上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

实验报告

LAB REPORT

数据通信

Winsock ex2

姓 名: 裘炜程

学 号: 516030910287

班 级: F1603602

一、实验要求

1、Write a stream based echo server printing out the message received from the client, echoing it back, until the client closes the connection.

Write a stream based echo client sending messages to the echo server, receiving each message returned by the server. Terminate the connection when “quit” is entered.

2、Modify your solution to Exercise 2 to write a stream based echo server, which can simultaneously handle multiple clients connecting to it. No modification of the client code is necessary, but multiple instances of the client should be started. Hint: use Windows threads functions.

二、实验原理

1、EchoClient 基本步骤

（1）初始化 Winsock: WSAStartup( )

（2）创建 socket:socket( )

（3）接受地址信息，result 以链表储存该信息，每一个节点存放一个 ip 及其信息

（4）根据 result 连接 server: connnect( )

（5）发送接收数据: send( )、receive( )

（6）关闭连接: shutdown( ), closesocket( ), WSACleanup( )

2、EchoServer 基本步骤

（1）初始化 Winsock: WSAStartup( )

（2）创建 socket:socket( )

（3）接受地址信息，result 以链表储存该信息，每一个节点存放一个 ip 及其信息

（4）绑定 socket: bind( )

（5）在 socket 上监听: listen( )

（6）接受客户端发来的请求: accept( )，并创建一个 ClientSocket

（7）用 ClientSocket 发送接收数据: send( )、receive( )

（8）关闭连接: shutdown( ), closesocket( ), WSACleanup( )

3、核心函数

（1）connect( )：客户端，传入创建好的 socket 和地址信息

Call the **connect** function, passing the created socket and the **sockaddr** structure as parameters.

```
/*  int connect(
    _In_ SOCKET s,     //A descriptor identifying an unconnected socket.
    _In_ const struct sockaddr *name, //A pointer to the sockaddr structure to which
                                      //the connection should be established.
    _In_ int    namelen //The length, in bytes, of the sockaddr structure pointed to by the name parameter.
   ); */

   // Connect to server.
   iResult = connect( ConnectSocket, ptr->ai_addr, (int)ptr->ai_addrlen);
   if (iResult == SOCKET_ERROR) {
       closesocket(ConnectSocket);
       ConnectSocket = INVALID_SOCKET;
   }
```

（2）bind( )：服务器，传入创建好的 socket 和地址信息

For a server to accept client connections, it must be bound to a network address within the system.

```
/*    int bind(
      _In_ SOCKET  s, // A descriptor identifying an unbound socket.
      _In_ const struct sockaddr *name, // A pointer to a sockaddr structure of the local address to assign
                                        // to the bound socket .
      _In_ int  namelen // The length, in bytes, of the value pointed to by the name parameter.
     ); */
```

The **sockaddr** structure holds information regarding the address family, IP address, and port number. Client applications use the IP address and port to connect to the host network.

// Call the **bind** function, passing the created **socket** and **sockaddr** structure returned from the **getaddrinfo** function as parameters. Check for general errors.

```
iResult = bind( ListenSocket, result->ai_addr, (int)result->ai_addrlen);
    if (iResult == SOCKET_ERROR) {
        printf("bind failed with error: %d\n", WSAGetLastError());
        freeaddrinfo(result);
        closesocket(ListenSocket);
        WSACleanup();
        return 1;
    }
freeaddrinfo(result); // Once the bind function is called, the address information returned
                      // by the getaddrinfo function is no longer needed.
```


（3）listen( ):服务器监听，传入上一步创建的 socket 和最大连接数

After the socket is bound to an IP address and port on the system, the server must then listen on that IP address and port for incoming connection requests.

```
/*  int listen(
    _In_ SOCKET s,   // A descriptor identifying a bound, unconnected socket
    _In_ int    backlog   // The maximum length of the queue of pending connections.
   );
```

If no error occurs, **listen** returns zero. Otherwise, a value of **SOCKET_ERROR** is returned.

```
if ( listen( ListenSocket, SOMAXCONN ) == SOCKET_ERROR ) {
    printf( "Listen failed with error: %ld\n", WSAGetLastError() );
    closesocket(ListenSocket);
    WSACleanup();
    return 1;
}
```

（4）accept( )：服务器，传入监听的 socket，传出收到的地址，返回一个客户端 socket 结构

Once the socket is listening for a connection, the program must handle connection requests on that socket.

```
/*  SOCKET accept(
    _In_  SOCKET  s,  // A descriptor that identifies a socket that has been placed in a listening state with the listen
                      // function. The connection is actually made with the socket that is returned by accept.
    _Out_  struct  sockaddr  *addr,  // An optional pointer to a buffer that receives the address of the connecting entity
    _Inout_ int     *addrlen  // An optional pointer to an integer that contains the length addr parameter.
    ); */
```

If no error occurs, **accept** returns a value of type **SOCKET** that is a descriptor for the new socket. This returned value is a handle for the socket on which the actual connection is made.

```
//Create a temporary SOCKET object called ClientSocket for accepting connections from clients.
        SOCKET ClientSocket;
        ClientSocket = INVALID_SOCKET;

//The following example just listens for and accepts only a single connection.

        // Accept a client socket
        ClientSocket = accept(ListenSocket, NULL, NULL);
        if (ClientSocket == INVALID_SOCKET) {
            printf("accept failed: %d\n", WSAGetLastError());
            closesocket(ListenSocket);
            WSACleanup();
            return 1;
        }
        closesocket(ListenSocket); // No longer need server socket
```

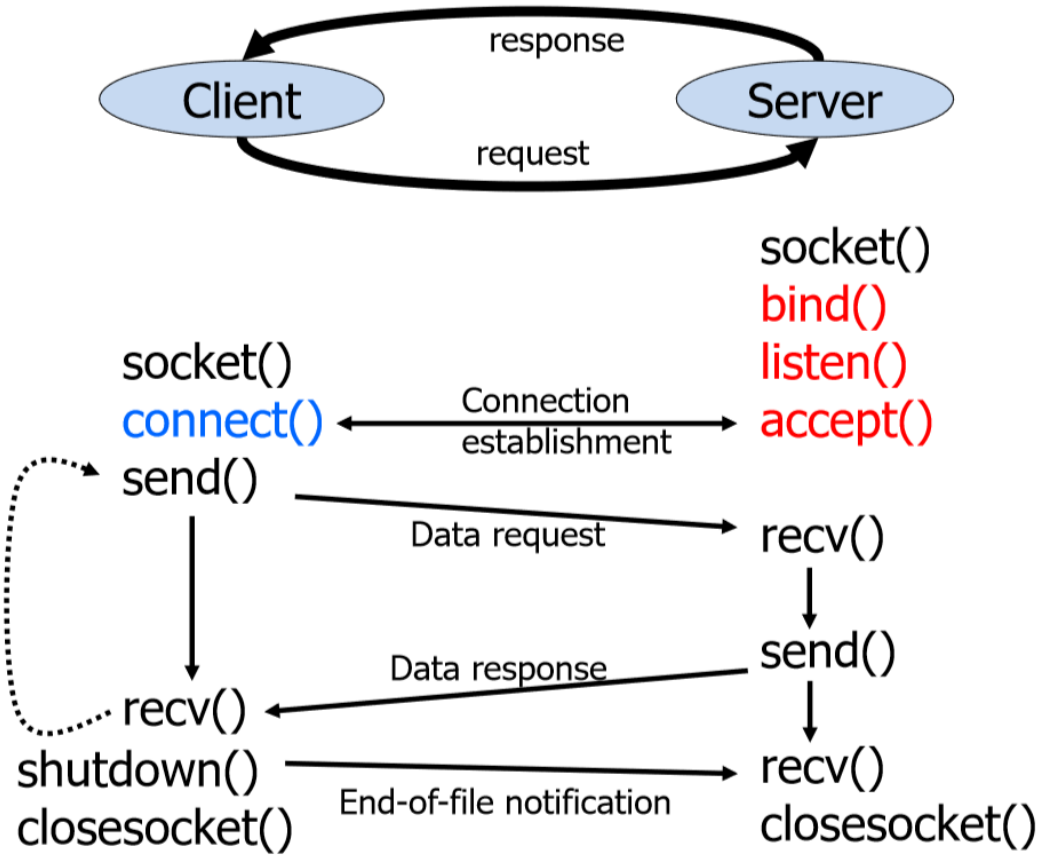

（5）send( )、recv( )：客户端和服务器接受发送数据，输入发送的字符串和长度

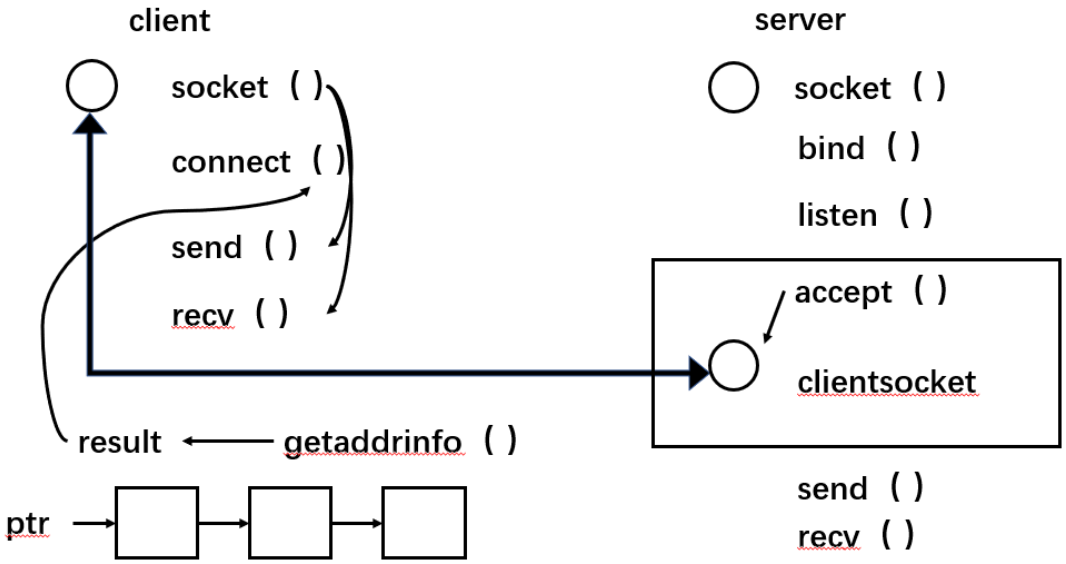
```
#define DEFAULT_BUFLEN 512

char recvbuf[DEFAULT_BUFLEN];
int iResult, iSendResult;
int recvbuflen = DEFAULT_BUFLEN;

// Receive until the peer shuts down the connection
do {

    iResult = recv(ClientSocket, recvbuf,
recvbuflen, 0);
    if (iResult > 0) {
        printf("Bytes received: %d\n",
iResult);
```

```
// Echo the buffer back to the sender
        iSendResult = send(ClientSocket,
recvbuf, iResult, 0);
        if (iSendResult == SOCKET_ERROR) {
            printf("send failed: %d\n",
WSAGetLastError());
            closesocket(ClientSocket);
            WSACleanup();
            return 1;
        }
        printf("Bytes sent: %d\n",
iSendResult);
    } else if (iResult == 0)
        printf("Connection closing...\n");
    else {
        printf("recv failed: %d\n",
WSAGetLastError());
        closesocket(ClientSocket);
        WSACleanup();
        return 1;
    }

} while (iResult > 0);
```

50

其他的函数已经在第一次实验报告中分析过了，这里就不再说明。总体来说服务器和客户端的连接过程和函数连接为：





4、多线程处理

不断地接受客户端的请求，直到客户端关闭。在循环中不断创建 clientsocket，然后对于每一个线程进行 send 和 recv 的处理。

三、实验结果

1、EchoClient 和 EchoServer

```
Microsoft Windows [版本 10.0.17134.706]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\hp>cd C:\Users\hp\Desktop\数据通信\winsock_ex2\Echo\Debug

C:\Users\hp\Desktop\数据通信\winsock_ex2\Echo\Debug>EchoClient 127.0.0.1
apple
Received: apple
banana
Received: banana
quit
Bytes Sent: 0
Connection closed

C:\Users\hp\Desktop\数据通信\winsock_ex2\Echo\Debug>
```

```
Microsoft Windows [版本 10.0.17134.706]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\hp>cd C:\Users\hp\Desktop\数据通信\winsock_ex2\EchoServer\Debug

C:\Users\hp\Desktop\数据通信\winsock_ex2\EchoServer\Debug>EchoServer
Received: apple
Received: banana
Connection closing...

C:\Users\hp\Desktop\数据通信\winsock_ex2\EchoServer\Debug>
```

2、EchoClient 和多线程的 EchoServer2

```
C:\Users\hp\Desktop\数据通信\winsock_ex2\Echo\Debug>EchoClient 127.0.0.1
apple
Received: apple
quit
Bytes Sent: 0
Connection closed
```

```
C:\Users\hp\Desktop\数据通信\winsock_ex2\Echo\Debug>EchoClient localhost
banana
Received: banana
quit
Bytes Sent: 0
Connection closed
```

```
C:\Users\hp\Desktop\数据通信\winsock_ex2\EchoServer2\Debug>EchoServer2
Count=1
Connection 1 accepted!!!
        Client socket number: 296
        IPv4 address: 127.0.0.1
        Port nuber: 2756
Count=2
thread beninging...
Received from client... data = 'apple'
Connection 2 accepted!!!
        Client socket number: 304
        IPv4 address: 127.0.0.1
        Port nuber: 2760
Count=3
thread beninging...
Received from client... data = 'banana'
thread completed...
thread completed...
```

四、实验思考

在代码中出现一个问题:在每一次 send 时总要多发送一位,如:send(ConnectSocket, sendbuf, int(strlen(sendbuf)+1), 0)。原因是每一次发送的数据放在 char 中，并固定了位数，而 char 存放数据时会在字符串最后加一位终结符，因此在发送时需要将这一位一起发送，表示在传到终结符一位时数据结束，接收端收到结束信息就可以打印。如果不发送终结符，接收端会将整个 char 包括缓冲区的东西全部打印，因而出现“烫烫烫”。

在实验中,其他同学发现如果在客户端输入结束命令 CTRL+C,服务器端会显示异常，说明本来代码中并没有对 CTRL+C 进行处理。因此我的解决方法是分析 CTRL+C 发送时的情况，实际上在发送这个的时候会先发一个空串，然后返回一个-1，而 quit 是返回 0，

因此在接受时将recv((unsigned int)client_s, in_buf, sizeof(in_buf), 0) == 0的判断改为<=0即可，但是还存在的问题是空串依然会被打印出来，这个问题如何解决还没有想到。