CS 2690 Programming Assignment 3
Using Microsoft Visual C++ and Windows 7, 8.1 or 10

Windows Sockets UDP Echo Client and Server in C

Approximate time required: 3 - 4 hours

Introduction

For the third programming assignment, you will modify your Program 1 TCP Echo Client and your Program 2 TCP Echo Server to use the UDP protocol. The basic functionality will remain the same – send a message and echo it back. However, the TCP stream sockets will be replaced by UDP datagram sockets. All related files can be found in Canvas under `/Programming Assignments/Program 3/`.

As before, you will use the Microsoft Visual C++ compiler in one of recent versions of Visual Studio. If you don't have access to a Windows environment, you can use one of the computers in the Network Lab (CS 516) to code, compile and test. For testing, we will run both client and server on the same computer, in loopback mode. To debug your client and server, you may need to run Wireshark v3.0.x on the same computer.

There is no specific discussion of UDP sockets in the textbook, so you will need to refer to the Lecture 14 slides that cover UDP Sockets programming (14-3 through 14-9). Note that it is not sufficient to simply replace the TCP/stream sockets with UDP/datagram sockets. Redesign is required for both the client and server.

Evaluation of Programming Assignments

The three programming assignments are worth a total of 10% toward your semester grade. Each programming assignment will be worth 10 points. This submission, which will include both the client and server, 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, 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

Program Files

Name your source code files `WSEchoClientUDPv6.c` and `WSEchoServerUDPv6.c`. (There will be no file equivalent file to `ProcessClient.c` in Program 2.) Use the `DisplayFatalErr.c` module from Program 1 to handle errors in both programs. For this assignment, executables will not be provided for testing.

Program Specifications

As with the TCP client and server, both UDP programs will be implemented as command line programs in C. They will obtain their input data from command line arguments, as follows:

```
WSEchoServerUDPv6 <server port>
```

For example:

```
WSEchoServerUDPv6 5000
```

and…

```
WSEchoClientUDPv6 <server IPv6 addr> <server port> <"Msg to echo">
```

For example:

```
WSEchoClientUDPv6 ::1 5000 "Is anyone out there?"
```

You may design the server to use a default port number if none is provided as a command line argument. Use a port number above the range of well-known port numbers (1 – 1024).

All network communication will be achieved using the Windows Sockets API. Do not use C++ socket classes.

Include the program number, your name, section, date, Windows version and Visual Studio version in the program header comments. The comments containing the declaration on cheating must be included.

Include any header (`.h`) files required to support Windows Sockets.

Use C or C++ style comments to document each call to the WinSock API. Use the CS 1410 style guidelines described in `stlye.pdf` (located in the `/Programming Assignments/Program 1/` folder in Canvas).

Feel free to insert additional `printf()` and `getchar()` statements into your server code for debugging purposes, *but you must remove them and recompile before submitting your program to Canvas,* so that the grader doesn't have to press ENTER repeatedly to get your server to accept a client connection and echo the message. Failing to do this may lower your score.

The following procedural activities are required.

1. Verify the correct number of command line arguments have been provided by the user on the client and server command lines. You don't need to validate the content of those arguments. Use the chosen default port if desired.

2. Display a message on the server console that includes your initials, similar to this:

   ```
   JD's IPv6 echo server is ready for UDP client...
   ```

3. For this assignment, your client may leave the null terminator (`'\0'`) at the end of the message when it is transmitted to the server, instead of removing it as we did with the TCP client. Note that if you leave the null terminator in place, the length of the transmitted message will be one byte longer than what `strlen()` returns, and both the client and server will need to adjust for that.

Using a null terminator or other marker to indicate the end of the message is one way that the application layer can be notified that it has received the entire message. Adding a message header containing the message length would be another, as would using a fixed message length, where the length is known in advance to both client and server.

4. When a client sends a message to the server, your server will display the IP address and port number of the client, and the server's own port number (from the server's `sockaddr_in6`), as shown below:

   ```
   Processing the client at ::1, client port 12345, server port 6789
   ```

   On the server, the client's IPv6 address will be stored in the client `sockaddr_in6` structure by `recvfrom()`, after it returns. You can use `inet_ntop()`, which is similar to `inet_pton()`, to convert that 16-byte numeric big-endian address into a text string for display. (See slide 14-8 and https://docs.microsoft.com/en-us/windows/desktop/api/ws2tcpip/nf-ws2tcpip-inetntopw for details.) Verify that all numeric values are displayed correctly.

5. As with Program 2, the server will echo the entire message back to the client. Unlike Program 2, when the Program 3 server's call to `recvfrom()` returns with the message from the client on the socket, that same socket can be used to echo the message back to the client with `sendto()`. A second socket is not needed with this simple server because there is no client socket handle being returned by a call to `accept()` (see slide 14-8).

   You will need to check for errors in the server's receive process. All server functions except for calls to `DisplayFatalErr()` will be handled within `main()`. Unlike Program 2, you will not call a `ProcessClient` function in a separate server module to echo the client's message.

6. Note that with UDP, the server does not need to call `recvfrom()` repeatedly to ensure that the entire message from the client has been received. Single calls to `recvfrom()` and `sendto()` will handle the echo process for one message on the server. As with your TCP server, this UDP server must be able to echo multiple messages from clients (one at a time), so place the calls to `recvfrom()` and to `sendto()` in a `for (;;)` "forever" loop, as you did with the TCP echo server.

7. Your server can assume that interaction with one client is finished as soon as it completes one `recvfrom()` and `sendto()` sequence. Unlike the TCP server, your UDP server can't watch for a return value of 0 indicating a close (FIN) from the client, since UDP doesn't open and close connections.

8. Before you display the echoed message on the client console, your client must verify that the server which sent the message is really the echo server. Do this (on the client) by comparing the server's IP address in the `sockaddr_in6` returned by the client's `recvfrom()` with the server's address used by the client's `sendto()`. That will necessitate using two different `sockaddr_in6` structures on the client. (Those are named `serverInfo` and `fromAddr` on slides 14-5 and 14-6, respectively.) Again, note that both `recvfrom(0` and `sendto()` can use the same socket.

   If the two server addresses that you compared are different, generate an error message on the client. Bear in mind that the IPv6 addresses stored in `sockaddr_in6` are not text strings, they are 16 byte numeric fields. You might want to investigate the `memcmp()` function as a way to compare them.

The client must also verify that the number of bytes received back from the server equals the number of bytes that it originally sent to the server. Remember to account for the null terminator, if necessary. Since there is no way for the server to predict the client in advance, your server won't need to verify the client's IP address or message length.

9. While waiting in the "forever" loop for client communication, the server can be terminated with CTRL+C. No special programming is required to enable this. Because we are using this crude termination mechanism for the server, no call to `closesocket()` is needed on the server to release the socket, and no call to `WSACleanup()` will be needed to release the WinSock DLL resources. Those functions will be handled by the operating system. (Omitting `closesocket()` and `WSACleanup()` would be considered poor design in a real sockets program.)

Testing Your UDP Echo Client and Server

Design and code your UDP echo client and server, build the solutions, and debug any compiler errors. After your programs compile successfully, test them in loopback mode. Use the debugger if needed, and remember that with the debugger, you must enter command line parameters in advance (see `Compiling a C Winsock Program in Visual Studio Community 201*.pdf`.)

Verify that your server echoes the message back to your client, and that multiple client messages can be echoed while the server continues to run.

Debugging in Wireshark

If your client is not receiving the echoed message, try capturing the exchange with Wireshark and verifying the transmission of your UDP packets, as we did in Program 2.

If you can't see your UDP/IPv6 packets amongst all of the other traffic, try setting your display filter to `ipv6.addr == ::1`.

If Wireshark is not capturing any loopback traffic, go to **Control Panel | Network and Sharing Center | Change adapter settings**, verify that you see *Npcap Loopback Adapter*, and make sure it is enabled. Additional suggestions can be found on page 6 of `Program 2.pdf`.

Submitting Your Programs to Canvas

Before submitting your programs, review the Program Specifications section to confirm that your programs meet the requirements. The grader will not compile your code. If your programs do not execute from the Microsoft 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 any other required files 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 `prog3dbh.zip`. Once you have zipped together your files, upload the zip file to Canvas. Be sure to include all the files listed in the Summary of Deliverables below.

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

Summary of Deliverables

- `WSEchoClientUDPv6.c`
- `WSEchoClientUDPv6.exe` (compiled in Release mode)
- `WSEchoServerUDPv6.c`
- `WSEchoServerUDPv6.exe` (compiled in Release mode)
- All of the above contained in a single zip file named `prog3***.zip`
- Do not submit `DisplayFatalErr.c` or any other files
- Do not submit the entire Visual Studio project