Distributed Computing Tutorial 9/**Assignment 2 Part C.**

Due date: 15 October Sunday 23.59 AWST

Marks: 15

This tutorial is the Part C of your assignment 2. **It is an individual submission.**

Have you ever wondered how Folding@Home works? Maybe you're more a fan of BitTorrent? We will build a simple peer-to-peer application that allows users to post jobs in Python. To achieve this, we're going to do a hybrid Web Service/.NET Remoting system.

The Applications

The parts you're going to need to build are:

- An **ASP.NET MVC Web Service** that will host a list of client machines and saves the relevant information into a local database
- A desktop application (**with NET Remoting**) that uses the Web Service to find and communicate with other clients.
- An **ASP.NET CORE Website** that communicates with the Web service to display the information as a dashboard.

The desktop application does require some libraries besides the Newtonsoft.JSON library and the RestSharp library you've installed before. You will also need Iron Python, which can also be installed by NuGet.

A. The Web Server (ASP.NET MVC Web Service with Local DB) [3 Marks]

The web server is the easiest part and probably the bit you'll want to do first. The Web Server's job is to allow client programs to find each other. This is a very common peer-to-peer infrastructure strategy, as clients are often on home networks hidden behind firewalls and NAT routers.

You will need to build a web service that can accept two kinds of requests. Clients need to be able to register themselves and post their IP address and the port they are hosting their .NET Remoting service on so the server can share it with others. Clients also need to be able to ask the server for a list of other clients so that it can connect to their IP addresses and ports to ask for jobs.

The best way to approach this is to build a static model as you did for the Data Tier in the previous tutorial. This needs to have a list of "clients", which will need, at the very least an IP address and a port.
Your controller will then need to allow for both requests (adding themselves to the list and requesting others). How you set the local DB is up to you!

You'll need to get clients to post to the Web server whenever they finish a job, and the server saves the information in the DB.

Make sure you test your system before progressing; it'll make debugging later easier.

B. The Client Desktop Application (with NET Remoting) [7 marks]

The client application needs to be a desktop application, just like the one back in Tutorial 2 and 3. You're going to need to add the Newtonsoft.JSON library, RestSharp library, and Iron Python libraries from NuGet.

There are going to need to be three separate threads:

- The GUI thread, which you don't have to make, it comes with the GUI.
- The Networking thread, which is going to connect to the server and other clients to find and do jobs, which you will need to make separate to the GUI via delegates or the async keyword (delegates do work here as you don't have to touch any GUI variables, but it's up to you).
- The Server thread, which will manage the connections from other clients. This is managed by .NET Remoting, so all you need to do is make sure you set up a server as you did in the second and third tutorials.

The GUI

The GUI is going to need to present a couple of things. Firstly it needs some means of users inputting Python code. This can be from a file, or directly input into a text box. This would need to be submitted to the Server thread, probably via some kind of static object.

The GUI will also need some way of displaying if it is working on a job in the background and how many jobs it has completed. You can do this with a specific button. You'll also need some way of querying the Networking thread (probably again with a static object) for how many jobs it's done and if it's currently working on one.

The GUI code is also going to have to start the other two threads on initialisation, as the GUI is basically the main for this program.

The Networking Thread

The Networking thread needs to do two things in a loop:

- Look for new clients

- Check each client for jobs, and do them if it can.

The first part needs to query the Web Service for a list of other clients. It will use this for the next part.

For each of the clients in the Web Service list (that are not the current client), the Networking thread needs to connect to the .NET Remoting server at the IP address and port in the list. It then can query if any jobs exist and download them.

Upon successfully downloading a job, the Networking thread needs to use Iron Python to execute the code (see below) and post the answer back to the client that hosted the job.

You'll need to get clients to post to the Web server whenever they finish a job.

The Server Thread

The Server thread is the client's job board, essentially. It needs to host a .NET Remoting service that allows other clients to ask what jobs are available, allows clients to download jobs, and allows clients to upload solutions or answers to those jobs, once they've gone ahead and done it.

You'll need a ServiceContract to achieve this, although because the jobs are (or at least are probably) static, and we don't want two clients trying to submit the answer at the same time... it may be a good idea to make the functions implementing the contract their own synchronisation contexts. Depending on how you feel about that, you may decide to turn on .NET's default service synchronisation system. See your second and third tutorial to figure out how to do that.

Assuming you get all these bits built, you'll have a fully functioning peer-to-peer system!

C. The ASP.NET CORE Website [3 marks]

Build a dynamic website that lets you see at a glance who is connected to the swarm and how many jobs they have completed! The website is dynamic that refreshes every 1 minute (use JS to load the page in regular intervals with a 1-minute delay). Note the website receives the information from the web server.

Stuff you might want to know

While most of this you should be able to figure out using the knowledge you've gathered during this semester, there are some parts you probably don't know about, and that will be handy for you to know. An example is Iron Python, most of you won't know Python, let alone whatever Iron Python is! Some of these will be useful to know into the future, especially if you end up building web services in industry.

Iron Python

Iron Python is a really neat library that allows Python to run on the .NET CLR. It allows for Python code to be managed, and directly interacted with, by the .NET VM. This is super useful for dynamic programming applications like the one you're building, as you can just input a string containing Python code.... And then execute it!

The downside is that Iron Python is way behind Python itself, so you have to remember that Python 2 is the only language that works.

To run Python code on Iron Python, you first need to have the script in some string variable. Something like the following, which adds two variables:

```
def test_func(var1,var2):
    return var1 + var2
```

Assuming you've got this in a string variable, you can run the code from inside C# with something like the following:

```
using IronPython.Hosting;

//....stuff here....//

int var1,var2;
var1 = 0;
var2 = 1;
ScriptEngine engine = Python.CreateEngine();
ScriptScope scope = engine.CreateScope();
engine.Execute(scriptIsInThisVar, scope);
//This uses C# Dynamic types. They can be *anything*
dynamic testFunction = scope.GetVariable("test_func");
var result = testFunction(var1,var2);
```

The resultant variable is then in the result variable.

As the Python code is running in the .NET CLR, exceptions from inside Python can transition (semi smoothly) into the C# environment, so you **can** wrap the Python code in try-catch statements. This is a very good way of checking for invalid code and program failures, which is necessary as you don't want someone else's buggy code crashing your perfectly happy program!

Base 64 Encoding

As some of you may have noticed in previous tutorials, sending file paths and URI's over JSON (or over the network in general) is fraught with danger. This is because the various characters in a path can confuse the computer on the other end. It could be a file path, or it could be escape characters... for example.

This problem can also rear it's head when dealing with sending files over networks, as they are represented as a series of bytes (chars) and can be confused with all sorts of things.

The best way to deal with this situation is to encode **anything** going over the internet (especially with Web Services) in a standard format that everyone understands, and that only uses characters everyone is absolutely sure aren't special in some way. HTTP GET variables do something like this using HTTP encoding, if you've ever seen something%20like%20this, you've already seen it. %20, for example, decodes to the space character, which is illegal in URLs.

The way that most applications encode data that may be affected by transit (or by internal mechanisms too) is by encoding that data in Base 64. This is so common that it's actually included by default in C#'s System library, as well as in Python, Javascript, Java, and basically every language more advanced than C.

To encode a given string (say, a chunk of Python code, or an IP address) in Base 64, you can use something similar to the following:

```
if (String.IsNullOrEmpty(textVar))
{
    return textVar;
}

byte[] textBytes = Text.Encoding.UTF8.GetBytes(textVar);
return Convert.ToBase64String(textBytes);
```

And to decode it back to a string, you can do the following:

```
if (String.IsNullOrEmpty(base64Text))
{
    return base64Text;
}

byte[] encodedBytes = Convert.FromBase64String(base64Text);
return Text.Encoding.UTF8.GetString(encodedBytes);
```

It's that simple! You *should* use Base 64 for complicated strings going over the network, as it protects them somewhat from anything going wrong. In this tutorial, you should aim to use it as an intermediary format for Python code.

Hashes for Verification (using SHA256)

Another problem with sending data across the internet, regardless of your encoding, is knowing nothing was lost or modified in transit. The easiest way of resolving this problem is to use cryptographic hashes to uniquely identify every string. Basically, if at the other end of the internet communication the hash and the string don't match, something went wrong, and you might want to try again.

To produce a hash, you can use the System.Security.Cryptography library in C#. We'll be using SHA256, as it's secure enough for at least the mid-future, and it's used by all kinds of things (it's really popular with open-source projects too!)

SHA256 hashes are stored as a byte array, so you'll need to keep that in mind when sending them around. To create one for a given string, you can do the following:

```
using System.Security.Cryptography;

///....Stuff here....//

string data = ;//something here
byte[] hash = sha256Hash.ComputeHash(Text.Encoding.UTF8.GetBytes(data));
```

To verify a hash, all you have to do is make a hash of the string you want to check and then make sure that the hash you made and the hash you were given are *exactly* the same. If they are, all good! If they aren't.... well... try getting that string again.

You should be using this for your Python code snippets, just in case they get mangled in transit. The best way to combine this with Base64 is to make the SHA256 hash based on the Base64 encoded text, *not* the raw text. This means you know if something is wrong *before* you try to decode it.

Other stuff to do

If you've gotten this far, you should try spinning up a few clients on your computer and see what happens. After that, add these features:

1. If a client closes, they stay on the server. This will cause problems (probably including crashes if you've not built your program with good enough exception handling), develop some way of getting the server to check if the client is still there, and to clear the list of "dead" clients. [**2 mark**]

Marking guidelines:

Full marks (100%): all the functionalities are executed without errors with the proper exception handles.

Partial marks (X%+Y%): X% functionalities are executed without errors with the proper exception handles. Y% Coding efforts are there for non-executable parts.

Partial marks (X%): X% functionalities are executed without errors with the proper exception handles. No Coding efforts are there for non-executable parts.

Partial marks (X%-1): X% functionalities are executed without errors; however, no exception handling.

The corresponding marks for each functionality are provided throughout the document between third brackets.

Submission Guidelines:

Submit your assignment electronically, via Blackboard, before the deadline.
To submit, do the following:

1. Fill out and sign a declaration of originality. A photo, scan or electronically-filled out form is fine. Whatever you do, ensure the form is complete and readable!
Place it (as a .pdf, .jpg or .png) inside your project directory.
2. Zip your entire project directory. Leave nothing out.
3. Submit your zip/tar.gz file to the assignment area on Blackboard.
4. Re-download, open, and run your submitted work to ensure it has been submitted correctly.

You are responsible for ensuring that your submission is correct and not corrupted. You may make multiple submissions, but only your newest submission will be marked. The late submission policy (see the Unit Outline) will be strictly enforced. Please note:

- DO NOT use WinRar.
- DO NOT have nested zip/tar.gz files. One is enough!
- DO NOT try to email your submission as an attachment. Curtin's email filters are configured to discard emails with potentially executable attachments silently. In an emergency, if you cannot upload your work to Blackboard, please instead upload it to Google Drive, a private GitHub repository, or another online service that preserves immutable timestamps and is not publicly accessible.

Marking Demonstration: You must demonstrate and discuss your application with a marker in a one-to-one online/in-person session. Most of the marks for your assignment will be derived from this demonstration. The demonstrator will ask you to rebuild and run your application (provided by the demonstrator) and demonstrate its major features. They may ask you about any aspect of your submission.

The demonstration schedule and policy will be published later (Check blackboard announcements). We may also cancel the demonstration-based marking due to unavoidable circumstances. In that case, it will be a full inspection-based marking.

Academic Integrity: This is an assessable task. If you use someone else's work or obtain someone else's assistance to help complete part of the assignment that is intended for you to complete yourself, you will have compromised the assessment. You will not receive marks for any parts of your submission that are not your original work.

Further, if you do not reference any external sources, you are committing plagiarism and collusion, and penalties for Academic Misconduct may apply. Please see Curtin's Academic Integrity website for information on academic misconduct (which includes plagiarism and collusion).

The unit coordinator may require you to provide an oral justification of or to answer questions about any piece of written work submitted in this unit. Your response(s) may be referred to as evidence in an Academic Misconduct inquiry.