Important note

This module will use .NET 8.0 Long Term Support as .NET 9.0 Short Term Support was only released in Nov 24 and therefore the lab machines cannot be guaranteed to work correctly with this version.

SETTING UP YOUR COMPUTER

If you are working on this module at home, you will need to set up your machine correctly to ensure you can successfully complete the labs and coursework. The best thing to do would be to match the Lab PC setup as closely as possible.

Visit https://azureforeducation.microsoft.com/devtools and download Visual Studio 2022 Enterprise Edition. Ensure it is fully updated to the version in the labs.

Once installed you should choose these workloads (Tools -> Get Tools and Features):

- ASP.NET and web development
- .NET desktop development
- Data storage and processing

Under 'Individual Components' ensure that these components are also checked:

- **.NET 8.0 Runtime (Long Term Support)**
- .NET Core SDK
- Advanced ASP.NET features
- CLR data types for SQL Server
- Connectivity and publishing tools
- Data sources for SQL Server support
- IIS Express
- SQL Server Command Line Utilities
- SQL Server Data Tools
- SQL Server Express 2016 LocalDB
- SQL Server ODBC Driver
- NuGet package manager
- ASP.NET and web development prerequisites
- Entity Framework 6 tools

Click Modify if you have added any of these options.

A local firewall may prevent your application(s) from gaining network access. Usually this will not be a problem when you are using the localhost (loopback adapter: 127.0.0.1) address but if you want to set up communication between multiple LAN devices or connect over the Internet you may need to allow access in software firewalls and/or in your router settings. Typically the labs use ports 5000-5002. This is usually an unnecessary step so don't worry unless you have a problem.

Often, Windows Defender will ask you to allow network activity when starting a server so you can choose to click allow if it is your code running.

Dist. Sys. Client / Server Recap

Objective

The aim of this tutorial is to provide a brief recap to client server communication using the socket classes available in .NET. This should recap what you may have done in previous modules relating to networking or introduce you to some low-level networking. The workshop consists of two exercises:

1. Creating a simple two-project solution with a client and a server application. A sample code for the client and the server will be given. Your main task will be to write a simple Pig Latin translator. The client will send a plain English message to the server, the server will translate it to Pig Latin and send the result back to the client to display to the user.

2. Appendix A gives guidelines on how to set up multiple startup projects in Visual Studio (so that both your client and server start when running your solution from Visual Studio).

Note: If there are any questions or something is unclear in this tutorial, please do ask a member of module staff or a demonstrator.

Introduction

Network sockets are one of the oldest and most pure mechanisms for programmatically establishing connections across the network and exchanging data between computers. Sockets are still widely used because of their simplicity. The internet is built on socket connections.

In this practical we will revisit how to build simple network applications in C# using the socket classes provided by .NET. Most languages and environments provide network socket capabilities and you could easily implement the examples shown here using Java, C/C++, or just about any other language of your choosing on almost any operating system.

It is important to understand this concept because most frameworks now hide the actual connection from the developer and yet understanding the principles of what is going on under the hood is often invaluable to solving errors and ensuring your code is efficient.

Sockets and Ports

A socket is simply the logical endpoint of a communications channel and provides a mechanism for creating a virtual connection between processes on the same or different computers. Every socket has a unique socket address which consists of a combination of the local computer's network address and a port number. A port is a logical communication channel with a unique port number; port numbers are used to differentiate between logical channels on the same computer. A single computer can have up to 65535 ports (i.e. up to 65535 logical communication channels).

Each network-based application program or protocol has a unique port number associated with it which is used in all communications. For example:

- Web servers using the HTTP protocol use port 80 as the default communication port.
- Mail servers using POP protocol use port 110 as the default communication port.

When developing a network application it is up to the developer to choose the port which will be used for communications. Any value in the range 0 to 65535 is valid, but ports 0 through to 1024 are normally reserved by the operating system for specific applications and it is normal practice to choose a port number in the range 1025 – 65535 for non-standard applications.

A network connection is established by creating a socket on the client computer which is logically connected to a specific port on a server computer located elsewhere in the network. The socket created at the client does not specify which port number should be used; instead the operating system arbitrarily chooses an available port number. However we must always specify the port number on the server to which the client should be connected. For example, if we were implementing a Web browser we would create a socket that connects to port 80 on a Web server, as port 80 is specified in the HTTP protocol as the default port for communication with Web servers using HTTP. We do not know, or care, which port is associated with the client socket created by the Web browser for this connection; it is sufficient to know that the client socket has been created and the network connections established. The process of establishing a network connection is illustrated in Figure 1.

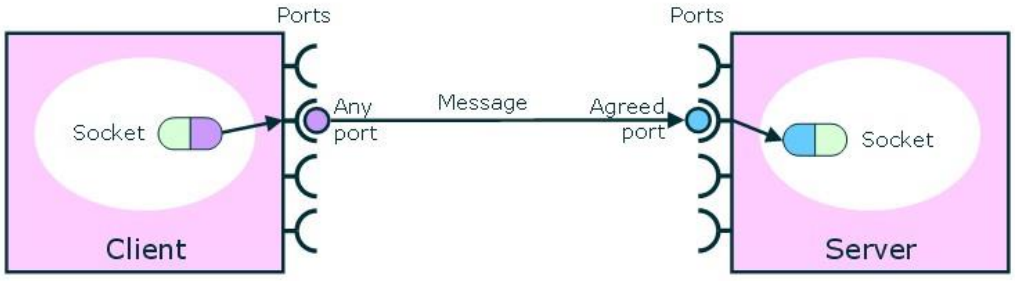


Figure 1. Illustration of client and server sockets.

Creating Client Sockets in .NET

.NET has an extensive networking library and provides several different ways of creating and interacting with sockets. The simplest to use are the **`TcpClient`** and **`TcpListener`** classes respectively. These are found in the **`System.Net.Sockets`** namespace.

The **`TcpClient`** class implements the socket used on a client to connect to a known port on a server elsewhere in the network. When a **`TcpClient`** is created we must supply the network address and port number of the server to which a connection is to be established.

The **`TcpListener`** class implements a server socket which simply listens for incoming connections. When we create a **`TcpListener`** we only need to specify the port number on the local machine with which the **`TcpListener`** should be associated.

Whenever the **`TcpListener`** accepts an incoming connection it creates a new **`TcpClient`** instance which acts as the endpoint of the communication on the server machine. Consequently, after a connection has been accepted by a **`TcpListener`**, there is a **`TcpClient`** instance at either end of the connection, one on the client end and one on the server end.

Every **`TcpClient`** instance has an I/O stream associated with it which can be retrieved by calling the **`TcpClient.GetStream()`** method. This returns an instance of the **`NetworkStream`** type; this is an I/O stream which behaves in an identical fashion to any other I/O stream.

Various input (reader) and/or output (writer) classes can be constructed on top of the **`NetworkStream`** to allow data to be written to or read from the **`NetworkStream`**. Data written to a **`NetworkStream`** is transmitted over the network connection and can be read from the stream by the process at the far end of the network connection.

The processes involved in building a network client application using .NET are straightforward:

1. Create a **`TcpClient`** instance, passing the name or address of the server and the port number on the server as constructor arguments
2. Retrieve the I/O stream associated with the **`TcpClient`** socket by calling the **`GetStream()`** method
3. If required, construct appropriate readers and writers on top of the network I/O stream
4. Using the readers/writers or the raw network stream, write data to and read data from the network connection and hence to/from the server.

Exercise 1: Revisiting Server Sockets in .NET

The **`TcpListener`**[1] class implements the server socket used by a server to wait for incoming network connections. When a **`TcpListener`** is created we must supply the port number to which it should be bound; the **`TcpListener`** will then wait for connections arriving at that port.

Once the **`TcpListener`** has been created its **`Start()`** method must be called to start it listening, then its **`AcceptTcpClient()`** method must be called to make it wait for an incoming connection. The **`AcceptTcpClient()`** only returns when a new connection is established and it returns a **`TcpClient`** instance which is the endpoint of the connection at the server and must be used for communicating with the client.

The **`TcpListener`** does not provide any means of communicating with the client, all it does is wait for incoming connections and return **`TcpClient`** instances for those connections. We must use these **`TcpClient`** instances for all communication to/from the client.

Creating a Simple Translator

In this exercise we will develop a rudimentary client/server application which will translate text to Pig Latin. We will implement this using ASCII encoding.

To make the 'Pig Latin' form of an English word for our translator:

- If the word starts with a consonant, this consonant is transposed to the end of the word and the letters '*ay*' are appended. So "pig" becomes "igpay", while "latin" becomes "atinlay".
- If a word starts with two or more consonants move all of the consonants to the end of the word and add "ay." So "plate" becomes "ateplay" and "llama" becomes "amallay".
- If a word starts with a vowel add the word "way" at the end of the word. So "animal" becomes "animalway" and "empower" becomes "empowerway".

Your server will listen for incoming connections on a specific port; when a connection is made it will use the associated **`TcpClient`** instance to read the request message from the client, translate the message in the request and write it back out to the client using the **`NetworkStream`** associated with the **`TcpClient`**.

Note that due to restrictions on the use of port numbers below 1024 and security policies in place in the computing laboratories, it will not be possible for us choose just any port. If you are doing this work in the Fenner lab or using a remote connection to the Fenner lab, ports **5000-5002** should be open for your use in all the lab tutorials, so we will be using port 5000 in this exercise.

---

[1] In other environments, e.g. Java and Unix, a **`ServerSocket`** is the equivalent of a **`TcpListener`**.

Steps to make this solution (some already completed in the sample code):

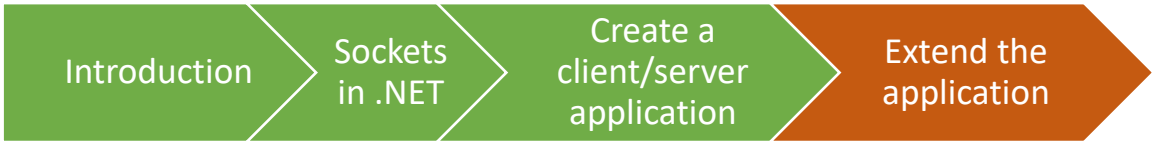
- Create a new C# console application in Visual Studio

First we will create the server.

- In the entrypoint, add some code which creates a new **`TcpListener`** instance bound to localhost (127.0.0.1) on port 5000.
- You should then add code to call the **`Start()`** method of the **`TcpListener`** and then call its **`AcceptTcpClient()`** method.
- The **`TcpClient`** instance returned from **`AcceptTcpClient()`** should be assigned to a local variable for later use.
- Call **`GetStream()`** to retrieve the **`NetworkStream`** from the **`TcpClient`**.
- Call the **`Read()`** method to read in the first byte from the **`NetworkStream`**. This identifies how long the proceeding message is, in bytes (this restricts the message to a maximum size of 255 characters).
- Call the **`Read()`** method to read in the given number of bytes from the **`NetworkStream`**. This is the message to be translated. Convert these bytes to ASCII and perform the translation.
- Serialize the translated message back into bytes, add a byte to the start of the data to allow the client to determine how long the message is and **`Write()`** back out to the stream for the client.

Add a new Project to your Solution in Visual Studio. This will be the Client.

- The client requests a message from the user.
- When a string is entered the client should serialize the message to send to the server. The first step is to convert the string to a byte array using ASCII encoding.
- It should now add a single byte before the message to identify how many bytes are in the proceeding message.
- A connection should be made to localhost (127.0.0.1) on port 5000 by calling **`Connect()`** on a new **`TcpClient.`**
- Call **`GetStream()`** to retrieve the **`NetworkStream`** from the **`TcpClient`**.
- **`Write()`** the serialised message out to the stream for the server.
- **`Read()`** in the first byte from the **`NetworkStream`**. This identifies how long the proceeding message is, in bytes (this restricts the message to a maximum size of 255 characters). Then, the client calls the **`Read()`** method to read in the given number of bytes from the **`NetworkStream`**. This is the translated message.
- Convert these bytes to ASCII and output to the Console.



Sample code for a client and a server application is given below. This represents guidelines for you and an example of how the code of your client and server application may look.

Your task is to:

1. Implement the client and the server application in separate projects in the same Visual Studio solution.

2. Use the guidelines in Appendix A to make your client and set up both your client and server projects as the startup projects.

3. Complete the translation code to conform to the three rules:
   - If the word starts with a consonant, this consonant is transposed to the end of the word and the letters *'ay'* are appended. So "pig" becomes "igpay", while "latin" becomes "atinlay".
   - If a word starts with two or more consonants move all of the consonants to the end of the word and add "ay." So "plate" becomes "ateplay" and "llama" becomes "amallay".
   - If a word starts with a vowel add the word "way" at the end of the word. So "animal" becomes "animalway" and "empower" becomes "empowerway".

4. Extend the client and the server implementation so that the translated message is sent back to the client, read by the client and displayed to the user.

5. You may also experiment with transmitting the data between different machines by using the LAN IP address of your machine rather than the internal localhost loopback address (beware you *may* have some fun with firewalls if attempting this at home).

In a later lab we will look at ways to ensure the server can continue to accept new clients and ways to prevent the client from freezing whilst it waits for a server response.

SERVER:

```
using System;
using System.Net;
using System.Net.Sockets;
using System.Text;

TcpListener tcpListener = new TcpListener(IPAddress.Parse("127.0.0.1"), 5000);
tcpListener.Start();
TcpClient tcpClient = tcpListener.AcceptTcpClient(); NetworkStream nStream =
tcpClient.GetStream();

string message = ReadFromStream(nStream);
Console.WriteLine("Received: \"" + message + "\"");

string translatedMessage = Translate(message);

// TODO: Serialize the translated message and write it to the stream

Console.ReadKey(); // Wait for keypress before exit

static string Translate(string message)
{
    string translatedmessage = "TEST RESPONSE";
    string[] words = message.Split(' ');
    foreach (string word in words)
    {
        // TODO: Perform translation
    }
    return translatedmessage;
}

static string ReadFromStream(NetworkStream stream)
{
    int messageLength = stream.ReadByte();

    byte[] messageBytes = new byte[messageLength];
    stream.Read(messageBytes, 0, messageLength);
    return Encoding.ASCII.GetString(messageBytes);
}
```

CLIENT:

```
using System;
using System.Net.Sockets;
using System.Text;

TcpClient tcpClient = new TcpClient();
tcpClient.Connect("127.0.0.1", 5000);
NetworkStream nStream = tcpClient.GetStream();

Console.WriteLine("Enter a message to be translated...");
string? message = Console.ReadLine();

if (message != null)
{
    byte[] request = Serialize(message);
    nStream.Write(request, 0, request.Length);

    // TODO: Read response from stream and display to user
}

Console.ReadKey(); // Wait for keypress before exit

byte[] Serialize(string request)
{
    byte[] responseBytes = Encoding.ASCII.GetBytes(request);
    byte responseLength = (byte)responseBytes.Length;


    byte[] rawData = new byte[responseLength + 1];
    rawData[0] = responseLength;
    responseBytes.CopyTo(rawData, 1);

    return rawData;
}
```

Appendix A: Using Multiple Startup Projects in Visual Studio

When creating a Visual Studio project which produces an executable file as output, it is possible to run the executable that it produces in the debugger by pressing F5 or selecting *Start* from the *Debug* menu[2]. Such a project is known as the start-up project because execution of its output can be initiated by the Visual Studio environment.

If a solution contains more than one project then normally only one of the projects will be designated as the start-up project, even if more than one project produces an executable output. By default, the start-up project is the first project with an executable output added to the solution. The start-up project has its names displayed in bold typeface in the Solution Explorer window.

A key feature of networked and distributed applications is that they consist of more than one executable component. One of the most useful facets of networked and distributed applications is that they can be developed and tested on a single machine simply by running all of the application components on that machine. When developing such applications with Visual Studio you would normally create a single solution which contains separate projects for each of the components. When it comes to testing and debugging the application you want all of the components to start executing, not just the one that is designated as the start-up project. Fortunately Visual Studio allows the default start-up behaviour to be modified so that multiple executables can be started when the user presses F5 or selects Start from the Debug menu. It does this by allowing us to define multiple start-up projects and this feature is very useful in distributed application development. We will use multiple start-up projects in most of the distributed applications we develop with Visual Studio.

You can specify which projects should be executed when you start debugging by right clicking on the solution name in the Solution Explorer window and selecting *Properties*. In the solution properties dialog box that appears, click the *Startup Project* entry under *Common Properties*. A screen similar to that shown in Figure 2 will be displayed. To enable multiple executables to be run when debugging starts, click the radio button next to '*Multiple Startup Projects*' and use the entries in the table below to specify which projects from the solution should be executed and in what order.

When you select Multiple Startup Projects, all of the projects in the current solution are added to the table. The column to the right of their name contains a dropdown box which allows you to specify whether the project should *Start*, *Start Without Debugging* or *None* (meaning it is not executed) when F5 is pressed. Projects start executing in sequential order from the top of the list down. You can change the order in which projects start by highlighting a project in the table and using the 'Move Up' and 'Move Down' buttons to adjust its order in the list. When creating distributed applications you will often have some server components and some client components. The server components should appear in the list before the clients that call them. This ensures that the server components are actively executing before any client component tries to call them.

[2] It is also possible to start the executable without running it in the debugger by pressing Ctrl+F5 or selecting Start Without Debugging from the Debug menu

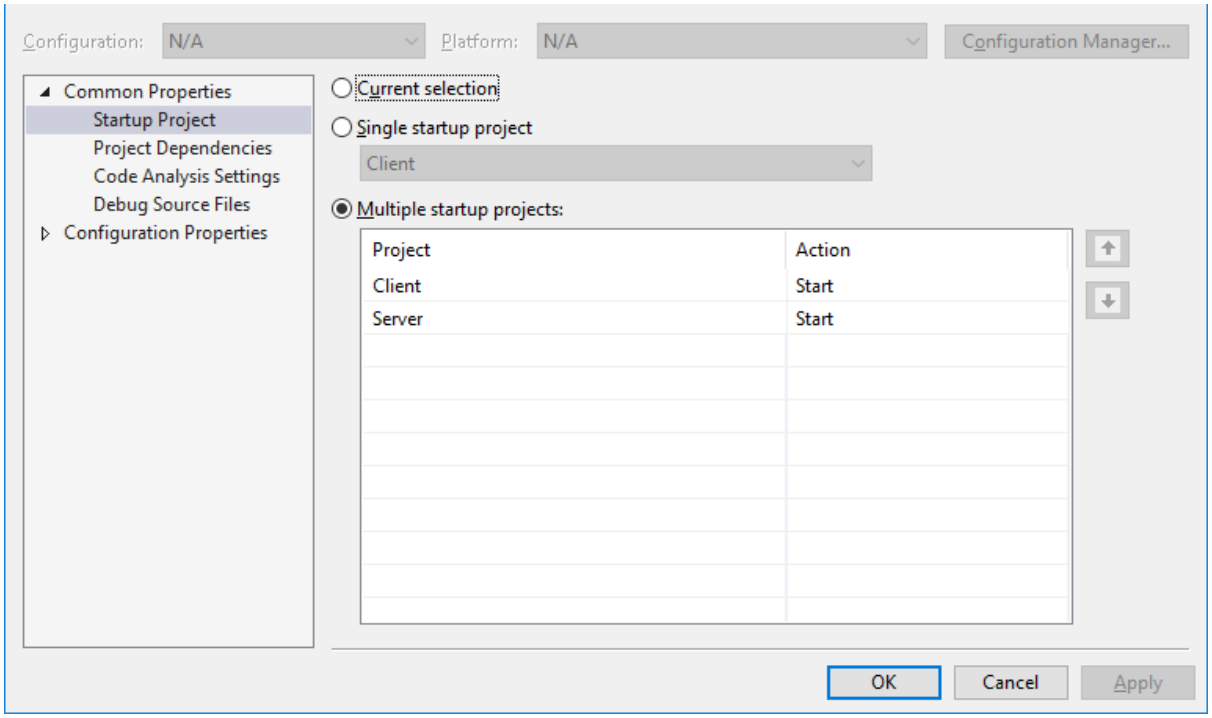


Figure 2. Setting multiple startup projects in Visual Studio Solution properties.