CS 2690 Programming Assignment 1
Using Microsoft Visual C++ and Windows 7, 8.1 or 10

Windows Sockets TCP Echo Client in C

Approximate time required: 3 - 5 hours

Introduction

For the first programming assignment, you will code a console-based client application in C that uses Windows Sockets ("Winsock") to transmit a message to a server. The server will echo the message, which will then be received and displayed by the client. A working server will be provided for testing, along with a program template that you can use as the basis for your client. You will add the Winsock function calls and related code to the client template. All relevant files can be found in Canvas, under `/Programming Assignments/Program 1/`. Note that a working Program 1 will be required in order to complete Program 2.

You will use Microsoft Visual C++ for all CS 2690 programming assignments. These assignments assume that you are using Visual Studio Community or Visual Studio Enterprise 2017 or 2019. If you don't have access to a Windows 7, 8.1 or 10 computer, you can use one of the computers in the Network Lab (CS 516) to code, compile and test. CS 516 computers run DeepFreeze and revert to their original state each time they reboot, so be sure to build your project on a flash drive that you can take with you when you leave the lab. The first time you use Visual Studio in the Network Lab, it will take a few minutes to set up the configuration for your user. Any UVU computer lab that has Visual Studio can be used for coding, but only CS 516 has all of the required test software.

Program 1 will require approximately 130 lines of code and the logic is straightforward. However, there are many potential pitfalls in low-level C programming. Start right away to allow plenty of time for debugging. Make sure that you understand each data structure and function call, and test carefully.

Why C?

So why are we using C? Java, C++, C# and Python all provide perfectly good socket classes that are relatively easy to use – at least when everything works properly. When it doesn't work, however, these classes obscure the Winsock API. The socket classes in all of the aforementioned languages are based on Berkeley Sockets (UNIX, Linux & Mac OS), or on the Windows Sockets API. As often as not, those classes are coded in C (especially in the *NIX environment). Communications protocols such as TCP and IP are also coded in C or C++ in most operating systems. So it's worth knowing something about C, which remains an important programming language. (See `https://www.tiobe.com/tiobe-index/` for one compilation of current programming language rankings.)

In this course, we'll use the Windows Sockets ("Winsock") API for the programming assignments. Windows and Berkeley Sockets both provide low-level access to socket calls, along with the option of creating separate threads, which is important for programming servers. Programming in C will allow you to see exactly what occurs as your client and server connect and communicate. You'll also get a feel for how socket-related data is converted from local "host" format into the "network" format required by both versions of sockets. And because the socket classes in other languages are based on this API, you'll gain an understanding of what is going on "under the hood" in future network programs that you may design in other languages.

Evaluation of Programming Assignments

The three CS 2690 programming assignments are worth a total of 10% of your semester grade. Each programming assignment will be scored on a 10 point scale. Your submission will be evaluated as follows:

Program meets all specifications	10 points
Program works properly, but does not meet all specifications	9
Program is generally correct, but has minor functional problems	7 – 8
Part of the program is correct, but has major functional problems	4 – 6
Partially correct source code, does not compile (no .exe submitted), or source code is difficult to read	1 – 3
Source code doesn't match .exe, or is unreadable	0
Program is virtually the same as another student's submission or otherwise plagiarized	0

As noted in the syllabus, submitting material(s) obtained from the Internet or from another student, as one's own work, without proper acknowledgment of the source, is a flagrant offense. Detecting such plagiarism is trivial. *If you submit code that you did not write, you will fail the course.* Posting programming assignment solutions to public Internet sites is also considered to be cheating by the Computer Science Department. If you use GitHub or another version control site, put your code in a private repository.

Compiling a C Program with the Visual C++ IDE

Before you begin writing this program, read `Compiling a C Winsock Program in Visual Studio Community 201*.pdf` (for your version of Visual Studio), found in Canvas under `/Programming Assignments/Program 1/`. C programs that use Windows Sockets won't compile properly unless you make the changes to your Visual Studio settings described in these documents.

You will also want to refer to the PowerPoint slides that discuss TCP and client Sockets programming. That includes Lectures 10 and 11, and Lecture 12 slides through 12-5 only (not the Server Sockets slides that follow). You will find the Sockets discussion on textbook pages 36 – 44 helpful as an introduction, however the C code examples in the text use Berkeley Sockets, which is somewhat different than Windows Sockets. *Use the PowerPoint slides for specific Windows Sockets code examples.* For greater detail on all of the Winsock functions, see `https://msdn.microsoft.com/en-us/library/windows/desktop/%20ms741394(v=%20vs.85).aspx`.

Source and Executable Files Provided

`Program 1 Template.c` in the `…/Program 1/` folder contains an outline of the echo client program. The comments in this file outline the sequence of events and some pitfalls to avoid. References to all required function prototypes and header (`.h`) files are included, and can be used as-is. You can rename this file to `WSEchoClientv6.c` and edit it, adding the C code required to create the working echo client. Remove any irrelevant comments from your version of `WSEchoClientv6.c` and follow the CS 1410 style guidelines found in `stlye.pdf` (also in the `…/Program 1/` folder). If your source code is difficult to read (messy formatting,, etc.), it won't be graded.

The C language does not provide built-in exception handling. `Program 1 Template.c` contains the function prototype for an error handling function called `DisplayFatalErr()` which can be used as-is. A complete working source code version of the `DisplayFatalErr()` file is provided for you in `DisplayFatalErr.c`. Use `DisplayFatalErr.c` to handle any errors that occur *after the Winsock DLL has been successfully initialized* in `WSEchoClientv6.c`. Be sure that the source code files for `DisplayFatalErr.c and WSEchoClientv6.c` are both included under your project's *Source Files* in Solution Explorer (see Figure 1 below), in two separate `*.c` files.

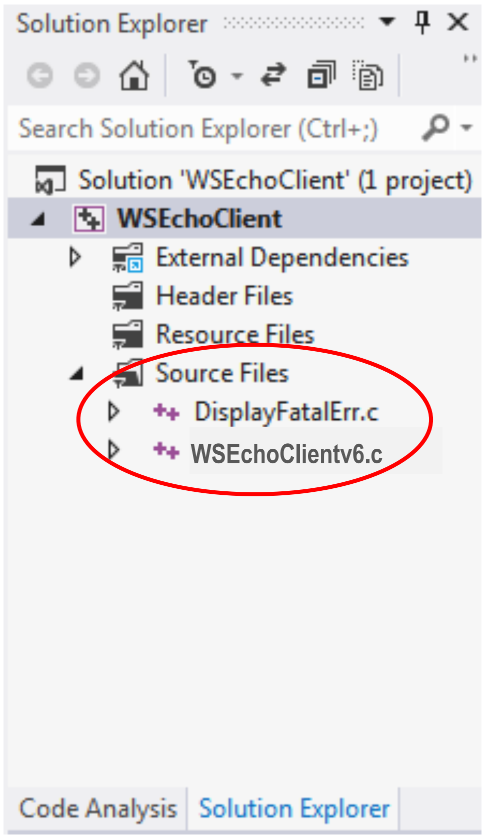


Figure 1. Solution Explorer with Project Containing Two Source Code Files

`WSEchoServerv6.exe` (also found under `/Programming Assignments/Program 1/` in Canvas) is the echo server that your client will communicate with. For Program 2, you will code your own version of this server. This version was compiled with Visual Studio Community 2017 under Windows10. *Download `WSEchoServerv6.exe` from Canvas and verify that it will run from the Windows command line on your computer.* (To open a command line window in Windows 8.1 or 10, use WINDOWS KEY+X, then click **Command Prompt**.) After changing to the folder where `WSEchoServerv6.exe` is located, test it by typing:

```
WSEchoServerv6 5000
```

If the server is running, Windows should display:

```
DH's IPv6 Echo Server is ready for client connection...
```

Type CTRL+C to terminate the server. If it won't run, see the section titled **Testing Your Echo Client** on page 6 for additional help.

Program Specifications

Your echo client will be implemented as a Windows command line program in C (not C++). It will obtain its input data from command line arguments, as follows:

```
WSEchoClientv6 <server IPv6 addr> <server port> <"Msg to echo">
```

For example (using the IPv6 loopback address):

```
WSEchoClientv6 ::1 5000 "Is anyone out there?"
```

Note that when your program retrieves its arguments from the command line, the program name itself is the first argument, at index 0 (`argv[0]`). The IPv6 address is index 1, etc. Arguments are delimited by blanks, so use double quotes around any individual argument that contains embedded blanks, as shown with the message to be echoed above.

All network communication will be done using the Windows Sockets API. *Programs that use C++ socket classes will not be graded.*

Edit the program header comments in `Program 1 Template.c` to list your name, section, date, Windows version and Visual Studio version. The comments containing the declaration on cheating must be included in the code that you submit.

Include the header (`.h`) files required to support Windows Sockets, as given in `Program 1 Template.c`. Use C or C++ style comments to document each call to the Winsock API. For example:

```
/* A C style comment may extend across multiple lines, and must
be terminated */

// A C++ style single-line comment requires no terminator. Use
// either or both types of comments as desired.
```

Code the following required procedural activities in the order listed. (These are indicated with comments in `Program 1 Template.c`.) You may want to reference that file while reading this section.

1. Verify the correct *number* of command line arguments have been entered by the user. (See the Lecture 12 slide titled "C Command Line Arguments and Finding Length".) You don't need to validate the *content* of the arguments, just count them.

2. Initialize the Winsock DLL.

3. Create a TCP socket that uses IPv6.

4. Convert the server information obtained from the command line arguments into the proper big-endian sockets format and load it into the `sockaddr_in6` structure. See the Lecture 11 slides titled “Populating sockaddr_in6 with Server Info” and “Converting the Port Number to Internet Format”.

5. Establish a TCP connection with the server (`WSEchoServerv6.exe`). Both client and server will be running on the same computer.

6. Transmit the message entered on the command line to the server. TCP will not transmit a message of length 0. Check for that, so that your client’s `recv()` call does not hang waiting for an echoed message from the server that will never arrive. Note that you are transmitting a byte stream, *not a C text string.* Make sure that the null string terminator character (`'\0'`) passed in with your test message from the command line is not transmitted to the server. See the Lecture 12 slide titled “C Command Line Arguments and Finding Length” for more information on how to do this.

7. Receive (read) the *entire message* after it is echoed back from the server. Count the total number of bytes received and verify that you have received the same number of bytes that you previously transmitted to the server. Design your client so that a big message won’t overflow your receive buffer. Test this by using a tiny receive buffer in your client (3 or 4 bytes) and sending a message larger than that size, thus forcing your client to call `recv( )` multiple times to receive the entire message. Bear in mind that calling `recv( )` simply transfers bytes from the TCP receive buffer to your application’s buffer. It doesn’t cause additional data to be transmitted over the network.

8. Display the echoed message on the console. You can display each part of the original message individually as you read it, or combine the parts and display the entire message after the final call to `recv( )`. See the Lecture 11 slide titled “recv( )” for details.

9. The TCP protocol allows either the client or server to close the connection. In this program, your client must close the TCP connection. (The test server won’t do it.)

10. Release Winsock DLL resources (“cleanup”) and exit.

11. For three extra credit points, enhance your WSEchoClientv6 so that it can also accept a dotted decimal IPv4 address and connect to an IPv4 server. Test this feature using `WSEchoServerv4.exe` (found in the same folder in Canvas). This will require your client to send IPv4 packets to the IPv4 server when you enter an IPv4 address on the command line. When you enter an IPv6 address with *the same client program*, it will connect to `WSEchoServerv6.exe` using IPv6 packets. (Make sure the appropriate server is running when you test.) Adding this feature isn’t trivial, but it doesn’t take very much additional code. Some research will be required. If you do this, include a comment when you submit your program noting that it supports IPv4 addresses, so that the grader will know to test for it. Make sure that you have the basic version of this program working properly before you complicate it with the extra credit feature.

Testing Your Echo Client

Code your echo client, build the solution in Visual Studio, and debug any compiler errors.

Download `WSEchoServerv6.exe` from `/Programming Assignments/Program 1/` in Canvas and store it in the folder of your choice.

We will test your client and server by running them both on the same computer, in loopback mode. Testing is most easily done by running the client and server from within two separate command line windows, rather than from within the debugger (although testing your client with the debugger may be helpful). Note that the client and server executables require the C++ .NET libraries, which you should have previously installed with Visual Studio.

After changing to the folder where you copied the server (`cd C:\Users\`... or whatever), start `WSEchoServerv6.exe` using the following command line syntax. Choose a server ephemeral port number between 1024 and 65535, for example:

```
        WSEchoServerv6 5000
```

The server's IP address that the client will connect to will be the IPv6 loopback address `::1`. ('`::1`' is IPv6 shorthand for 15 bytes of zeros and a hex 01 in the 16th byte.) Run the client on the same machine, for example:

```
        WSEchoClientv6 ::1 5000 "This is my test message"
```

As the client runs, respond to the `printf()` prompts on the console and verify that your client works correctly. After this test, leave your client and server command line window(s) open for use in the next section.

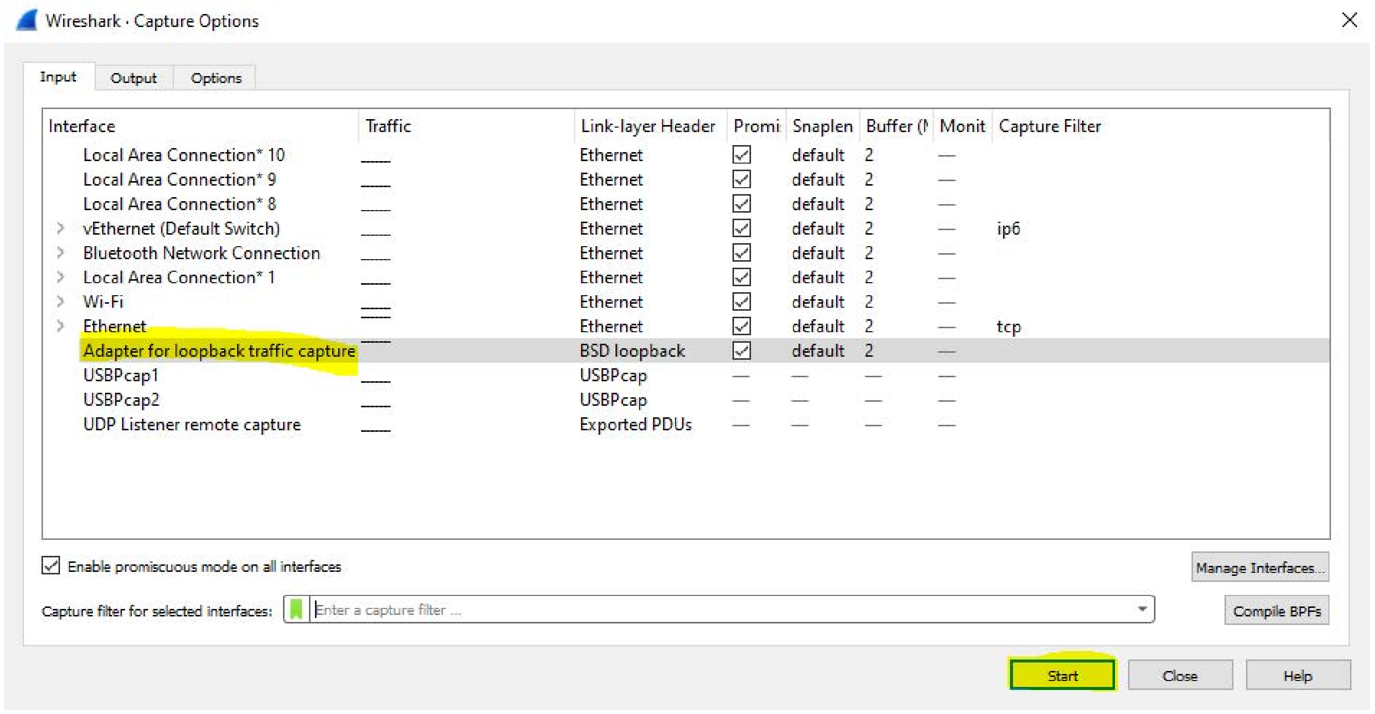
Testing With Wireshark

Wireshark didn't originally support monitoring traffic on local host. (Originally, wireshark only captured actual data sent and received from a network adapter.) However, since WireShark 3, it has included npcap, see https://www.wireshark.org/docs/wsug_html_chunked/ChBuildInstallWinInstall.html and search for npcap. Npcap allows wireshark to monitor and capture localhost/loopback traffic.

Debugging Sockets programs can be tricky. Read through the remainder of this section completely before starting the server and client.

The following procedure assumes loopback mode, with the client and server running on the same computer in separate command line windows.

Start WSEchoServerv6.exe as before (if necessary). Then, using wireshark, choose to capture for the "Adapter for loopback traffic capture", see below:


Wireshark · Capture Options

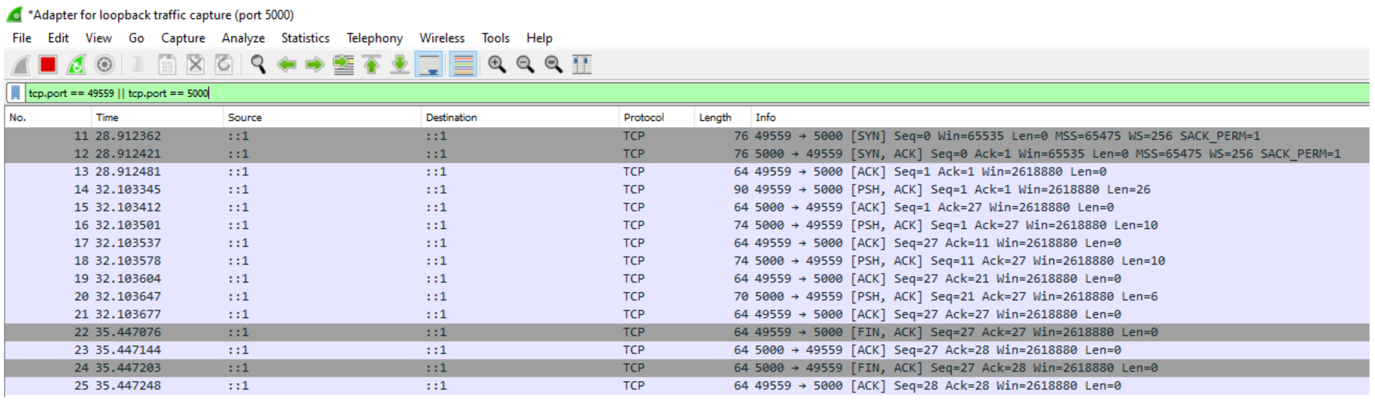
Input | Output | Options

Interface	Traffic	Link-layer Header	Promi	Snaplen	Buffer (I	Monit	Capture Filter
Local Area Connection* 10		Ethernet	✓	default	2	—	
Local Area Connection* 9		Ethernet	✓	default	2	—	
Local Area Connection* 8		Ethernet	✓	default	2	—	
vEthernet (Default Switch)		Ethernet	✓	default	2	—	ip6
Bluetooth Network Connection		Ethernet	✓	default	2	—	
Local Area Connection* 1		Ethernet	✓	default	2	—	
Wi-Fi		Ethernet	✓	default	2	—	
Ethernet		Ethernet	✓	default	2	—	tcp
Adapter for loopback traffic capture		BSD loopback	✓	default	2	—	
USBPcap1		USBPcap	—	—	—	—	
USBPcap2		USBPcap	—	—	—	—	
UDP Listener remote capture		Exported PDUs	—	—	—	—	

✓ Enable promiscuous mode on all interfaces

Capture filter for selected interfaces: Enter a capture filter ...

Run the client completely to connect, send, receive, and disconnect. If your capture includes multiple attempts, then, in the server's output, note the server's client and server ports. For example, the last time I ran WSEchoServerv6.exe, the server gave the following output, "Processing the client at ::1, client port 49559, server port 5000" in the adapter capture filter, enter: tcp.port == <CLIENT PORT#> || tcp.port == <SERVER PORT#>. Therefore, continuing with the example above, the filter would be: "tcp.port == 49559 || tcp.port == 5000" (without the quotes, see screenshot below):


*Adapter for loopback traffic capture (port 5000)

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.port == 49559 || tcp.port == 5000

```
No.   Time        Source   Destination   Protocol   Length   Info
  11 28.912362    ::1      ::1           TCP          76 49559 → 5000 [SYN] Seq=0 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM=1
  12 28.912421    ::1      ::1           TCP          76 5000 → 49559 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65475 WS=256 SACK_PERM=1
  13 28.912481    ::1      ::1           TCP          64 49559 → 5000 [ACK] Seq=1 Ack=1 Win=2618880 Len=0
  14 32.103345    ::1      ::1           TCP          90 49559 → 5000 [PSH, ACK] Seq=1 Ack=1 Win=2618880 Len=26
  15 32.103412    ::1      ::1           TCP          64 5000 → 49559 [ACK] Seq=1 Ack=27 Win=2618880 Len=0
  16 32.103501    ::1      ::1           TCP          74 5000 → 49559 [PSH, ACK] Seq=1 Ack=27 Win=2618880 Len=10
  17 32.103537    ::1      ::1           TCP          64 49559 → 5000 [ACK] Seq=27 Ack=11 Win=2618880 Len=0
  18 32.103578    ::1      ::1           TCP          74 5000 → 49559 [PSH, ACK] Seq=11 Ack=27 Win=2618880 Len=10
  19 32.103604    ::1      ::1           TCP          64 49559 → 5000 [ACK] Seq=27 Ack=21 Win=2618880 Len=0
  20 32.103647    ::1      ::1           TCP          70 5000 → 49559 [PSH, ACK] Seq=21 Ack=27 Win=2618880 Len=6
  21 32.103677    ::1      ::1           TCP          64 49559 → 5000 [ACK] Seq=27 Ack=27 Win=2618880 Len=0
  22 35.447076    ::1      ::1           TCP          64 49559 → 5000 [FIN, ACK] Seq=27 Ack=27 Win=2618880 Len=0
  23 35.447144    ::1      ::1           TCP          64 5000 → 49559 [ACK] Seq=27 Ack=28 Win=2618880 Len=0
  24 35.447203    ::1      ::1           TCP          64 5000 → 49559 [FIN, ACK] Seq=27 Ack=28 Win=2618880 Len=0
  25 35.447248    ::1      ::1           TCP          64 49559 → 5000 [ACK] Seq=28 Ack=28 Win=2618880 Len=0
```

Take, and submit, a few screenshots from your capture:

1. Show the syn and the syn-ack establishing the connection.
2. Show the data that was sent by the client to the server. You do this y selecting the push, ack packets that have data at the end of the packet. Then you click on the data (note: it will be in hex), then you right click and say copy | ...as printable text. If you paste into notepad you can see the extended ascii message data as text. Show a screenshot of the data in hex. Using a tool like paint, add the translation of the printable chars.
3. Using a screen capture, show that the data echoed from the server back to the client is the same raw bytes that the client sent.
4. Show the fin and fin, ack

Submitting Your Program to Canvas

Before submitting your program, review the preceding Program Specifications section to confirm that your echo client program meets all requirements. The grader will not compile your code. If your program does not execute from the Windows command line, or if an executable is not provided, you can earn no more than 30% of the possible points. If you have compiled your program in Debug mode, recompile it in Release mode before you submit it. See `Compiling a C Winsock Program in Visual Studio Community 2019.docx` page 6 for instructions on how to compile in Release mode.

Combine your source code (`*.c`), executable (`*.exe`), and your *TCP Spy* screen capture into a single zip file. *Do not submit any other files and do not submit the entire Visual Studio project.* Name the zip file with the program number and your initials, for example `prog1dbh.zip`. Upload the zip file to Canvas. Be sure to include all the files listed in the Summary of Deliverables below. Do not include `DisplayFatalErr.c` or `WSEchoServerv6.exe`.

If you have disabled your firewall, be sure to enable it after testing is complete.

Summary of Deliverables

- `WSEchoClientv6.c` (based on `Program 1 Template.c`)
- `WSEchoClientv6.exe` (compiled in Release mode)
- Your WireShark screen captures in `.pdf`, `.doc`, `.docx`, `.odt` or `.rtf` format
- All of the above contained in a single zip file named `prog1***.zip`, where *** is your initials
- **Don't** submit `DisplayFatalErr.c`, `WSEchoServerv6.exe`, or any other files
- **Don't** submit the entire Visual Studio project