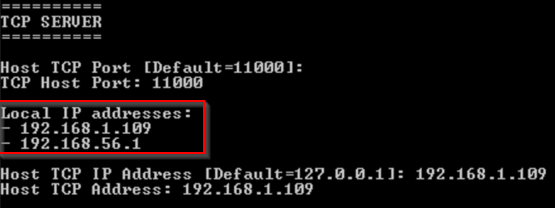
# Written Report

As demonstrated in the lecture with Python code, creating a client/server pair that echoes messages in uppercase using TCP and UDP sockets is not very challenging when using a high level language. What is a bit more challenging is writing reusable code that is easily implemented in a client or server application. Therefore, my solution includes a common library that exposes transport, client, and server functionality to applications that need to transmit or receive UDP or TCP traffic. There are more details on this approach along with UML diagrams in the Source Code section if you are interested in the software design of my solution.

The most challenging aspect of this project was finding good resources for using the lower level Windows Sockets library rather than the higher level and more commonly used abstractions like TcpClient. Furthermore, socket examples for TCP transport were more readily available than UDP. From this, I concluded that TCP is a very common, general purpose transport while UDP is a bit more specialized.

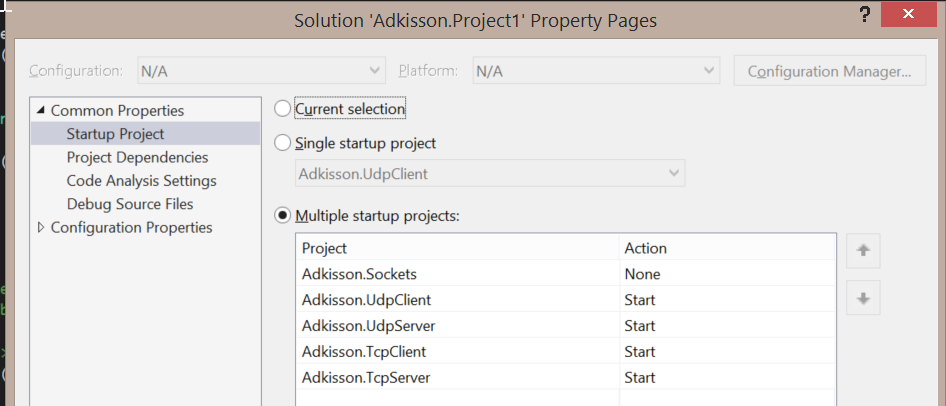
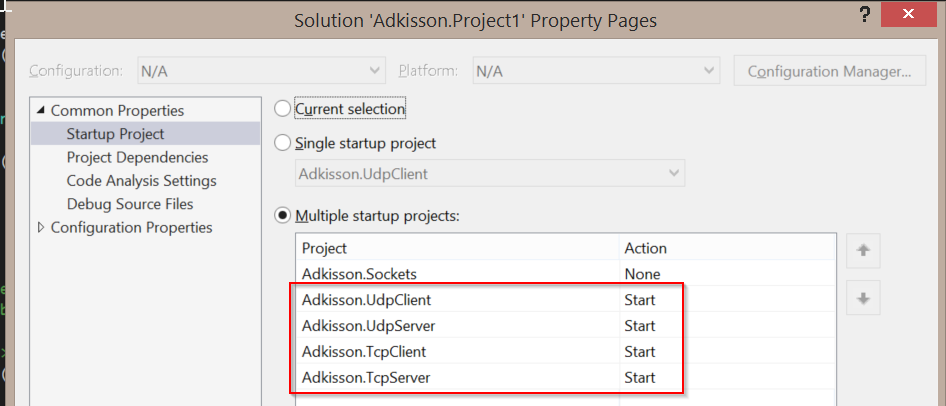
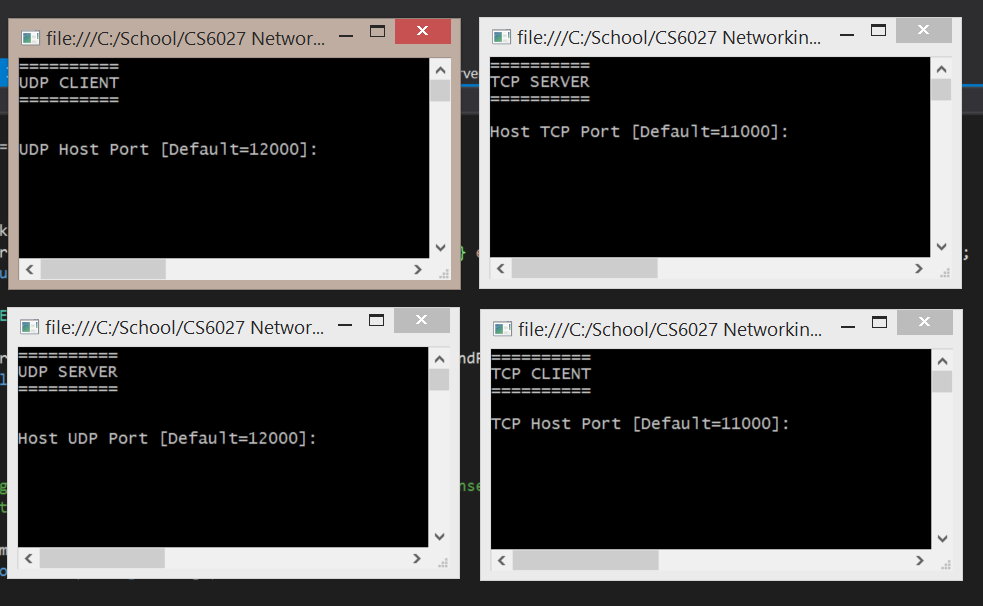
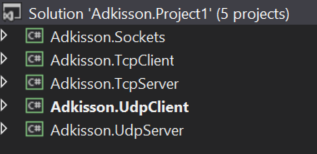
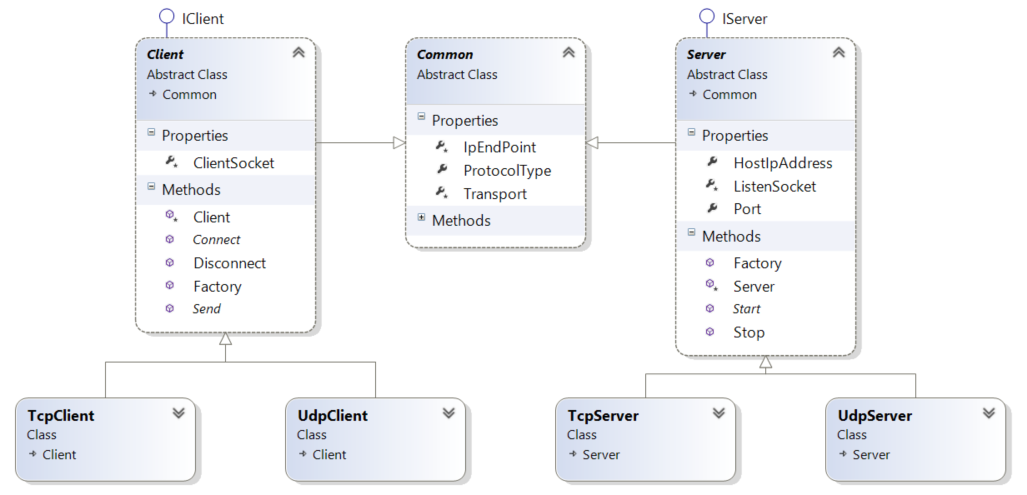
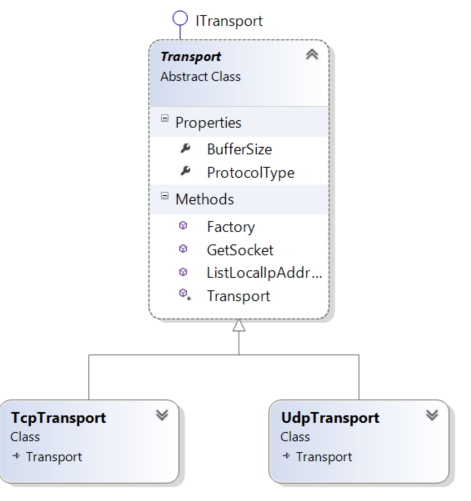
Another challenge was the Windows Socket API has a lot of methods that do nearly identical things with slightly different method signatures. For example, Send versus SendTo both transmit data to a client. While MSDN and StackOverflow were helpful, I ultimately purchased an out of print book, *TCP/IP Sockets in C#* by Makofske, Donahoo, and Calvert (2004, Morgan Kaufmann Publishers), to better understand the available API methods and choose what was most appropriate to the project.

The part that took me longest to complete was not related to the actual socket programming. It was trying to find a way to automatically bind the server to the right IP address. Binding to the loopback (127.0.0.1) is easy, but I needed the machine’s network IP to enable remote communication. I found the methods necessary to retrieve the interfaces and addresses on the machine, but there are multiple IPs on my machines (Ethernet, Wifi, and virtual IPs for VMs). I finally decided rather than have the code pick the wrong one, I would list the IPs on the machine and let the user decide when starting the server. In a more sophisticated solution, I would have made the server multithreaded and bound each thread to an IP address.   
  


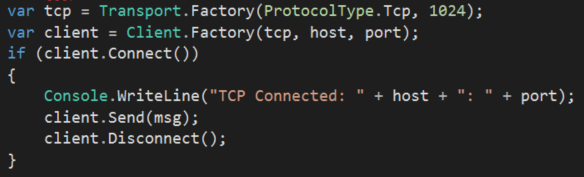
Establishing communication between the client and server over my home network only involved a minor local firewall change on the server machine to allow TCP and UDP traffic on the ports the user chooses when starting the consoles.

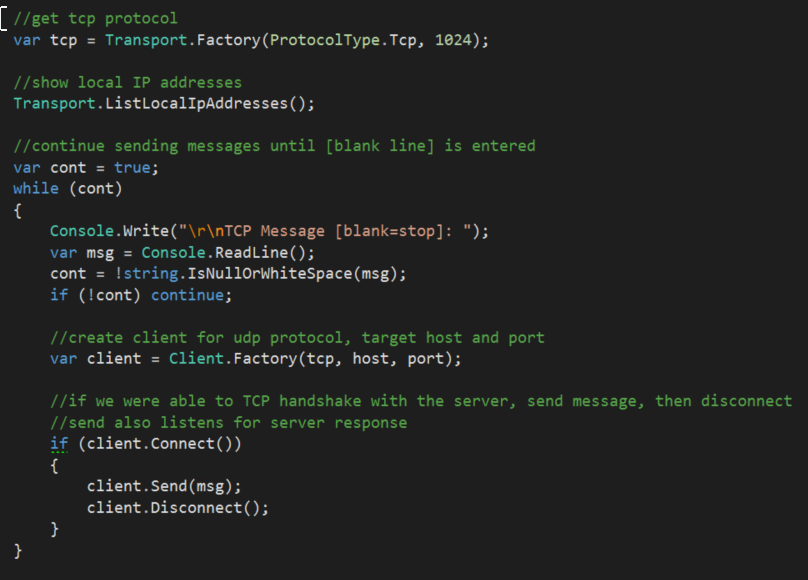
I enjoyed this project. Network programming feels a bit like magic. My prior experience has been mostly confined to HTTP calls to REST APIs on the web using higher level libraries. It was a pleasure to better understand how the mechanics of these connections work, particularly TCP.

# Source Code

1. All code was written by me, Jeff Adkisson, with no partner. My reference material included MSDN, StackOverflow, and a book on socket programming in C#: *TCP/IP Sockets in C#* by Makofske, Donahoo, and Calvert (2004, Morgan Kaufmann Publishers). I did not copy/paste any blocks of code.
2. For your convenience, binary executables are available in the zip file’s Binaries folder.
3. Test runs were performed on Windows 8 and Windows 7.
4. The source code was written in C# for the .NET 4.5 framework and is included in a Visual Studio 2013 project.
5. To compile and start all four consoles from Visual Studio:  
   1. Locate **Adkisson.Project1.sln** in the include source code zip file to open project in Visual Studio.
   2. Click on **Adkisson.Project1** in the Solution Explorer window:  
      
   3. Press Alt-Enter to open the solution properties. Select Startup Project:  
      
   4. Set the UdpClient, UdpServer, TcpClient, and TcpServer projects to Start. The Sockets project Action should be None.  
      
   5. Click **OK** to close the solution properties.
   6. Press F5 to compile the solution and start all four consoles:  
      
6. The source is split into 5 projects:  
     
     
   1. The Sockets project is a class library referenced by the other projects to provide TCP and UDP client/server functionality.
   2. The TcpClient, TcpServer, UdpClient, and UdpServer are Windows console applications that reference the Sockets project to enable TCP and UDP socket connections between systems.
7. The Sockets project provides common functionality to all four consoles to make creating clients and servers simple:  
     
     
   To create a new client or server, you inject a Transport protocol into the Factory method for the Client or Server class:  
   

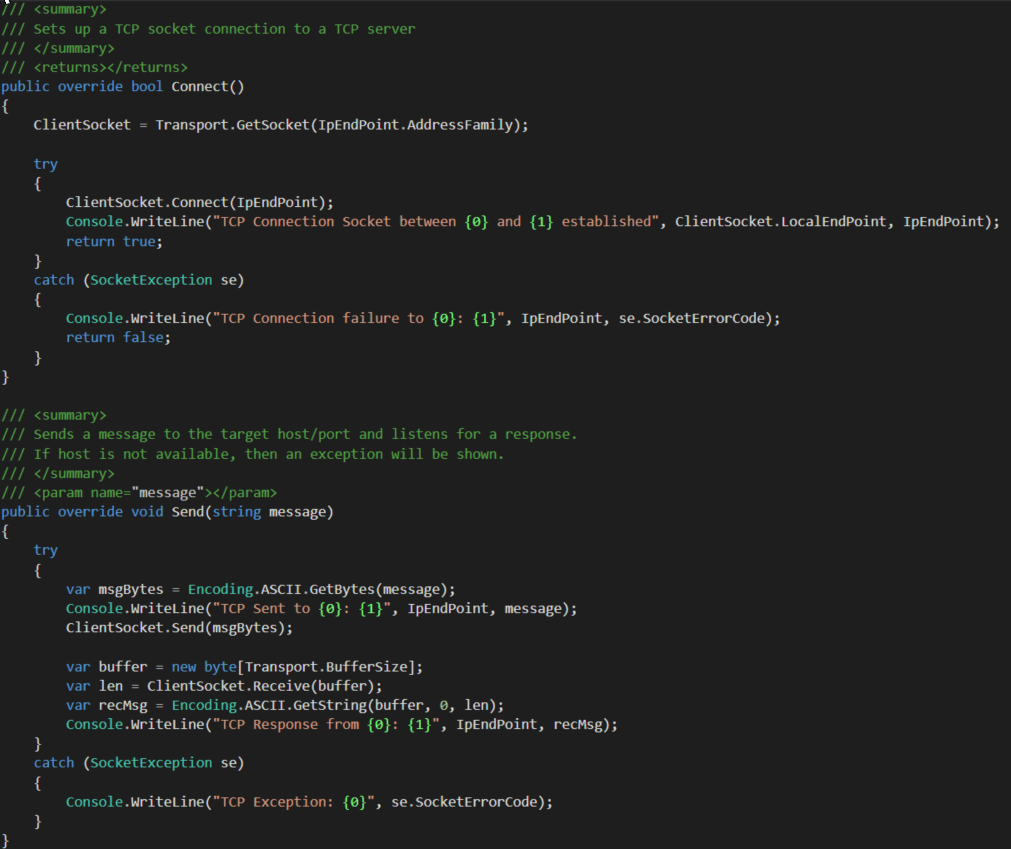
This makes client and server code within the console projects very simple:

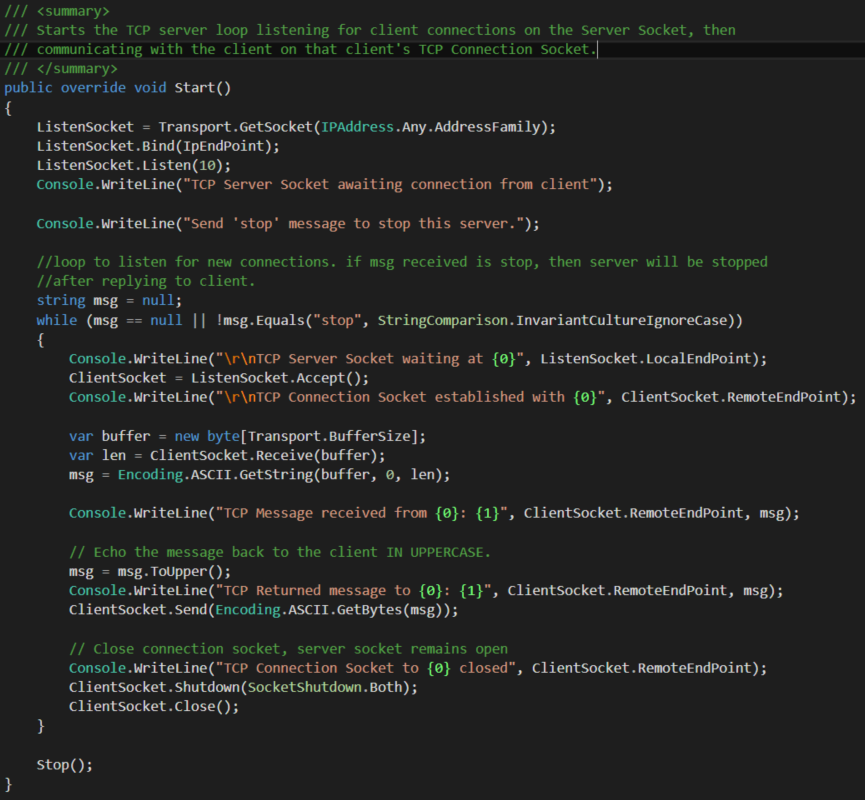


1. All code is commented and clearly indented:  
     
   

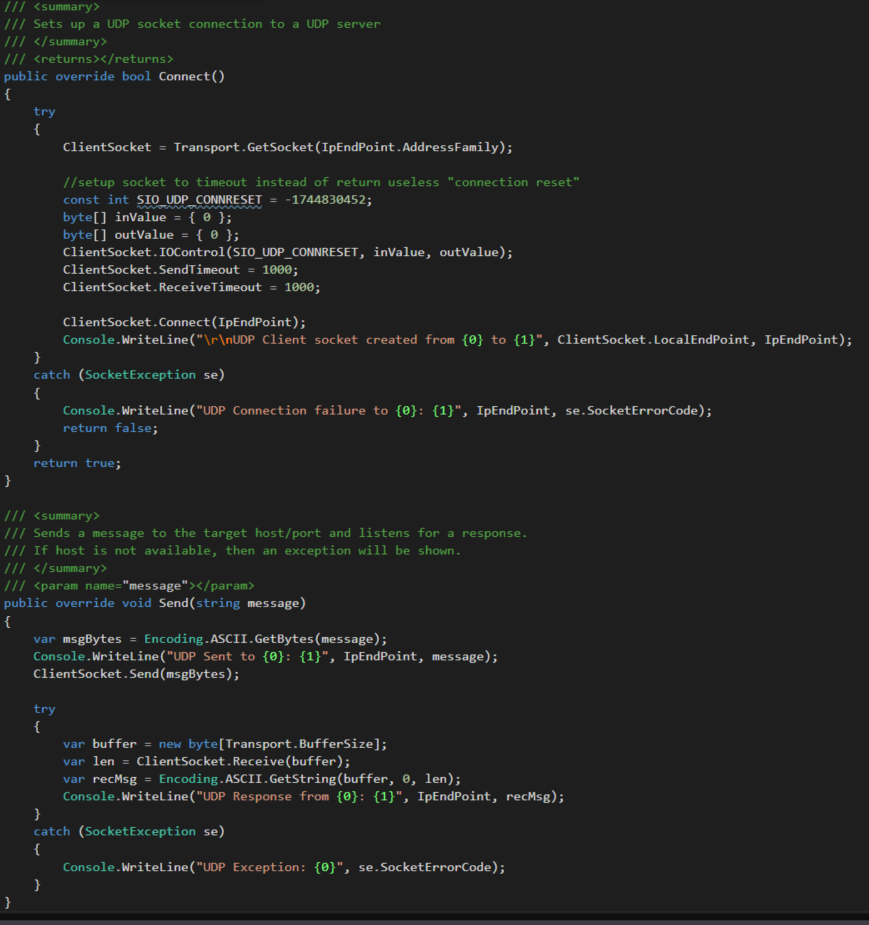
# Source Code of Key Socket Mechanisms

TCP Client – establish connection and send/receive a message:

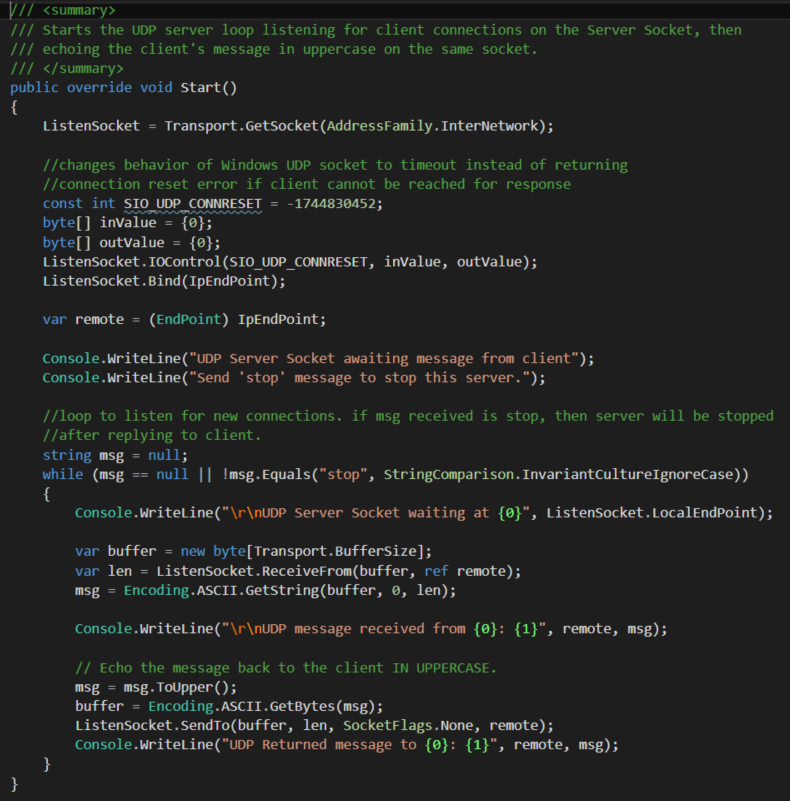


TCP Server – listen for clients and echo inbound messages in uppercase:  
  


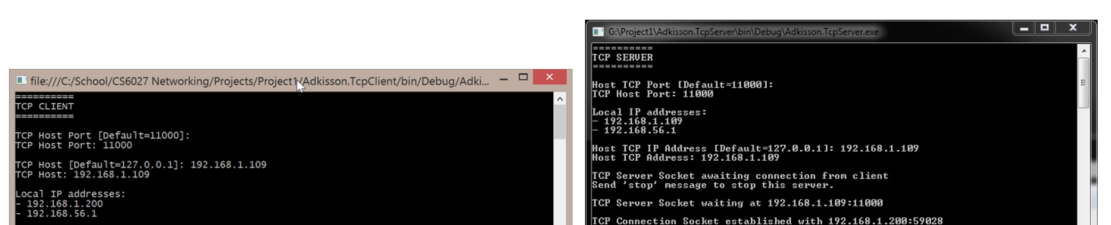
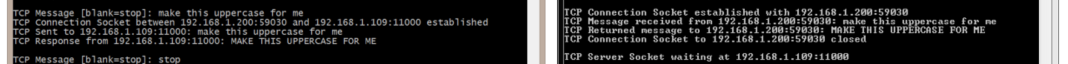
UDP Client – establish a connection to a UDP server, send messages, and write server reply:



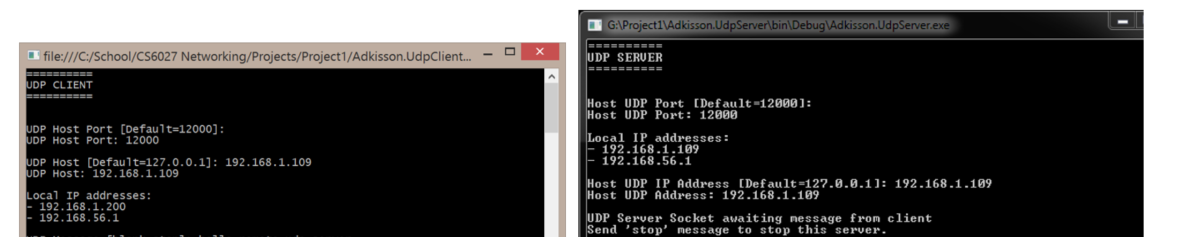
UDP Server – listen for clients and echo inbound messages in uppercase:



# TCP Demonstration

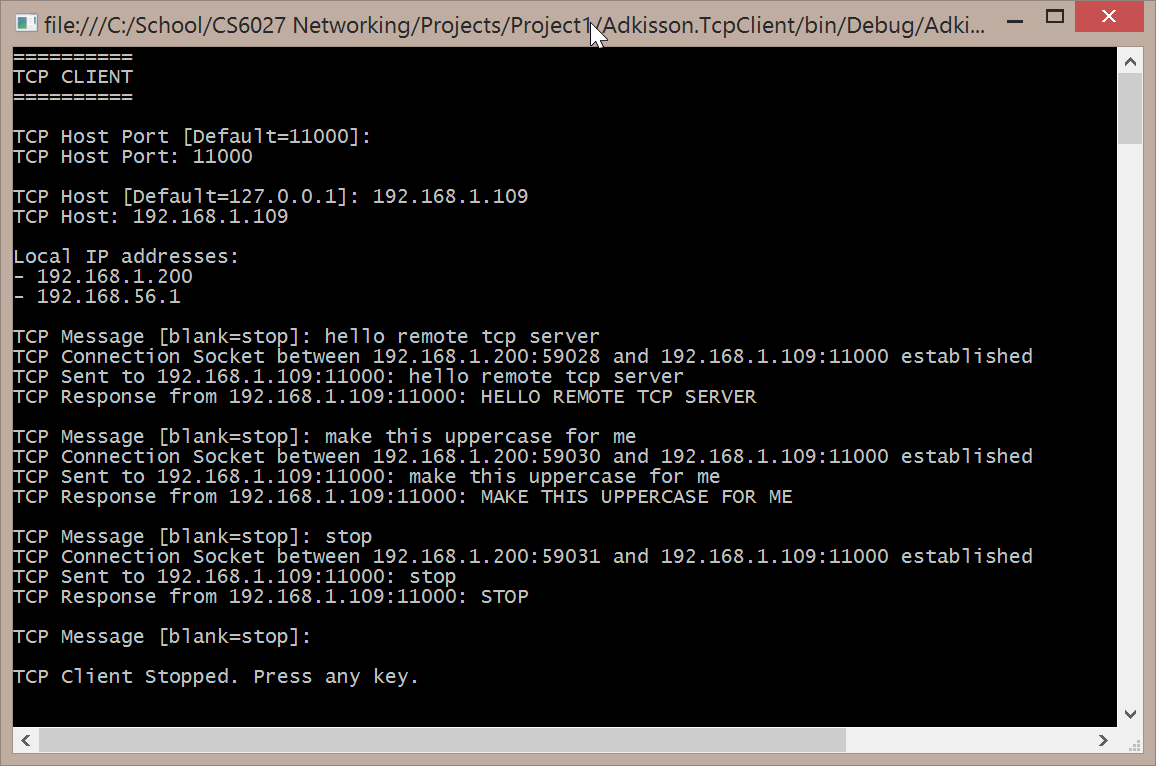
1. I setup the TCP client on 192.168.1.200 and the TCP server on a separate machine on 192.168.1.19. I set the port on both machines to 11000. Note that on the server, I had to open the firewall port to allow TCP traffic to port 11000.  
   
2. I sent the message ‘hello remote tcp server’ from the TCP client to the TCP server. The client opened a TCP socket to the server. The server’s Server Socket on port 11000 accepted the connection, then established a Connection Socket for the client’s use. The server replied ‘HELLO REMOTE TCP SERVER’, then closed the connection socket. The Server Socket remained open.  
     
   You will note that the client’s TCP socket port (OS assigned) is 59028. This is reflected on both systems.
3. The client sent ‘make this uppercase for me’ to the server. The server replied ‘MAKE THIS UPPERCASE FOR ME’:  
   
4. The client sent ‘stop’ to the server. The server replied ‘STOP’ and also closed the Server Socket and terminated its listening loop:  
   
5. I entered a blank line into the client to terminate the client send loop:  
   

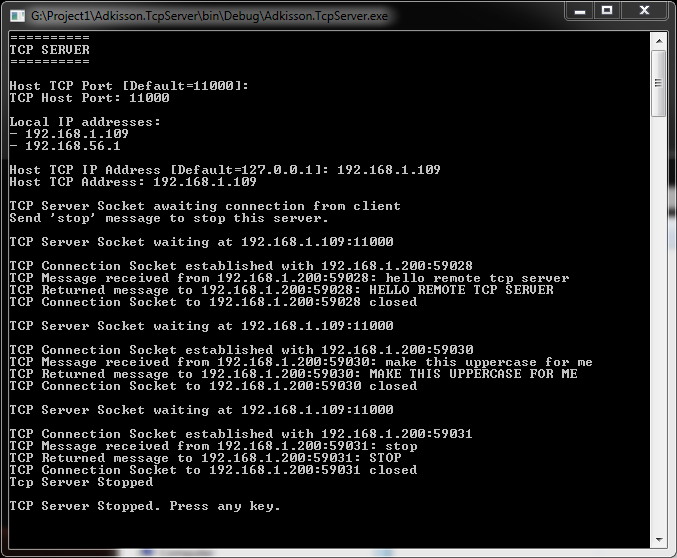
# UDP Demonstration

1. I setup the UDP client on 192.168.1.200 and the UDP server on a separate machine on 192.168.1.19. I set the port on both machines to 12000. Note that on the server, I had to open the firewall port to allow UDP traffic to port 12000.
2. I sent the message ‘hello remote udp server’ from the UDP client to the UDP server. The client opened a UDP socket to the server. The server’s Server Socket on port 12000 accepted the message and replied ‘HELLO REMOTE UDP SERVER’, then continued listening on port 12000.  
   You will note that the client’s UDP socket port (OS assigned) is 59470. This is reflected on both systems.
3. I sent the message ‘I love you’ from the UDP client to the UDP server. The server replied ‘I LOVE YOU’ and continued waiting for further messages.  
   
4. I sent the message ‘stop’ from the UDP client to the UDP server. The server replied ‘STOP’, then closed the Server Socket and stopped the listening loop:  
   
5. I entered a blank message into the UDP client to stop its sender loop:  
   

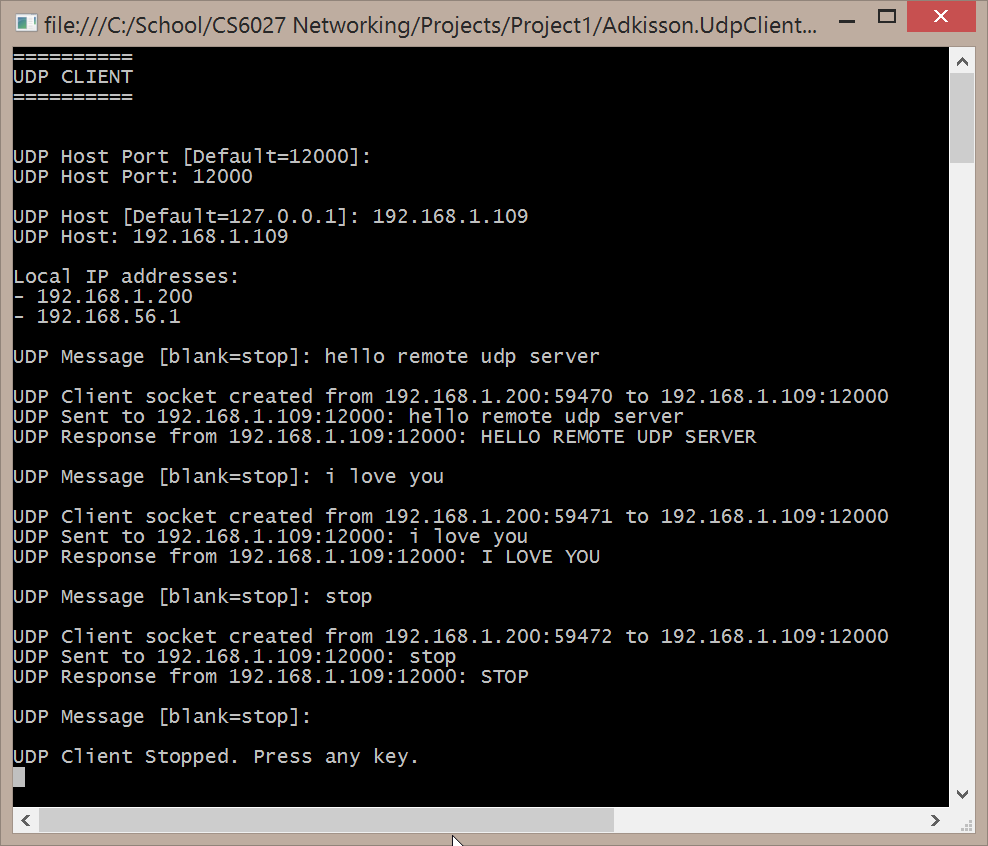
# Screenshots

TCP Client on 192.168.1.200

  
  
TCP Server on 192.168.1.109



UDP Client on 192.168.1.200



UDP Server on 192.168.1.109

