select()*	**执行同步I/O多路复用**

总结：网络编程中
可能出现/需要考虑的问题

- Socket工作中的问题
  - 通信阻塞：发送量超过发送缓冲区容量，或者接收不及时导致接收缓冲区满。
  - 数据到达通知：对于接收缓冲区，由于数据到达是随机事件，通信处理方式：1. 定期查询接收缓冲区；2. 有数据到达时，通知用户处理

- 网络异构特性所产生的问题
  - 字节序
  - 字的长度：不同的系统中，对于相同的数据类型可能用不同的长度表示
  - 字节定界问题：不同的平台上给结构（struct）或联合（union）打包的方式也是不同的

- 通信模式的问题——阻塞与非阻塞
  - 在网络编程中，通信可以选择阻塞与非阻塞两种模式
  - 对于发送端，若底层协议没有空间存放用户数据，应用程序会选择进行等待，或者直接返回而不等待
  - 在应用进程调用接收函数接收报文时，如果是在阻塞模式下，若没有到达的数据，则调用将一直阻塞直到有数据到达或出错为止
  - UDP协议不同，因为UDP没有发送缓冲，所有UDP协议即使在阻塞模式下也不会发生阻塞，即没有真正的阻塞模式

- 阻塞与非阻塞（续）
  - 在连接建立阶段，不管是阻塞还是非阻塞模式，发起连接请求的一方总是会使调用它的进程阻塞，阻塞间隔最少等于到达服务器的一次往返时间
- 通信模式对应用程序性能有的影响
  - 非阻塞模式，应用程序轮询耗费CPU
  - 阻塞模式，应用程序I/O操作被阻塞

完