**Winsock Exercises 3**

**Exercise 3.1** (WebClient)

Write a client program to execute a single HTTP GET to a Web server.

A web client performs the following steps:

1. Initializes Winsock.
2. Creates a socket.
3. Connects to the server.
4. Sends and receives data.
5. Disconnects.

Refer to the instructions in exercise 1 for creating a basic Winsock application and initializing Winsock, and follow the steps described below.

1. Declare an addrinfo object that contains a sockaddr structure and initialize it. For this application, the Internet address family is unspecified so that either an IPv6 or IPv4 address can be returned. The application requests the socket type to be a stream socket for the TCP protocol.

struct addrinfo \*result = NULL,

\*ptr = NULL,

hints;

ZeroMemory( &hints, sizeof(hints) );

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

2. Call the getaddrinfo function requesting the IP address for the server name passed on the command line. The TCP port on the server that the client will connect to is defined by DEFAULT\_PORT as 80.

#define DEFAULT\_PORT "80"

// Resolve the server address and port

iResult = getaddrinfo(argv[1], DEFAULT\_PORT, &hints, &result);

if (iResult != 0) {

cout << "getaddrinfo failed: " << iResult << endl;

WSACleanup();

return 1;

}

3. Create a SOCKET object called ClientSocket.

SOCKET ClientSocket = INVALID\_SOCKET;

4. Call the socket function and return its value to the ClientSocket variable.

// Attempt to connect to the first address returned by

// the call to getaddrinfo

ptr = result;

// Create a SOCKET for connecting to server

ClientSocket = socket(ptr->ai\_family, ptr->ai\_socktype, ptr->ai\_protocol);

if (ClientSocket == INVALID\_SOCKET) {

cout << "Error at socket(): " << WSAGetLastError() << endl;

freeaddrinfo(result);

WSACleanup();

return 1;

}

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

int iResult;

// Connect to server.

iResult = connect( ClientSocket, ptr->ai\_addr, (int)ptr->ai\_addrlen);

if (iResult == SOCKET\_ERROR) {

closesocket(ClientSocket);

ClientSocket = INVALID\_SOCKET;

}

// Should really try the next address returned by getaddrinfo

// if the connect call failed

// But for this simple example we just free the resources

// returned by getaddrinfo and print an error message

freeaddrinfo(result);

if (ClientSocket == INVALID\_SOCKET) {

cout << "Unable to connect to server!\n";

WSACleanup();

return 1;

}

6. The send function to be used by the client once a connection is established.

#define DEFAULT\_BUFLEN 512

const char \*sendbuf = "GET / HTTP/1.0\n\n";

// Send an initial buffer

iResult = send(ClientSocket, sendbuf, (int) strlen(sendbuf), 0);

if (iResult == SOCKET\_ERROR) {

cout << "send failed: " << WSAGetLastError() << endl;

closesocket(ClientSocket);

WSACleanup();

return 1;

}

cout << "Bytes Sent: " << iResult << endl;

7. When the client is done sending data to the server, the shutdown function can be called specifying SD\_SEND to shutdown the sending side of the socket. This allows the server to release some of the resources for this socket. The client application can still receive data on the socket.

// shutdown the connection for sending since no more data will be sent

// the client can still use the ClientSocket for receiving data

iResult = shutdown(ClientSocket, SD\_SEND);

if (iResult == SOCKET\_ERROR) {

cout << "shutdown failed: " << WSAGetLastError() << endl;

closesocket(ClientSocket);

WSACleanup();

return 1;

}

8. The receive function to be used by the client once a connection is established.

char recvbuf[DEFAULT\_BUFLEN];

int recvbuflen = DEFAULT\_BUFLEN;

// Receive data until the server closes the connection

do {

iResult = recv(ClientSocket, recvbuf, recvbuflen, 0);

if (iResult > 0){

cout << "Bytes received: " << iResult << endl;

cout << recvbuf << endl;

}

else if (iResult == 0)

cout << "Connection closed\n";

else

cout << "recv failed: " << WSAGetLastError() << endl;

} while (iResult > 0);

9. When the client application is done receiving data, the closesocket function is called to close the socket.

// cleanup

closesocket(ClientSocket);

WSACleanup();

**Hand in**

1. Your program WebClient.cpp
2. A report showing (with screen shots)
   1. The response from "www.sjtu.edu.cn" on port 80 to the message "GET / HTTP/1.0\n\n ".
   2. The response from "www.baidu.com" on port 80 to the message "GET / HTTP/1.0\n\n".
   3. The response from "www.sina.com" on port 80 to the message "GET / HTTP/1.0\n\n".
   4. Are the responses the same? Explain why.
   5. Other interesting results you have got.

**Exercise 3.2** (WebServer)

Write a program to accept a single connect from a web browser (i.e., acts as an HTTP server) and responds with an HTML message.

#define DEFAULT\_PORT "80"

char outbuf[DEFAULT\_BUFLEN];

// Receive from the Web browser

// - The return code from recv() is the number of bytes received

iResult = recv(ClientSocket, recvbuf, BUF\_SIZE, 0);

for (i=0; i<iResult; i++)

printf ("%c", recvbuf[i]);

// Copy the HTML response into the out buffer

strcpy\_s(outbuf, "<html><body><hr>This is a response <b>message</b> in HTML \

format. <font color=red>Wow!</font><hr></body></html>");

// Send HTML response to the client

send(ClientSocket, outbuf, strlen(outbuf), 0);

**Hand in**

1. WebServer.cpp
2. A report showing the execution of your program and other interesting results.

**Exercise 3.3**. (WebServer2)

Write an extended Web server for Windows that serves HTML, text, and GIF images. Hint: to ensure simultaneous processing of multiple clients each GET should spawn its own thread.

#include <fcntl.h> // For binary handle options

#include <sys\stat.h> // For binary write()

#include <io.h> // Needed for open(), close(), write()

#include <process.h> // Needed for \_beginthread() and \_endthread()

//----- HTTP response messages ----------------------------------------------

#define OK\_IMAGE "HTTP/1.0 200 OK\r\nContent-Type:image/gif\r\n\r\n"

#define OK\_TEXT "HTTP/1.0 200 OK\r\nContent-Type:text/html\r\n\r\n"

#define NOTOK\_404 "HTTP/1.0 404 Not Found\r\nContent-Type:text/html\r\n\r\n"

#define MESS\_404 "<html><body><h1>FILE NOT FOUND</h1></body></html>"

//----- Defines -------------------------------------------------------------

#define BUF\_SIZE 1024 // Buffer size (big enough for a GET)

#define PORT\_NUM 80 // Port number for a Web server

//----- Function prototypes -------------------------------------------------

void handle\_get(void \*in\_arg); // Thread function to handle GET

//----Add these variables in main()----

struct sockaddr client\_addr;

int addr\_len;

SOCKET client\_s = INVALID\_SOCKET;

// Main loop to listen, accept, and then spin-off a thread to handle the GET

while(1)

{

printf("main loop: linstening ... \n");

// Listen for connections and then accept

listen(ListenSocket, 50);

addr\_len = sizeof(client\_addr);

client\_s = accept(ListenSocket, (struct sockaddr \*)&client\_addr, &addr\_len);

if (client\_s == -1)

{

printf("ERROR - Unable to create a socket \n");

exit(1);

}

printf("client socket accepted, %d... \n",client\_s);

// Spin-off a thread to handle this request (pass only client\_s)

if (\_beginthread(handle\_get, 4096, (void \*)client\_s) < 0)

{

printf("ERROR - Unable to create a thread to handle the GET \n");

exit(1);

}

}

printf("main loop completed. close server socket... WSAcleanup \n");

// Close the server socket and clean-up winsock

closesocket(ListenSocket);

WSACleanup();

//===========================================================================

//= This is the thread function to handle the GET =

//===========================================================================

void handle\_get(void \*in\_arg)

{

unsigned int client\_s; // Client socket descriptor

char in\_buf[BUF\_SIZE]; // Input buffer for GET request

char out\_buf[BUF\_SIZE]; // Output buffer for HTML response

int fh; // File handle

int buf\_len; // Buffer length for file reads

char command[BUF\_SIZE]; // Command buffer

char file\_name[BUF\_SIZE]; // File name buffer

int retcode; // Return code

int j;

// Set client\_s to in\_arg

client\_s = (unsigned int) in\_arg;

printf("thread %d... \n", client\_s);

// Receive the (presumed) GET request from the Web browser

retcode = recv(client\_s, in\_buf, BUF\_SIZE, 0);

printf("thread %d...received web request: \n", client\_s);

for (j=0; j<retcode; j++)

printf ("%c", in\_buf[j]);

// If the recv() return code is bad then bail-out (see note #3)

if (retcode <= 0)

{

printf("ERROR - Receive failed --- probably due to dropped connection \n");

closesocket(client\_s);

\_endthread();

}

// Parse out the command from the (presumed) GET request and filename

sscanf\_s(in\_buf, "%s %s \n", &command, BUF\_SIZE,&file\_name, BUF\_SIZE);

// Check if command really is a GET, if not then bail-out

if (strcmp(command, "GET") != 0)

{

printf("ERROR - Not a GET --- received command = '%s' \n", command);

closesocket(client\_s);

\_endthread();

}

// It must be a GET... open the requested file

// - Start at 2nd char to get rid of leading "\"

\_sopen\_s(&fh, &file\_name[1], \_O\_RDONLY | \_O\_BINARY,

\_SH\_DENYNO, \_S\_IREAD | \_S\_IWRITE);

// If file does not exist, then return a 404 and bail-out

if (fh == -1)

{

printf("File '%s' not found --- sending an HTTP 404 \n", &file\_name[1]);

strcpy\_s(out\_buf, NOTOK\_404);

send(client\_s, out\_buf, strlen(out\_buf), 0);

strcpy\_s(out\_buf, MESS\_404);

send(client\_s, out\_buf, strlen(out\_buf), 0);

closesocket(client\_s);

\_endthread();

}

// Check that filename does not start with a "..", "/", "\", or have a ":" in

// the second position indicating a disk identifier (e.g., "c:").

// - This is a security check to prevent grabbing any file on the server

if (((file\_name[1] == '.') && (file\_name[2] == '.')) ||

(file\_name[1] == '/') || (file\_name[1] == '\\') ||

(file\_name[2] == ':'))

{

printf("SECURITY VIOLATION --- trying to read '%s' \n", &file\_name[1]);

\_close(fh);

closesocket(client\_s);

\_endthread();

}

// Generate and send the response

printf("Thread %d, ...Sending file '%s' \n",in\_arg, &file\_name[1]);

//search .gif in file\_name

if (strstr(file\_name, ".gif") != NULL)

strcpy\_s(out\_buf, OK\_IMAGE);

else

strcpy\_s(out\_buf, OK\_TEXT);

send(client\_s, out\_buf, strlen(out\_buf), 0);

while(!\_eof(fh))

{

buf\_len = \_read(fh, out\_buf, BUF\_SIZE);

send(client\_s, out\_buf, buf\_len, 0);

}

// Close the file, close the client socket, and end the thread

\_close(fh);

printf("Thread %d, ...comleted sending file '%s' \n", client\_s, &file\_name[1]);

closesocket(client\_s);

printf("socket %d closed. \n",client\_s);

printf("thread %d ended. \n", client\_s);

\_endthread();

}

**Hand in**

1. WebServer2.cpp
2. A report showing the execution of your program and other interesting results.