Distributed Computing

Tutorial 4 and 5 Marks: 30, Due date: due date: Sunday, 10 September 2023 23:59:00 AWST

Reading material:
https://www.c-sharpcorner.com/uploadfile/vendettamit/delegates-and-async-programming/
https://www.c-sharpcorner.com/UploadFile/vendettamit/delegate-and-async-programming-C-Sharp-asynccallback-and-object-state/
https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/

What We're Doing Today

1. Adding a Business Tier to your application from last week
2. Adding a search function
3. Adding an access log for the business tier.

Introduction

Well after last week you should feel like you have a hang of this windows programming thing. Lets take things up a notch and do some more fun things with your network application. You're going to need to work a little harder here and figure things out for yourself, so I suggest you take a look at the code from this week's lecture for guidance… and ask your tutor if you get stuck!

Task 1: Adding a Business Tier [5 Marks]

First things first, you need to add a Business Tier. This means you'll need to create a new Console project inside your solution first. This console program needs to sit between your GUI and your Data tier project.

Your business tier is going to look a little like your Data tier, presenting an interface and an implementation to your GUI… however it's also going to have code reminiscent of your GUI in the implementation of those functions, as they are going to call the functions in the Data tier.

You will want to start by creating a new public interface called `BusinessServerInterface` that replicates all the functionality of the Data tier.

Then, you'll want to create an implementation called `BusinessServer` that for each function creates a `ChannelFactory<DataServerInterface>` and calls the required functions on the Data tier.

After you've written this, change the reference on your GUI to point to your Business tier rather than your Data tier, and change the `ChannelFactory<DataServerInterface>` to a `ChannelFactory<BusinessServerInterface>` accordingly.

If you've done it right, spinning up the Data tier, then Business tier, and *then* GUI will allow the program to run nicely.

If you don't start the Business tier, the GUI should consequentially not be able to work.

Once you've got the Business tier between the GUI and the data tier, you can move on.

Task 2: Search [10 marks]

You're going to add a search capability to your Business tier!

Setting up the Business Tier for search

The business tier needs to implement a function that will take in a string and return the contents of the first record that has a last name matching the string.

First things first, we're going to make this a hard thing to do, make sure your Data tier generates at least 100,000 records in it's database.

Next, implement the function. Remember to include it in the Business tier interface so that the GUI can actually use it!

Once you've got that all done, move on to…

Adding to the GUI so you can search

You will need to (at the very least) add a text box and button that will allow you to search the database.

The on-click function of the new button you add should take the string, call the search method of the Business tier via RPC, and then set the values of the other text boxes to the values returned by the function.

You will note when you try to do this for the first time that it takes a little bit of a while, and it freezes your GUI when you do it. This is the perfect kind of function to use an asynchronous call for!

Making it asynchronous

You will want to create a delegate.

A delegate is basically a specialised function pointer used by C#. So long as it is of a delegate type with the same function parameters as a function you want to call, it will allow you to call it without actually calling the function.

An example, for a calculator class as follows:

```
namespace MyExample {
   public class Calculator
   {
      public int Add(int num1, int num2) { return num1 + num2; }
      public int Subtract(int num1, int num2) { return num1 - num2; }
      public int Multiply(int num1, int num2) { return num1 * num2; }
   }
}
```

I can create and use a Delegate as follows:

```
public delegate int BinaryOperation(int operand1, int operand2); ← Delegate definition

public class DelegateTester
{
   public static void Main() {
      BinaryOperation binOp;                                    ← Declare a variable of delegate type
      Calculator calc;

      calc = new Calculator();
      binOp = calc.Add;                ← Point at Add() function of calc (the object, NOT the class!)
      System.Console.WriteLine("1 + 2 = " + binOp(1, 2)); ← binOp substitutes for calc.Add
      binOp = calc.Subtract;                                    ← Point at calc.Subtract()
      System.Console.WriteLine("1 - 2 = " + binOp(1, 2)); ← binOp now substitutes for calc.Subtract
   }
}
```

I can also Invoke a delegate to make the function execute in a separate thread:

```
public delegate int BinaryOperation(int operand1, int operand2);

public class CalcClient
{
   public static void Main() {
      BinaryOperation addDel;
      Calculator calc = new Calculator();
      IAsyncResult asyncObj1, asyncObj2;          ← Each IAsyncResult object tracks info for a single async call
      int iAddResult1, iAddResult2;

      addDel = calc.Add;                          ← Point delegate at function to be called asynchronously

      asyncObj1 = addDel.BeginInvoke(1, 2, null, null);     ← Start worker threads
      asyncObj2 = addDel.BeginInvoke(3, 4, null, null);       for async calls in parallel

      // Block, waiting for all async calls to complete
      iAddResult1 = addDel.EndInvoke(asyncObj1); ← Retrieve return value from each async call tracked by the
      iAddResult2 = addDel.EndInvoke(asyncObj2);    appropriate asyncObjX

      asyncObj1.AsyncWaitHandle.Close();          ← Clean up
      asyncObj2.AsyncWaitHandle.Close();

      System.Console.WriteLine("1 + 2 = " + iAddResult1);
      System.Console.WriteLine("3 + 4 = " + iAddResult2);
   }
}
```

And I can even get those threads to perform callback functions, which execute asynchronously once the delegate finishes.

```
public delegate int BinaryOperation(int operand1, int operand2);

public class CalcClient
{
   public static void Main() {
      BinaryOperation addDel;
      AsyncCallback callbackDel;                    ← AsyncCallback is defined by .NET for BeginInvoke()
      Calculator calc = new Calculator();

      addDel = calc.Add;
      callbackDel = this.OnAddCompletion;           ← Point callback delegate at callback function
      addDel.BeginInvoke(1, 2, callbackDel, null);          ← Start worker thread (*callbackDel note*)

      System.Console.WriteLine("Waiting for completion.");   ← Race condition with result output below
      System.Console.ReadLine();
   }
   private void OnAddCompletion(IAsyncResult asyncResult) {  ← Must be same signature as AsyncCallback
      int iAddResult;
      BinaryOperation addDel;
      AsyncResult asyncObj = (AsyncResult)asyncResult;      ← So that we can get the AsyncDelegate

      if (asyncObj.EndInvokeCalled == false) {      ← Must not call EndInvoke more than once!
         addDel = (BinaryOperation)asyncObj.AsyncDelegate;   ← Gain access to delegate for EndInvoke
         iAddResult = addDel.EndInvoke(asyncObj);            ← Would retrieve ref or out params here too
         System.Console.WriteLine("1 + 2 = " + iAddResult);  ← Race condition with result output in Main()
      }
      asyncObj.AsyncWaitHandle.Close();  ← Clean up
   }
}
```

This last one is the most important. What you can do is put the GUI into a "waiting state", which is still responsive, but doesn't let the user do anything... and then wake it up using the completion callback.

You may want to add a progress bar to your GUI. When your search button is clicked, set your text boxes IsReadOnly to true, disable the buttons on your GUI, and set your ProgressBar's style to Marquee, then call the RPC function for search on the business tier.

In the completion callback, set all the values of your text boxes, then set IsReadOnly to False, enable all buttons, and set the ProgressBar's style to Continuous.

Congratulations! You've made a responsive application using asynchronous delegates! I have added a

solution "delegateExample.zip" in the Lecture 4 as well. It will come handy.

Task 3: Making the same functionality using Async, Await and Task [5 Marks]

Create a new client APP using WPF. This program will do the same as your previous client. However, this time you will use Async, Await and Task. This is another way of making program asynchronous. I have added a solution AsynHandle.zip for your reference in Lecture 4. Note: this is a simple and cleaner way to handle threading in other languages as well, e.g., Javascript's 'promise'.

Task 4: Adding an access log [5 marks]

Last thing to do this week is to add an access log to the Business tier so we know what is happening.

This is in general a very good thing to do with Business tier applications, as the logic generally has business implications, and executives like to have something to look at to know that their applications are actually doing the right thing.

What we're going to do is build a `void Log(string logString)` function for *every* function that can be called on the Business tier. This will use a class variable `uint LogNumber` that will keep track of how many log-able tasks have been performed. This function should *not* be in the interface, as you do *not* want clients calling it directly. It's just for the Business tier itself.

Your function needs to write out how many tasks have been performed thus far, and the log string that was sent to it. This string needs to be *informative*. As informative as possible. Put as much information as you can into it.

As you can probably guess, there could be issues if multiple clients try to run this function at the same time, and it's likely going to happen that way as every function in the Business tier will call the log function. So we need to do some automatic thread synchronisation.

Thankfully, .NET has our back with some auto-thread-sync tools. Tag the function you write with the attribute `[MethodImpl(MethodImplOptions.Synchronized)]`. This will tell .NET to only allow one thread at a time to run this function, and thus automatically sync it for you. Handy!

Task 5: Cleaning up [5 marks]

As before, make sure you have good exception handling and can deal with unusual input.

What if for example:

1. Someone searches a name with no match?
2. Someone tries to search a number? Or special character?
3. Something goes wrong during the search? Maybe you should implement a timeout for the asynchronous call, just in case.