**Winsock Exercises**

**Exercise 1** (GetHostName)

Task 1: Write a sockets program to get the host name for a given IP address.

Project name: gethostname

Command: gethostname (printing out the host name of your local computer)

Command: gethostname xxx.xxx.xxx.x (printing out the host name of a remote host)

Task 2: Write a sockets program to get the IP address of a given host name.

Project name: gethostaddress

Command: gethostaddress www.sjtu.edu.cn

1. To create a basic Winsock application, create a new empty project with the source file:

#include "pch.h"

#include <winsock2.h>

#include <ws2tcpip.h>

#pragma comment(lib, "Ws2\_32.lib")// Used to link with Ws2\_32.lib

#include <iostream>

using namespace std;

int main(int argc, char \*argv[] ) {

return 0;

}

2. To initialize Winsock and initiate use of WS2\_32.dll:

WORD wVersion = MAKEWORD(2, 2); // Used to request version 2.2 of Windows sockets

WSADATA wsaData; // Data loaded by WSAStartup

int iResult; // Error check if WSAStartup successful

// Initialize Winsock

iResult = WSAStartup(wVersion, &wsaData);

if (iResult != 0) {

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

return 1;

}

3. To close the windows sockets, the following goes at the end of the program:

WSACleanup();

4. To get your local hostname:

char host\_name[128];

gethostname(host\_name, sizeof(host\_name));

cout << host\_name << endl;

5. To retrieve host name for a given IP address passed on the command line:

//To simplify determining buffer requirements for the host and serv parameters, the following values

//for maximum host name length and maximum service name are defined in the Ws2tcpip.h header

//file.

//#define NI\_MAXSERV 32

//#define NI\_MAXHOST 1025

// Validate the parameters

if (argc != 2) {

printf("usage: %s IPv4 address\n", argv[0]);

printf(" to return local hostname\n");

printf(" %s 127.0.0.1\n", argv[0]);

return 1;

}

DWORD dwRetval;

struct sockaddr\_in saGHN;

char hostname[NI\_MAXHOST];

char servInfo[NI\_MAXSERV];

u\_short port = 27015;

//-----------------------------------------

// Set up sockaddr\_in structure which is passed

// to the getnameinfo function

saGHN.sin\_family = AF\_INET;

inet\_pton(AF\_INET, argv[1], &saGHN.sin\_addr.s\_addr);

saGHN.sin\_port = htons(port);

//-----------------------------------------

// Call getnameinfo

dwRetval = getnameinfo((struct sockaddr \*) &saGHN,

sizeof(struct sockaddr),

hostname,

NI\_MAXHOST, servInfo, NI\_MAXSERV, 0);

if (dwRetval != 0) {

printf("getnameinfo failed with error # %ld\n", WSAGetLastError());

return 1;

}

else {

printf("getnameinfo returned hostname = %s\n", hostname);

return 0;

}

6. To retrieve the IP address of a host name passed on the command line:

// Declare and initialize variables

struct addrinfo \*result = NULL;

struct addrinfo \*ptr = NULL;

struct addrinfo hints;

// Validate the parameters

if (argc != 3) {

printf("usage: %s <hostname> <servicename>\n", argv[0]);

printf(" provides protocol-independent translation\n");

printf(" from an ANSI host name to an IP address\n");

printf("%s example usage\n", argv[0]);

printf(" %s www.contoso.com 0\n", argv[0]);

return 1;

}

//--------------------------------

// Setup the hints address info structure

// which is passed to the getaddrinfo() function

ZeroMemory(&hints, sizeof(hints));

hints.ai\_family = AF\_UNSPEC;

hints.ai\_socktype = SOCK\_STREAM;

hints.ai\_protocol = IPPROTO\_TCP;

printf("Calling getaddrinfo with following parameters:\n");

printf("\tnodename = %s\n", argv[1]);

printf("\tservname (or port) = %s\n\n", argv[2]);

//--------------------------------

// Call getaddrinfo(). If the call succeeds,

// the result variable will hold a linked list

// of addrinfo structures containing response

// information

dwRetval = getaddrinfo(argv[1], argv[2], &hints, &result);

if (dwRetval != 0) {

printf("getaddrinfo failed with error: %d\n", dwRetval);

WSACleanup();

return 1;

}

printf("getaddrinfo returned success\n");

// Retrieve each address and print out the hex bytes

struct sockaddr\_in \*sockaddr\_ipv4;

struct sockaddr\_in6 \*sockaddr\_ipv6;

char ipstringbuffer[46];

DWORD ipbufferlength = 46;

int i = 1;

for (ptr = result; ptr != NULL; ptr = ptr->ai\_next) {

printf("getaddrinfo response %d\n", i++);

printf("\tFlags: 0x%x\n", ptr->ai\_flags);

printf("\tFamily: ");

switch (ptr->ai\_family) {

case AF\_UNSPEC:

printf("Unspecified\n");

break;

case AF\_INET:

printf("AF\_INET (IPv4)\n");

sockaddr\_ipv4 = (struct sockaddr\_in \*) ptr->ai\_addr;

inet\_ntop(AF\_INET, &sockaddr\_ipv4->sin\_addr, ipstringbuffer, sizeof(ipstringbuffer));

printf("\tIPv4 address %s\n", ipstringbuffer);

printf("\tport = %d \n ", ntohs(sockaddr\_ipv4->sin\_port));

break;

case AF\_INET6:

printf("AF\_INET6 (IPv6)\n");

// the InetNtop function is available on Windows Vista and later

sockaddr\_ipv6 = (struct sockaddr\_in6 \*) ptr->ai\_addr;

printf("\tIPv6 address %s\n",

inet\_ntop(AF\_INET6, &sockaddr\_ipv6->sin6\_addr, ipstringbuffer, 46) );

printf("\tport = %d \n ", ntohs(sockaddr\_ipv6->sin6\_port));

break;

case AF\_NETBIOS:

printf("AF\_NETBIOS (NetBIOS)\n");

break;

default:

printf("Other %ld\n", ptr->ai\_family);

break;

}

printf("\tSocket type: ");

switch (ptr->ai\_socktype) {

case 0:

printf("Unspecified\n");

break;

case SOCK\_STREAM:

printf("SOCK\_STREAM (stream)\n");

break;

case SOCK\_DGRAM:

printf("SOCK\_DGRAM (datagram) \n");

break;

case SOCK\_RAW:

printf("SOCK\_RAW (raw) \n");

break;

case SOCK\_RDM:

printf("SOCK\_RDM (reliable message datagram)\n");

break;

case SOCK\_SEQPACKET:

printf("SOCK\_SEQPACKET (pseudo-stream packet)\n");

break;

default:

printf("Other %ld\n", ptr->ai\_socktype);

break;

}

printf("\tProtocol: ");

switch (ptr->ai\_protocol) {

case 0:

printf("Unspecified\n");

break;

case IPPROTO\_TCP:

printf("IPPROTO\_TCP (TCP)\n");

break;

case IPPROTO\_UDP:

printf("IPPROTO\_UDP (UDP) \n");

break;

default:

printf("Other %ld\n", ptr->ai\_protocol);

break;

}

printf("\tLength of this sockaddr: %d\n", ptr->ai\_addrlen);

printf("\tCanonical name: %s\n", ptr->ai\_canonname);

}

**Hand in**

1. Your program GetHostName.cpp, GetHostAddress.cpp
2. GetHostInfo.pdf showing (with screen shots)
   1. your local host name and IP address.
   2. the host information for IP addresses xxx.xxx.xxx.xx.
   3. the IP address of given host names (e.g. www.sjtu.edu.cn)
   4. other interesting results you wish to show.

**Exercise 2**. (Echo)

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.

Hint: modify the presented stream based TCP client (TCPClient.cpp) and server programs (TCPServer.cpp) to transfer multiple messages back and forth (until the client terminates the connection).

Client side (project name:EchoClient)

char sendbuf[DEFAULT\_BUFLEN];

// Loop until “quit” is entered

while(1)

{

// Type the message

gets\_s(sendbuf);

// Bail out if "quit" is entered

if (strcmp(sendbuf, "quit") == 0)

break;

//send the message to the echo server

send(ConnectSocket, sendbuf, int(strlen(sendbuf)+1), 0);

// Receive from the server and print the message on the screen

recv(ConnectSocket, recvbuf, recvbuflen, 0);

printf("Received: %s \n", recvbuf);

}

Server side (project name:EchoServer)

// Loop until client terminates connection

do

{

// Receive from the client, and bail out if client shut down

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

if (iResult > 0)

{

printf("Received: %s \n", recvbuf);

// Echo the buffer back to the sender

send( ClientSocket, recvbuf, iResult, 0 );

}

else if (iResult == 0)

printf("Connection closing...\n");

else {

printf("recv failed with error: %d \n", WSAGetLastError());

closesocket(ClientSocket);

WSACleanup();

return 1;

}

} while (iResult > 0);

Hand in

1. EchoClient.cpp, EchoServer.cpp
2. Echo.pdf showing (with screen shots)
   1. The execution of EchoClient and EchoServer
   2. Other interesting results you wish to show

**Exercise 2a**. (EchoServer2)

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.

#include <string.h> // Needed for memcpy() and strcpy()

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

Add these lines before main()

//----- Globals ---------------------------------------------------------------

int Count; // Thread counter

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

void do\_service(void \*client\_s); // Thread function

In the main () function add the following variables.

unsigned int client\_s; // Client socket descriptor

struct sockaddr\_in client\_addr; // Client Internet address

struct in\_addr client\_ip\_addr; // Client IP address

int addr\_len; // Internet address length

char ipstringbuffer[46];

Add the following lines after a socket is created and is put to listening state.

// Main loop (Loop forever)

Count = 0;

while (1)

{

Count++;

printf("Count=%d \n",Count);

// Accept a connection. The accept() will block and then return with

// client\_addr filled-in.

addr\_len = sizeof(client\_addr);

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

// Copy the four-byte client IP address into an IP address structure

// - See winsock.h for a description of struct in\_addr

memcpy(&client\_ip\_addr, &client\_addr.sin\_addr.s\_addr, 4);

// Print an informational message that accept completed

printf("Connection %d accepted!!! \n", Count);

inet\_ntop(AF\_INET, &client\_ip\_addr, ipstringbuffer, sizeof(ipstringbuffer));

printf("\tClient socket number: %d\n", client\_s);

printf("\tIPv4 address: %s\n", ipstringbuffer);

printf("\tPort nuber: %d\n", ntohs(client\_addr.sin\_port));

if (\_beginthread(do\_service, 4096, (void \*)client\_s) < 0)

{

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

exit(1);

}

}

// Never reached!!!

// Wait for all threads to finish

while(Count);

// Close open sockets

closesocket(ListenSocket);

// This stuff cleans-up winsock

WSACleanup();

}

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

//= Thread function to service a single client =

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

void do\_service(void \*client\_s)

{

char in\_buf[1024]; // Input buffer for response

printf("thread beninging... \n");

// Loop until client shut down

while(1)

{

// Receive from the client

if (recv((unsigned int)client\_s, in\_buf, sizeof(in\_buf), 0) == 0)

break; // when client shut down

printf("Received from client... data = '%s' \n", in\_buf);

// Echo the received message to the client

send((unsigned int)client\_s, in\_buf, (strlen(in\_buf) + 1), 0);

}

printf("thread completed... \n");

// Decrement for a completed thread

Count--;

// Close all open sockets and end the thread

closesocket((unsigned int)client\_s);

\_endthread();

}

Hand in

1. EchoServer2.cpp
2. EchoServer2.pdf showing (with screen shots)
   1. The execution of EchoServer2 with multiple echo clients
   2. Other interesting results you wish to show

**Exercise 3**. (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. WebClient.pdf 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 4**. (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. WebServer.pdf showing the execution of your program and other interesting results.

**Exercise 4a**. (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. WebServer2.pdf showing the execution of your program and other interesting results.

**Exercise 5. (**UdpChatting**)**

Write a chatting program using datagram socket and UDP protocol. People using your chatting program can send message to each other, as long as the IP address and Port number are known to each other, and the program is running.

**Hand in**

1. UdpChattig.cpp
2. UdpChatting.pdf showing the execution of your program and your thoughts and discussion.

**Exercise 6**. (myping)

Write a program to test the reachability of an Internet interface identified by an IP address or name. (The basic function of “ping” command)

Hints: Send an ICMP “echo request” to the destination, an ICMP “echo reply” will be sent back if the destination is reachable. (refer to RFC 792 for more information about ICMP)

1. Create a raw socket：socktype=SOCK\_RAW, protocol=IPPROTO\_ICMP;
2. Construct an ICMP message;
3. Use “sendto” to send the ICMP message to the remote machine;
4. Use “recvfrom” to receive any response.

**Hand in**

1. myping.cpp
2. myping.pdf
   1. the functions of your program and how to use them
   2. theory and protocols used in your program
   3. flow-chart of your program （showing the main APIs）
   4. execution of your program (screen shots)
   5. thoughts and discussion