COMP260 – Distributed programming **Worksheet WK 2**

IP & Socket Programming

Introduction

Last week, we looked at sockets and implementations of client and server roles in a variety of languages (Python & C#).

This week we will develop some simple programs to experiment with sockets, looking at interoperability, robustness & threaded applications.

Fundamental socket programming

Last week's lecture, (slides 16-18), show the fundamentals of socket programming; creating sockets, connecting and binding, listening and accepting and sending and receiving data as bytes. If you haven't experimented with the code from this lecture, now is a good time to do so.

```python
import socket

if __name__ == '__main__':
    mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    mySocket.connect(("127.0.0.1", 8222))

    testString = "this is a test from the python client"

    mySocket.send(testString.encode())

    while True:
        data = mySocket.recv(4096)
        print(data.decode("utf-8"))
```

```python
import socket
import time

if __name__ == '__main__':
    mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    mySocket.bind(("127.0.0.1", 8222))
    mySocket.listen(5)

    client = mySocket.accept()

    data = client[0].recv(4096)

    print(data.decode("utf-8"))

    seqID = 0

    while True:
        testString = str(seqID) +":" + time.ctime()

        client[0].send(testString.encode())

        seqID+=1
        time.sleep(0.5)
```

1. Create two python projects (one for client & one for server) and implement the code samples taking care to share IP addresses and ports between both applications (otherwise they will not connect).
2. Do the same thing with the C# example code

```csharp
class client
{
    static void Main(string[] args)
    {
        ASCIIEncoding encoder = new ASCIIEncoding();
        byte[] buffer = new byte[4096];

        Socket mySocket = new Socket(AddressFamily.InterNetwork
                                    , SocketType.Stream
                                    , ProtocolType.Tcp);

        mySocket.Connect (new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8222));

        mySocket.Send(encoder.GetBytes("this is a test from the csharp client"));

        while (true)
        {
            int result = mySocket.Receive(buffer);

            Console.WriteLine(encoder.GetString(buffer, 0, result));
        }
    }
}
```

```csharp
class server
{
    static void Main(string[] args)
    {
        ASCIIEncoding encoder = new ASCIIEncoding();
        byte[] buffer = new byte[4096];

        Socket mySocket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);

        mySocket.Bind(new IPEndPoint(IPAddress.Parse("127.0.0.1"), 8222));

        mySocket.Listen(5);

        Socket client = mySocket.Accept();

        int result = client.Receive(buffer);

        Console.WriteLine(encoder.GetString(buffer, 0, result));

        var seqID = 0;
        while (true)
        {
            var testString = seqID.ToString() + ":" + DateTime.UtcNow;

            client.Send(encoder.GetBytes(testString));

            seqID++;

            Thread.Sleep(500);
        }
    }
}
```

NB: the server application will need to be running first in these applications, we'll address this shortly

Fundamental interoperability
Once you have working apps, you can switch the client and server code so that the Python client will run with the C# server and visa versa. If you change the port ID for each client server pair, both sets of applications will work together.

App Robustness & architectural design
With these simple skeleton apps, both will generate exceptions if the execution conditions are not correct. To address this, we can wrap 'troublesome' code with try / catch() exception handling. Run the code to see what the troublesome use cases are and build a state diagram of how the apps should work as robust (apps that don't throw uncaught exceptions).

Both client and server will need to consider connected and disconnected states.

App Robustness implementation
Take your design solutions and implement them (Python or C#). You will have found a workable solution when the client and server can be started, stopped and restarted without the other application throwing exceptions

Message flow & Packets
Consider the following Python program. We would assume that the server on the right will print out each message as it gets it and prints it with a newline. Write and run the program to see what happens.

```python
import socket

if __name__ == '__main__':
    mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    mySocket.connect(("127.0.0.1", 8222))

    index = 0
    while True:
        tmp = 'sending ... ' + str(index)
        mySocket.send (tmp.encode())

        index += 1
```

```python
import socket
import time

if __name__ == '__main__':
    mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    mySocket.bind(("127.0.0.1",8222))
    mySocket.listen(5)

    client = mySocket.accept()[0]

    while True:
        data = client.recv(4096)
        print(data.decode('utf-8') +'\n')
        time.sleep(0.5)
```

The messages are being combined as the recv() function is reading 4096 bytes from the socket, if there's several messages in that queue, they all just get combined.

Create a socket protocol that will send packets of data that start with the size of the packet, rather than just raw text, something along the lines of <size><data>. This can be sent with two send() calls, one for the packet size and the other for the data. Likewise, the server will need to recv() the size and then the number of bytes.

`to_bytes()` & `int.from_bytes()` will help with encoding and decoding data in Python.

Use the working Python code you have made to create a similar C# environment, this should be interoperable with the Python applications

Client Threading
Threading is a concept that allows multiple functions to appear to execute at the same time. Unity's Coroutines aren't threads in the computer science but give us functionality that it very thread-like.

Here's a simple example of a Python application that uses threads to run two processes as different rates.

```
import time
import threading

def doOtherStuff():
    index = 0
    while True:
        print('doing other stuff thread ' + str(index))

        time.sleep(0.1)
        index += 1


if __name__ == '__main__':
    thread = threading.Thread(target=doOtherStuff, args=())
    thread.start()

    index = 0
    while True:
        print( 'main thread ' + str(index))


        time.sleep(1)
        index += 1
```

Run it to see how time.sleep is being used to pause program execution in each thread and allow both functions to give the impression of running together.

Threads are extremely useful for client/server programming as recv() & accept are both blocking functions. Let's assume we have a client/server application where the client will send a message to the server but it takes some time for the server to respond to the client. Waiting for the server with a recv() function will block the client from doing anything. We can run the recv() function in a thread which will allow the client to carry on doing other work.

Use the code above and the code from the previous exercise to:

1. Get the server to send a receipt message back to the client once it has received a message
2. Create a client-side 'acknowledge receipt' function that will receive the server message using recv(). This will need to run in a thread.
3. Have the acknowledge function print whenever it receives a message

If your code works correctly, the client should continue to send lots of messages to the server and will periodically print out acknowledgements from your thread.

Server Threading

Re-architect your existing server to support multiple clients. It's worth running clients from a python command line so you can easily create lots of them to test your server.

The server code will need to create a thread to handle each client's messages that are sent with recv() to the server. The main thread of the server will just need to accept new clients and create a thread with their socket to then handle client messages and send them back to the client.