1. **Introduction:**

This report will present our networking project. The idea behind it was to make an application that would use the socket API to connect two laptops via LAN and let them communicate through this application.

The socket API is used on the transport layer of the TCP/IP model. It allows a server to bind to a port, listen to connections, accept them, and finally communicate with any client connected to it. This API is pretty straightforward in its use, but does require some thread handling to maintain responsiveness for both server and client.

This report will describe two experiments:

* + A basic chat application that allows many clients to connect to a server and send messages through it.
  + A game of Pong that allows a host and a client to compete over the network.

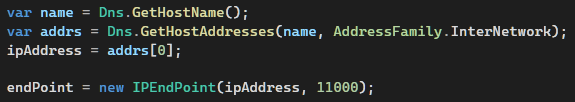
First, we will start by laying some foundation and basics used in both experiments, then we will dive deeper into the intricacies of each.

**The Socket API:**

As stated above, the socket API permits the communication between a server and many clients over the same LAN. Our applications will both be using the C# language to achieve the intended result.

Let us take a look at how a server can initiate its listening to allow new connections.

First, we should create a new socket. This is done in C# lie this:



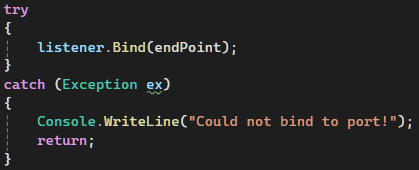


We start by trying to get the IP address we wish the connection to be done on. To do this, we get the name of the host (the machine the server will run on), then, from the name we get all the addresses, here, we specifically choose to only get IPv4 addresses by adding “AddressFamily.InterNetwork” as an argument. The result of “Dns.GetHostAddresses” is an array of IPAddress objects, so we set the IP address to the first element in the array.

The next step is to declare an EndPoint object using both the IP address chosen and an arbitrary port. Please note that the port can be chosen at will as long as it is not used by any other application. Also, we could use the IP address “0.0.0.0/0” as an endpoint address to allow connection from any network card or interface connected (ex. Use either WIFI address or the Ethernet address without the need to choose one in particular from the array of addresses.)

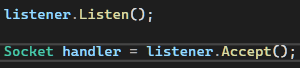
Finally, we create the socket using its class constructor where we can specify the address type used, the type of the socket (here we choose stream to allow data exchange) and lastly the protocol use, here, the TCP protocol.

Now that we have a socket ready to use, we need to bind it to the endpoint, this is done like this:



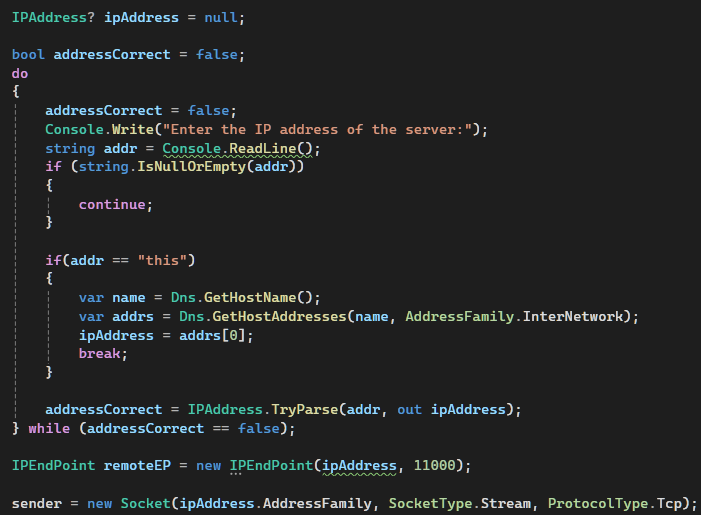
If for some reason, we could not bind, we can catch an exception and display an error.

Our server is now ready to listen to new connections, and accept them:



The socket listens to any incoming connection, and upon receiving one, we get the new socket that wishes to connect, we just have to accept the connection and store the new socket for later. It will allow us to communicate with the connected client.

This is all it takes for our server to accept a connection from a client. This process is somewhat similar for a client:



The client must connect to the server, so it has to be aware of its IP address to initiate a connection. We can get the IP address from a simple input from the user, if “this” was the input, we assume the client is on the same machine as the server, so we use similar steps to get the address. In the case the client is on another machine connected to LAN to the server, we try to convert the string input to a usable IP address using “IPAddress.TryParse” that takes the string and an empty IPAddress object and try to do the conversion, if it is successful, it returns true and fill the object with the parsed address, else it returns false, forcing the user to re-input a correct address.

Then, we create an endpoint with the same port. This is important because the application are communication through this same port.

The next step is to create our socket using the same method as the server.

Finally, we try to connect to the server using our remote endpoint:



The client can now connect to a server, and the server can accept the connection. But both don’t communicate yet. But fortunately, the data exchange is done using two simple methods and some buffers.









The Send method allows to send an array of bytes while the Receive method receives the bytes and fills the buffer with these bytes.

In brief, one of the two entities sends the data, and the other receives it. It is crucial here to make sure that not both try to receive data or send it at the same time to avoid errors.

And that is how the Socket API works. Understanding the basics will greatly help further down the line for both the test chat application, and the pong game that will shake things a bit as it uses its own implementation of the transport protocol

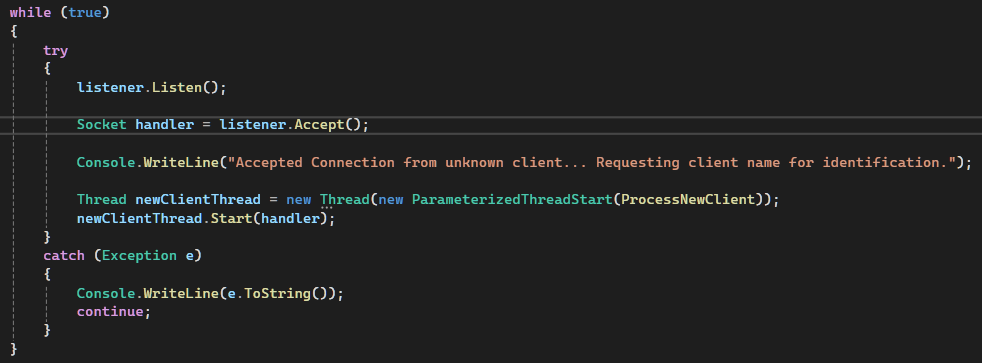
1. **The Socket API Test:**

Now that we understand the socket API, let’s see how we used it to make a simple chat application that allows multiple clients to connect to a server to send messages back and forth.

Before the communication starts, the server should listen to multiple connections, that’s why the listening is done inside a while loop that continuously runs until the server closes. Each new socket will participate in a two way communication between the server and the client

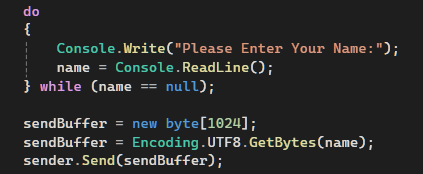
We will view this communication as it happens starting from the client’s connection getting accepted.

1. The server first gets the new accepted socket.



The server starts a thread for the new client to process the new client and return as fast as possible to listening for new connections.

1. At the same time, the client is prompted to enter a name for identification:



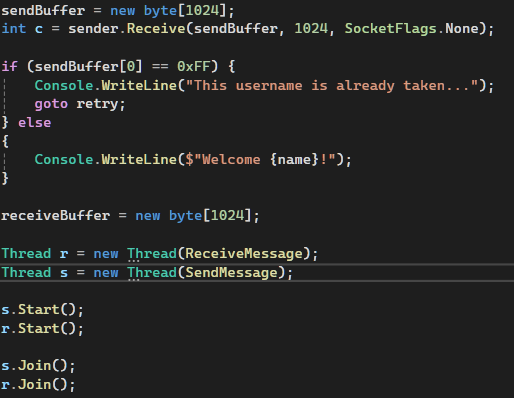
After a name is entered, it is sent to the server as bytes.

1. The server then receives this name:



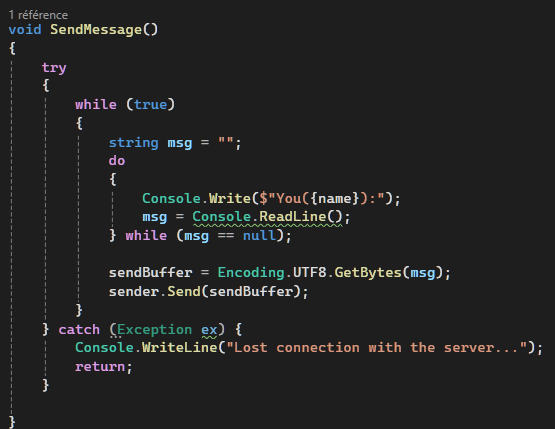
The name is converted back to a string. Then, the server checks within a dictionary of <string, Socket> objects (each string (key) refers to a Socket (value)), if the name is already inside the dictionary, it means that a client already holds this name, the server sends then a buffer full of 0xFF to signal the client that it has to re-enter a name. In the other case, a buffer full of zeroes is sent and the name and its corresponding socket is added to the dictionary. This check is used to ensure no two clients share the same name for identification purposes. Also notice the use of the block “lock” and within its arguments the variable “lockThread”. This is used to lock the thread when handling the dictionary to avoid any race condition that would result in a simultaneous modification of the dictionary which would lead to errors.

1. Upon receiving the result for the name picking, the client either asks for a new name, or finally starts the conversation:

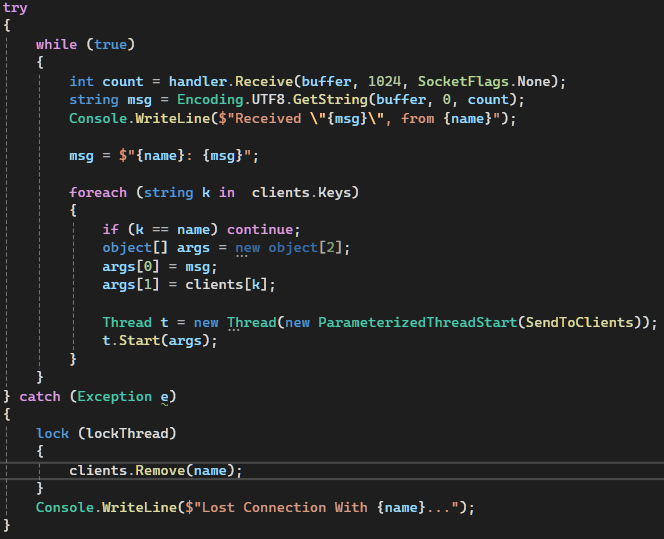


The conversation is started within two threads, one for sending messages, and one for receiving messages.

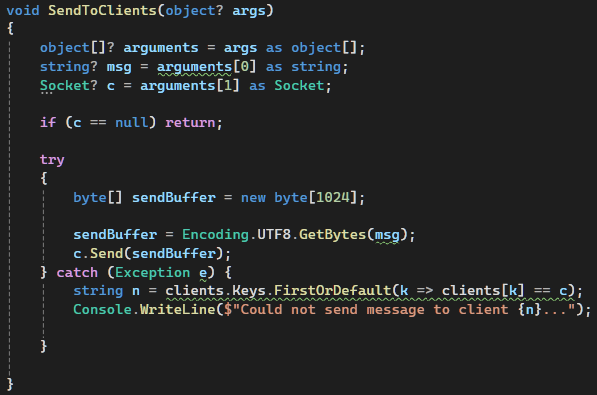
1. The client will send messages in the same manner as it sent its name to the server:



1. For each client, the server tries to receive new messages:



Upon doing so, it sends this message to all clients besides the sender. This is also done in another thread to not hang the receive end. Both the target socket and the message are entered as arguments for the thread inside an array (a thread can only take one argument, hence the use of an object array, this is because all classes inherit internally from this object class, and then can be added to this array then sent to the thread).



If an error occurred or the server lost connection to the client, it makes sure to remove it from the dictionary and closes the thread. The lock is also used here, while modifying the dictionary.

1. The message sent from the server is received on its dedicated thread on the client:



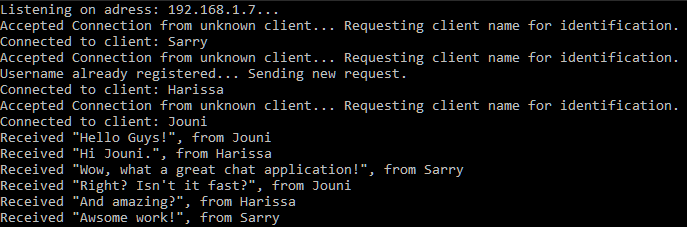
You may notice some weird code after receiving the message, this is used to print the message received above the input line of the user for better presentation.

Also, as seen, the client doesn’t need any thread lock. In fact, the clients’ thread will never try to modify any global variables, so locks would be useless.

And, that’s all there is to this chat application! It served as really good opening act and introduction to the socket API and the thread handling. It made use of all the basics introduced earlier while adding additional challenges ranging from printing on different areas of the console to handling new clients and providing a responsive communication mean.

Here are some outputs:

Server Output:



Clients Output:

