VANIER COLLEGE – Computer Engineering Technology – Autumn 2021

Network Systems Design (247-509-VA)

Leonardo Fusser (194995)

LABORATORY EXPERIMENT 3

UDP Implementation Using Winsock

NOTE:
To be completed in 2 lab sessions.

You are required to perform demo as stated in the lab. No formal lab report is required. Work must be submitted by the deadline specified in Lea.

Your report should include the following :

1. Final source codes well commented.
2. Explanation on changes in code, as required in the lab.
3. Screen shots with explanation, as required in the lab.
4. Discussion and conclusion.

This exercise is to be done individually except where specified in the procedure. **Each** student must submit a lab report with original observations and conclusions.

OBJECTIVES:

After performing this experiment, the student will be able to:

1. Perform networking socket programming using Winsock on Windows platform.
2. Perform basic client and server communication via UDP between 2 PCs.

THEORY

1. PowerPoint presentation of the class
2. http://www.codeproject.com/Articles/11740/A-simple-UDP-time-server-and-client-for-beginners

PROCEDURE

Part A: Send UDP packets (Client side)

1. Obtained the basic programming files for the lab. Create a new project in Microsoft Visual Studio for a *console* program that will act as *UDP client*.
2. Modify the code to send a UDP packet to a *random IP address* within your network. Briefly explain the main code changes done.
   - The main changes done to the code are as follows:

     The approach that was followed to test sending a UDP packet (that contains a unique hard-coded message) to a fictitious server was done in the following sequential manner:
     - Step 1: initialize Winsock (so that unique APIs can be used).
     - Step 2: create datagram socket.
     - Step 3: setup info of fictitious server (IP address, port number, etc…).
     - Step 4: send data (UDP packet) to fictitious server (using sendto() function).
     - Step 5: when finished sending data, shut-down datagram socket…
     - Step 6: then close datagram socket…
     - Step 7: then clean up and quit.

     The data that was sent to the fictitious server was just an array (called "SEND") that contained a hard-coded message that had the string "THIS IS A TEST!" stored in it. The "sendto()" function had this array passed as one of the main arguments to the function.

     The result from command prompt after executing this code is shown below in "Figure 1".

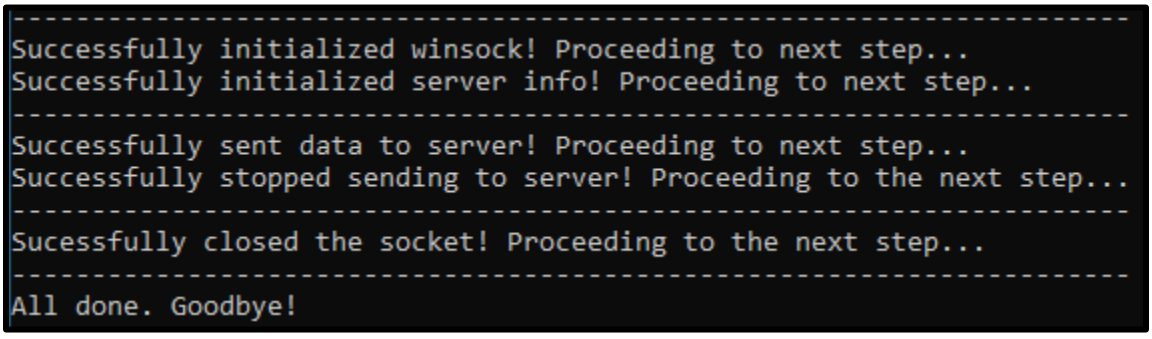


Figure 1. Command prompt result output shown for test sending UDP packet to fictitious server.

3. Use Wireshark capture to verify the functionality of your program.

   - The resulting test UDP packet that was sent out from the client, which contains a hard-coded message, can be found and analyzed using a Wireshark packet capture as shown in "Figure 2" below. From the result shown below, we can see all the various details of the test UDP packet that was sent out from the client (functionality explained in previous question). Here, we can see details such as the destination IP address (the fictitious server), destination port number (60001) along with the data that was contained in the UDP packet (the hard-coded message that reads "THIS IS A TEST!"). We can also see the originating details of this test UDP packet.

     As it was explained in a previous lab, there are obvious security flaws here, since any one with basic computer networking understanding could load a packet analyzer (such as Wireshark) to observe and potentially take advantage of these security flaws.

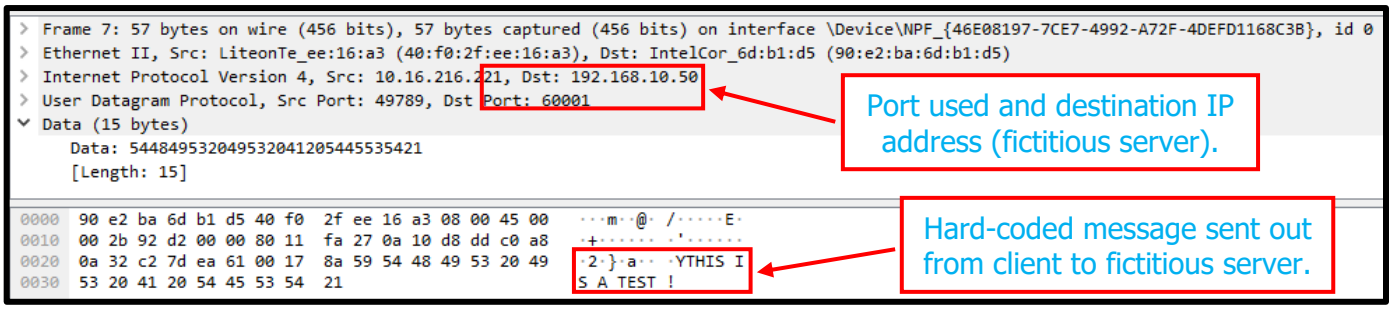


Figure 2. Wireshark packet capture verifying test sending UDP packet to fictitious server.

4. Modify the code to use argv[] to allow customized long messages. Again, use Wireshark to verify your program. Perform a screen shot of your Wireshark capture. Clearly illustrate and explain the features setup, and data transmitted.

   - The main changes done to the code are as follows:

     Similar to the previous approach, this new sequential approach was followed for sending a UDP packet (which contains a unique customized long message) to a fictitious server:

     - Pre-Step 1: check and take arguments passed in command prompt to store as a resulting string in a buffer called "SEND".
     - Step 1: initialize Winsock (so that unique APIs can be used).
     - Step 2: create datagram socket.
     - Step 3: setup info of fictitious server (IP address, port number, etc…).
     - Step 4: send data (UDP packet) to fictitious server (using sendto() function).
     - Step 5: when finished sending data, shut-down datagram socket…
     - Step 6: then close datagram socket…
     - Step 7: then clean up and quit.

     Unlike in the previous approach, the data that was sent to the fictitious server this time around was a resulting array (called "SEND") that contained the customized long message that was entered in command prompt when the program was executed. A for loop was used to take the arguments inputted by the user ("argv[]") in the command prompt and store them as a resulting string in the "SEND" array. "argc" was also used to track down how many arguments were inputted by the user in the command prompt. Similar to the previous approach, the "sendto()" function had this array passed as one of the main arguments to the function.

     Refer to the screenshots on the next page for proof of functionality.

```
D:\VS\2019 projects\Vanier\Network Systems Design\Lab3\x64\Debug>Lab3-PartA.exe Hello world!
---------------------------------------------------------------------
Custom message entered: Hello world!
---------------------------------------------------------------------
Successfully initialized winsock! Proceeding to next step...
Successfully initialized server info! Proceeding to next step...
---------------------------------------------------------------------
Successfully sent data to server! Proceeding to next step...
Successfully stopped sending to server! Proceeding to the next step...
---------------------------------------------------------------------
Sucessfully closed the socket! Proceeding to the next step...
---------------------------------------------------------------------
All done. Goodbye!
```

Figure 3. Command prompt result output after sending new UDP packet to fictitious server.

```
D:\VS\2019 projects\Vanier\Network Systems Design\Lab3\x64\Debug>Lab3-PartA.exe
---------------------------------------------------------------------
More arguments are needed when executing this program!
Exiting program...
Goodbye!
---------------------------------------------------------------------
```

Figure 4. Command prompt result output shown after attempting to send new UDP packet to fictitious server without passing enough arguments.

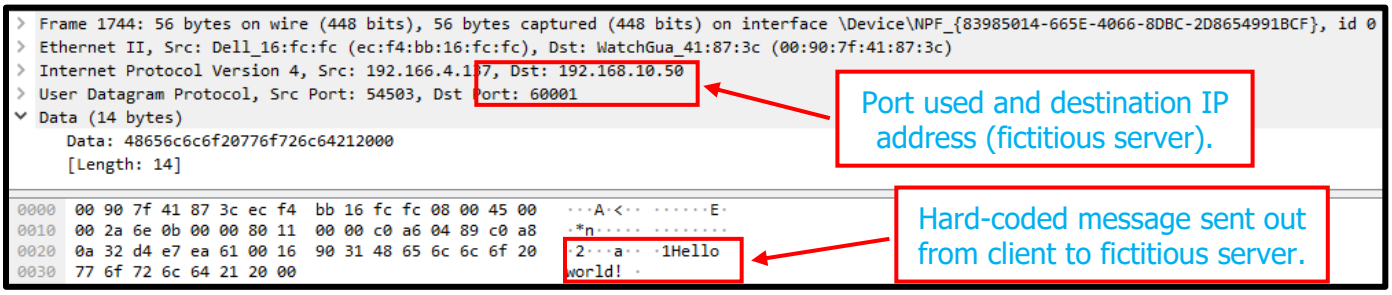


Figure 5. Wireshark packet capture verifying sending new UDP packet to fictitious server.

The screenshot in "Figure 3" shows the result of the newly modified code that accept arguments taken in at the command prompt using argv[] (customized long message), to be sent to the fictitious server. This time, the user not only has to enter the executable name of the client program, but also has to specify what custom message they would like to be sent out to the fictitious server, otherwise they will see the result shown in the screenshot in "Figure 4". If the user does not specify a custom message to be sent out to the fictitious server, the client program will complain about the issue and will close the program. This is done in order to prevent any wasted transmission from happening.

Similar to the previous approach, the resulting new UDP packet that was sent out from the client that contains the customized long message can be found and analyzed using a Wireshark packet capture as shown in "Figure 5" above. From the result shown above, we can see all the various details of the new UDP packet that was sent out from the client. Here, we can see details such as the destination IP address (fictitious server), destination port number (60001) along with the data that was contained in the UDP packet (the customized long message that reads "Hello world!"). We can also see the originating details of this new UDP packet.

5. Illustrate and explain your code and result to your instructor.

Part B: Receive UDP packets (Server side)

6. Create another new project for a console program that will act as *UDP server*.
7. Modify the code to achieve the following. Briefly explain the main code changes done.
    a) Continuously receive UDP packets, until it received an "end" as message from client.
    b) Print the message received on the screen.

➢ The main changes done to the code are as follows:

The following sequential approach was followed in order for receiving a UDP packet (that contains a unique, customized message) from the client:

- Step 1: initialize Winsock (so that unique APIs can be used).
- Step 2: create datagram socket.
- Step 3: setup info of server (IP address, port number, etc…) and bind socket.
- Step 4: **continuously** receive data (UDP packet) from client (using recvfrom() function) until "end" keyword has been received.
- Step 5: when "end" keyword is detected, shut-down datagram socket…
- Step 6: then close datagram socket…
- Step 7: then clean up and quit.

The data that is received by the server is placed in a resulting array (called "RECEIVE") that contains the customized long message that was entered in the command prompt on the client side. The "recvfrom()" function takes the captured UDP packet (data) and places the received customized long message in the "RECEVIE" array. This behaviour lasts forever until the "end" keyword has been detected, which will force the server to terminate. A while loop and an if statement was used in order to accomplish this task.

In order for the server to know when the "end" keyword has been received, the following code had to be implemented (simplified version of what was actually used):

```
...
char* END_RESULT;              //Pointer.
char END_CHECK[] = "end";      //Magic keyword.
...
While(1){
      Receive UDP packet (using "recvfrom()" function)…
      Print received long custom message…

/*Checks to see if "end" keyword has been received.*/
END_RESULT = strstr(_strlwr(RECEIVE), END_CHECK);
...
If (END_RESULT > 0){     //If "end" keyword has been received...
      Shut-down the socket…
      Close the socket…
      Clean up…
      Quit.
}
...
```

Figure 6. Simplified code that incorporates the "end" keyword checking.

The "END_RESULT" line takes the customized long message (stored in "RECEIVE" array) from the received UDP packet (data), converts it all to lowercase characters, then compares if the "end" keyword has appeared anywhere in the customized long message. Converting the customized long message to all lowercase characters makes it easier to check to see if the "end" keyword has been received.

The following screenshot in "Figure 7" shows the client side successfully sending a customized long message to the server side, and the server side successfully keeps receiving customized long messages and prints them until the "end" keyword has been detected.

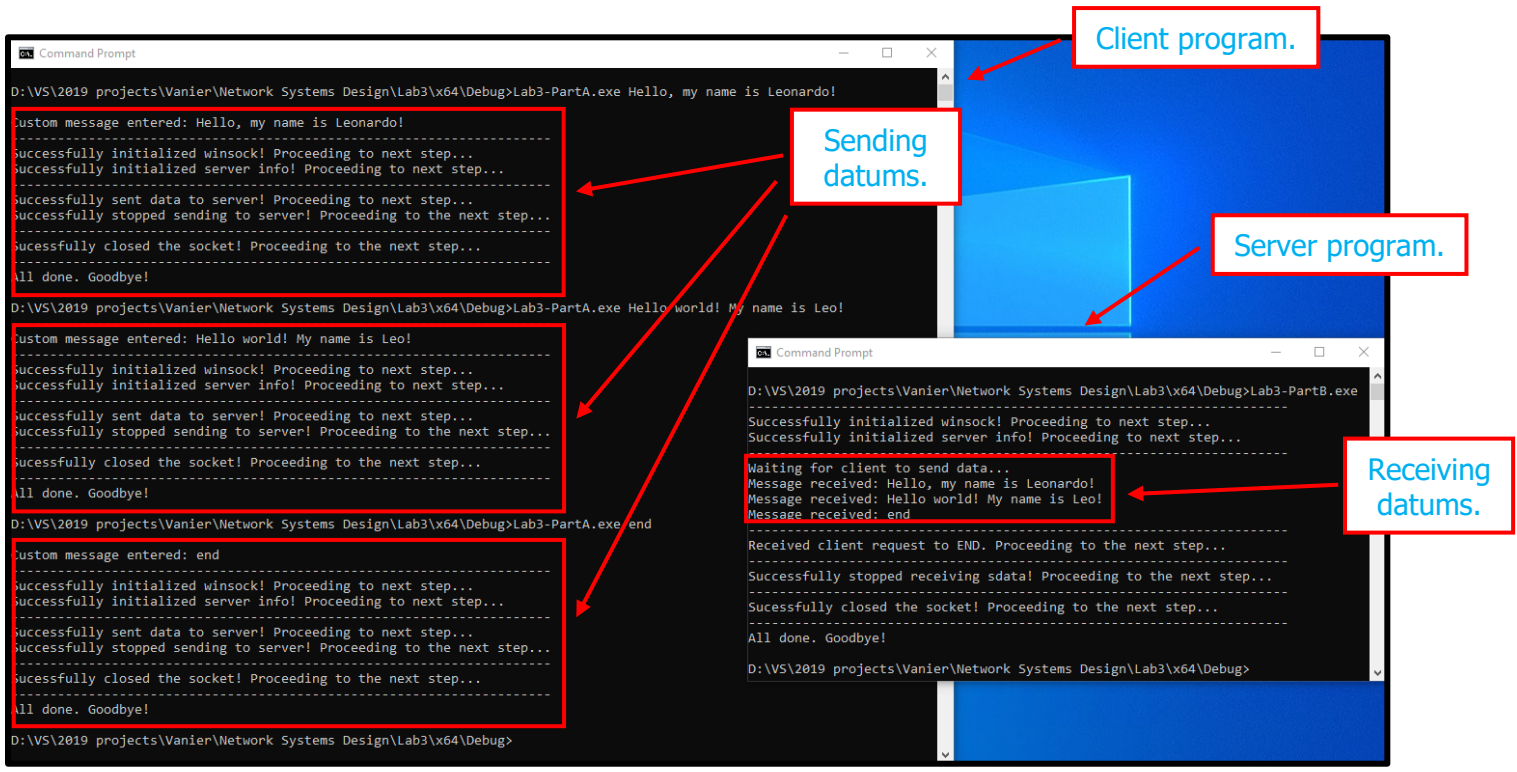


Figure 7. Client and server proof of functionality shown above in command prompt windows. Left window is client and right window is server.

8. Using *loopback IP address*, verify that UDP communication between your client and server. Perform screen shots to show the output of communications between client and server in loopback mode.

   - The previous screenshot in "Figure 7" shows client and server proof of functionality when the two (client and server) are using the loopback IP address for their communication/handshake between the two. In the screenshot in "Figure 8" below, we can verify if the client and server are really communicating using the loopback address by executing the "netstat -an" command. Here, we can see that the server is listening using the localhost (127.0.0.1) on port 60001, but there are no remote machines connected.

     Refer to the screenshot on the next page.

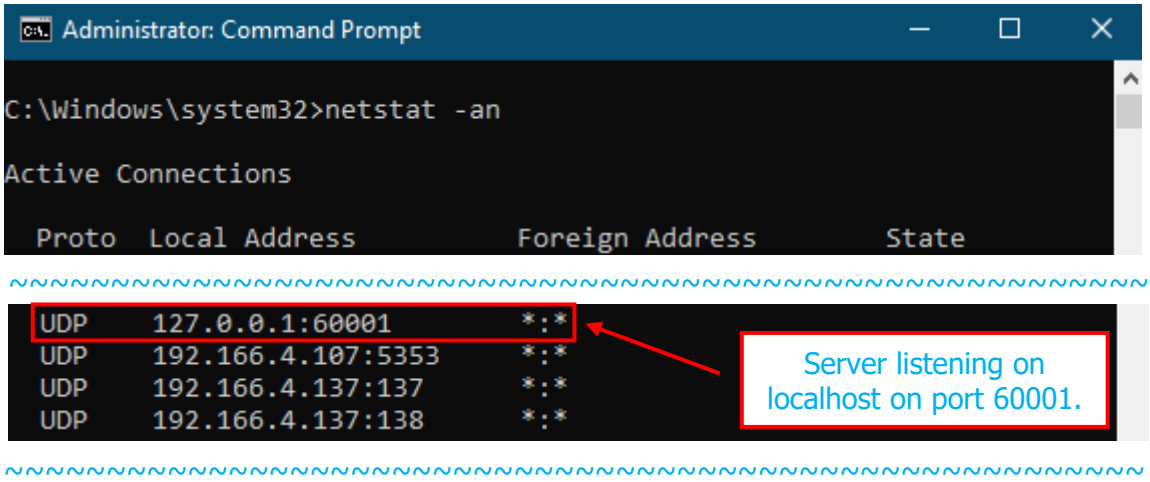


Figure 8. Output from command prompt after using "netstat -an" command (some output omitted for ease-of-viewing).

9. Now, modify your code to turn on a beep if the message received contained the keyword "beep". Briefly explain how this is achieved in your code. *Hints: use "\a" to generate the beep.*

   - The following code had to be implemented in order to perform the requested "beep" functionality. Similar to before (with detecting "end" keyword), the "BEEP_RESULT" line takes the customized long message (stored in "RECEIVE" array) from the received UDP packet (data), converts it all to lowercase characters, then compares if the "beep" keyword has appeared anywhere in the customized long message. Converting the customized long message to all lowercase characters makes it easier to check to see if the "beep" keyword has been received. Refer to the simplified code below (code from before added below):

```
...
char* BEEP_RESULT;                  //Pointer.
char BEEP_CHECK[] = "beep";         //Magic keyword.
char* END_RESULT;                   //Pointer.
char END_CHECK[] = "end";           //Magic keyword.
...
While(1){
      Receive UDP packet (using "recvfrom()" function)…
      Print received custom long message…

/*Checks to see if "end" keyword has been received.*/
END_RESULT = strstr(_strlwr(RECEIVE), END_CHECK);

/*Checks to see if "beep" keyword has been received.*/
BEEP_RESULT = strstr(_strlwr(RECEIVE), BEEP_CHECK);
...
If (BEEP_RESULT > 0){ //If "beep" keyword has been received...
Beep(750, 330);       //...do "beep" sound.
}
...
```

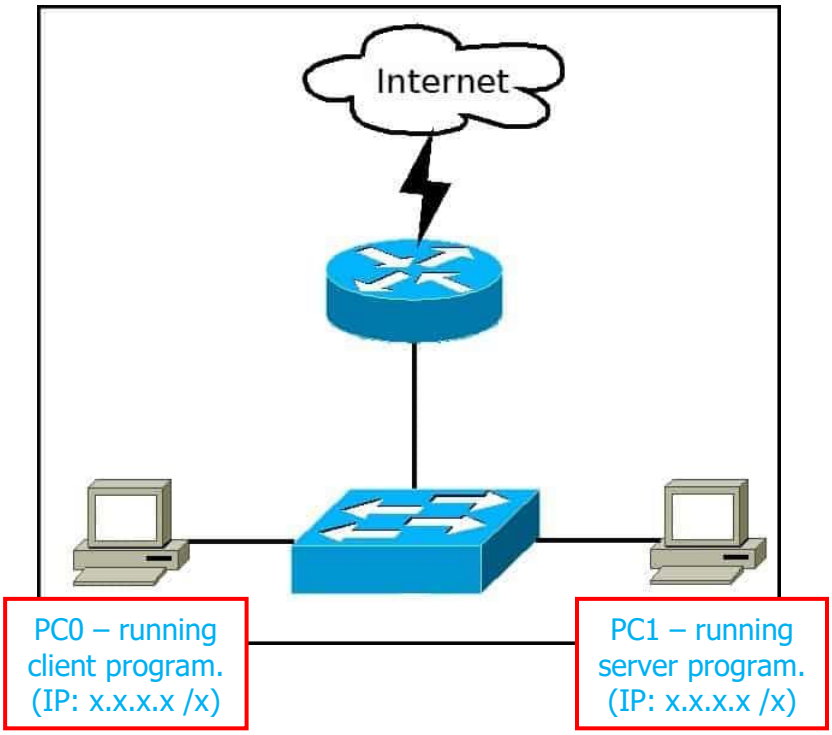
Figure 9. Simplified code that incorporates the "beep" keyword checking.

10. Illustrate and explain your code and result to your instructor.

Part C: Final communication between client and server PC (bonus!)

11. Use 2 PCs, where one of the PC functions as server while the other one a client. Setup the IP addresses in your code according. Illustrate the functionality of your program and demonstrate to your instructor.

- The IP addresses are configured to be the same in both client and server code. The IP address of the computer running the server was used both in the client and server code. The two PCs were connected and configured accordingly:



Code:

- Refer to "Lab3-PartA_LeonardoFusser.txt" for complete Part A code (too long to show here).
- Refer to "Lab3-PartB_LeonardoFusser.txt" for complete Part B code (too long to show here).

Discussion:

- In the very beginning, a very simple socket programming code was implemented. A client, was configured to send out a fixed hard-coded message to a "fake" server. Custom APIs were used in order to accomplish this. Described in this report, a sequential approach was followed in order to even send out the fixed hard-coded message to the "fake" server. The message was stored in an array, and after all the initialization was complete, the client would use the "sendto()" function to send the message to the "fake" client. Once the client was done sending the message, it would follow a terminating sequence (shut-down the socket, close the socket, etc…) before it would close completely. To verify that the message was transmitted, a Wirehshark packet capture was running at the same time as the client was running in command prompt. After some searching, the correct packet was found and some more thorough analysis was done. As shown in the first Wirehark screenshot in this report, the client did indeed send out the message to the "fake" server. The message sent was "THIS IS A TEST!".

  To make things a little more difficult, the code had to be improved so that any length of message could be sent to a "fake" server. Similar to the above approach, the client was configured to accommodate this new feature. A similar sequential approach that was mentioned above was followed, with the exception of taking advantage of argv[] and argc. A simple for loop was used to extract the custom long message (by iterating through value of argc) and copying each argument that was found in argv[] to the same array mentioned above. Once again, after all the initialization was complete, the client would use the "sendto()" function to send the message to the "fake" client. The terminating and closing sequence was the same as the one mentioned above. Once again, a Wirehshark packet capture was running at the same time as the new lient was running in command prompt. The search and analysis procedure is the same as the one mentioned above. As shown in the second Wirehark screenshot in this report, the client did indeed send out the long-customized message to the "fake" server. The message sent was "Hello World!".

  As a finally step, a server was implemented so that it can receive any message that the client was sending out. Additionally, this server had two distinct features: the ability to produce a beep when a "beep" keyword was detected and the ability to detect and close the server program when an "end" keyword was detected. A similar initialization procedure was followed, as explained above, with the exception of the addition of the "bind()" function. Once initialization was complete, the server would wait (in a forever loop) for a message to be received (using "recvfrom()" function). Various mechanisms were put in place to be able to detect if the "beep" or "end" keywords were detected (explained in this report). If the "beep" keyword was detected, the computer would produce a beeping tone or if the "end" keyword was detected, the server program would follow a similar terminating and closing sequence mentioned above before closing the server program entirely. As an additional feature, the server would keep printing what it had received to the output, until the "end" keyword was received.

  The very last step was to observe and test for full functionality. As shown in the last few screenshots in this report, the client and server are able to communicate not only on the same machine, but on two different computers connected to the same LAN. All the keyword detection mechanisms worked as expected, since the server produced a beeping tone or closed the program when either the "beep" or "end" keywords were detected.

Conclusion:

- Successfully implemented basic socket programming using Winsock on Microsoft Windows platform.
- Successfully implemented a basic client and server communication via UDP between two computers connected to a common LAN.
- Successfully used MS Visual Studio debug tools to troubleshoot communication issues.
- Successfully used Wireshark to verify communication functionality.
- Successfully used netstat command to also verify communication functionality.