Using Vsync from IronPython

Ken Birman

The Vsync system can be used from IronPython with the same ease as from C#, and in fact in nearly the same style of coding. IronPython is a high-quality version of Python that operates by translating programs into the .NET byte code representation, which is then executed. This translation occurs dynamically, hence IronPython can be thought of as a compiler or as an interpreter. It can be used in command line mode or used to create executable programs that will later run autonomously, just as you would with any version of Python. No special secret knowledge of the .NET environment is needed. The full Python language is supported and performance is said to equal or exceed that of other Python implementations for the same platforms, so there is no "downside" to the .NET embedding.

From the perspective of the potential Vsync user, this particular version of Python is fully interoperable with the Vsync library and hence we consider it to be one of our supported programming language options. One simply needs to understand how calls from IronPython to Vsync need to look, and there are also some very minor limitations that the user must be aware of.

Setting up to use Vsync from an IronPython Application

As first steps, the developer should download and install IronPython on their .NET system. The download is free and can be done from the codeplex.com web site maintained by the IronPython team. We'll assume that you are using Visual Studio 2010 and that you also download and install the Iron Python for Visual Studio extensions. Make sure to use current releases of all components; we make some use of "modern" features.

Because your program will be using the Vsync library, you must locate the library dll file and add it as a "reference" to your IronPython application. Accordingly, create a new IronPython project (File->New Project->IronPython Project) and then go the "Solution Explorer" window, which normally is displayed on the right hand side of the Visual Studio 2010 interaction GUI. Right click Project in Solution Explorer, select Open Folder in Windows Explorer Copy VsyncLib.dll to the opened folder. Right click the project node, click Add Reference, and then go to the Browse tab. Double click VsyncLib.

Next, open your .py file and add:

```
import clr

clr.AddReference('VsyncLib')
```

For better Python debugging, these additional steps are useful: Right click the project node, select Properties, select Debug, and change the Launch Mode to Standard Python launcher. And now you're all setup!

Building an IronPython program that uses Vsync

To build your program, there will be several simple steps. First, you need to understand how Vsync system calls look when coded from IronPython. We'll see that the notation changes in small ways but that there is a one-to-one correspondence for nearly everything.

Next, you need to understand the single restriction that currently applies: IronPython datatypes all look like a single list object to Vsync, and although we do understand how we could perhaps do better, right now this means that when creating new data types to register with Vsync you will need to build a small helper file coded in C#. This is a very minor task that involves just a few lines of code and you won't find it difficult to understand or carry out. The issue only arises if you need to register new types that Vsync would encounter within messages, or as arguments to multicast sends and handlers. If you work only with existing data types, you can ignore the issue entirely.

Last, there is a small magic incantation that your program must execute when launching itself, to initialize Vsync. This is true of any Vsync application and the IronPython version simply looks slightly different.

The discussion that follows assumes that you have a copy of the Vsync documentation handy, with all of those baffling C# code examples. We're going to show you how to code the same kinds of things in IronPython, but we decided not to write a whole new version of the documentation just for the IronPython community, or at least to do that some other day. So you'll need to get used to switching back and forth, but on the positive side, the languages are similar enough to make this very easy to do.

How Vsync looks from within IronPython

IronPython has a notion of an external package from which an API can be imported. Vsync is used by invoking this machinery. This entails a few lines of code to load the common-language-runtime access library, which is called the "clr", and then two more to load Vsync itself:

```
From System import Action
import clr
clr.AddReference('VsyncLib')
import Vsync
from Vsync import *
```

Note that these lines of code will throw runtime exceptions if you skipped the steps that add the Search Path and VsyncLib reference to your IronPython project, as mentioned on the first page. Also, these particular lines do assume that the name of the Vsync dll is VsyncLib.dll. If you rename the library then obviously you must edit these lines. We suggest that you not rename the library: it may confuse people

later if you hand your work off to someone else. Finally, be aware that if your search path is on Windows and includes special characters, search breaks (an IronPython bug). For example, can you see what would go wrong if your search path were "C:\Users\ken\Desktop\Vsync\bin\Release"? (Hint: \b has a special interpretation in Python… ☹).

A restriction on creating new data types

In the Vsync manual, one of the first things that the reader learns about is the idea that internally, Vsync moves data around in a special format using an object called an Vsync "Msg". The Msg class has a means of serializing and deserializing application-defined classes (new data types), and if your application defines data types of its own, and you want to pass them into Vsync or use them as parameters to group event handlers, those types must first be registered with the Vsync runtime.

IronPython has classes too, and they function much like C# classes. Nonetheless, you cannot register an IronPython class with Vsync at present. The problem is that because IronPython objects are highly polymorphic (you can add fields dynamically, and the notion of type is very fluid), our C# code has difficulty understanding the IronPython classes and objects and is not able to marshall them. We've considered several ways to solve this and settled on the one that seems simplest to us: if you really want to add additional data types to your application (rather than just working with the existing types, which Vsync does understand – things like int, float, string, arrays, etc), you should create a C# dll for that purpose. Just create a new C# "class" from visual studio, define the class in it and have a static initializer that registers the type with Vsync, and then import that class just as we imported the VsyncLib.dll. Then you can use these objects from within IronPython.

For example, you could write a method in C# called GenerateAByteArrayFromMyFoo(), or GABA() for short, and pass it the fields of your Foo object. If the fields are base types, like integers, Booleans, floats or strings, this will work, and you can then have your GABAFMF method call Msg.ToBArray from C#, where everything works properly. Later, from Python, you could call UnpackAFoo(), or UAF(), and have it return a vector of objects. Again, this should work. This is because IronPython has compiled-in code for passing base types between your IronPython code and C#, and back. But you will *not* be successful in teaching Vsync to directly understand IronPython types without this sort of intermediary method coded in C#. The underlying IronPython object representation is simply very complicated, not documented, and hence inaccessible to C#.

The good news is that with little trouble, you can mix C# with IronPython in a single application, and even a single group. The data types used in C# mostly have direct cousins in IronPython and the converse also holds. Just keep in mind that every process that joins a given group must set up all the handlers for that group before joining it, and must have handlers with the same type signatures.

One limitation to be aware of is that although Vsync understands most C# base types, it does not understand list types. Thus in practice, you will be restricted to using base types, arrays of them, and the handful of types that Vsync itself defines, such as the Address and View types. This is probably adequate for most applications. Fancier ones would need to use the C# helper approach, but we doubt that doing so will be a common need.

The alternative that we ruled out would have involved coding a helper method that could answer some of the questions Vsync needs to pose when it sees an object and must make sense of its type. We would have Vsync call the helper, and the helper would have methods that take each of the needed actions. But this was complicated and when we tried mocking it up, slowed the Vsync system down more than we were comfortable with. Our solution does force you to either make due with existing types or work with a few lines of C#, but on the positive side, it works well and is quite fast.

Launching Vsync

Once you've imported the Vsync namespace you can launch the system, as follows, This will take a few seconds if Vsync isn't already active, but should be fast if the system is running on your network using the same "port numbers" (see the full manual for an explanation of how these work).

```
VsyncSystem.Start()
```

At this point your application can issue calls to any of the Vsync API methods.

Doing Vsync calls from within your IronPython code

The basic approach to using Vsync involves defining functions that will handle any upcalls from Vsync. For this you need to both define the handler function and to register it. That registration step involves a special API we've added for use just from IronPython that tells Vsync which type signatures match the upcall. A single IronPython function can be registered many times with multiple type signatures if you wish to take full advantage of IronPython's untyped function API options. The following code snippet illustrates the approach. First, we declare the upcall handler functions, like this:

```
def myfunc(i):
    print('Hello from myfunc with i=' + i.ToString())
    return
def myRfunc(r):
    print('Hello from myRfunc with r=' + r.ToString())
    g.Reply(-1)
    return
def myViewFunc(v):
    print('New view: ' + v.ToString())
    print('My rank = ' + v.GetMyRank().ToString())
    for a in v.joiners:
        print('  Joining: ' + a.ToString() + ', isMyAddress='+a.isMyAddress().ToString())
    for a in v.leavers:
        print('  Leaving: ' + a.ToString() + ', isMyAddress='+a.isMyAddress().ToString())
    return
```

Now we can register them as handlers for some group, this way:

```
g.RegisterHandler(0, Action[int](myfunc))
```

```
g.RegisterHandler(1, Action[float](myRfunc))
g.RegisterViewHandler(ViewHandler(myViewFunc))
```

Finally, we can use the Vsync message sending APIs (all of them are available) to do point-to-point, multicast and multi-query requests,. Here are examples that illustrate the general approach, with those three lines now shown as part of the setup for a group called “FooBar”:

```
g = Group('FooBar')
g.RegisterHandler(0, Action[int](myfunc))
g.RegisterHandler(1, Action[float](myRfunc))
g.RegisterViewHandler(ViewHandler(myViewFunc))
g.Join()
g.Send(0, 17)
res = []
nr = g.Query(Group.ALL, 1, 98.8, EOLMarker(), res);
print('After Query got ' + nr.ToString() + ' results: ', res)
```

As you can see, this call results in an upcall to the myRfunc method, with a float as its argument. That method replies (in this case always sending an integer valued -1… a boring reply) and then a list of these -1 values is received by the application that issued the Query. We didn’t use the fancier Vsync timeout features but we certainly could have done so. Everything shown in the C# version of the manual seems to work without problems in our tests of the system.

Notice the “Action” and “ViewHandler” functions in the register statements. These are your way of telling Vsync that the argument (myfunc, myRfunc, or myViewFunc) supports some particular type signature. You can call Action many times for the same function and it can have a list of types, not just [int], as its arguments. Thus, for example, one could call Action[int,string,float] to tell Vsync that some function can handle upcalls that pass in an int, a string and a floating point number. This is obviously not as flexible as the full IronPython function calling options can support, but you can do quite a lot this way. A single function can be registered as many times as you like.

Notice that although C# generic methods (those that take types as arguments) look different when called from IronPython, they are all available. Thus, the Vsync aggregation operations, which are used in very large groups (those that might have thousands of members) can be called by just passing the desired types using the func[type] notation seen above rather than the func<type> notation used in C#.

Once your program has set up groups and joined them, many applications will need to “wait” for things to happen. We do this by calling

```
VsyncSystem.WaitForever()
```

IronPython supports multithreaded applications. As long as the thread that started Vsync is waiting (and there are many ways to wait, although this is a good one), the remaining threads can do anything you like.

Drop us a note if you have any other questions, but in theory with these simple mechanisms and the Vsync manual at your side (think of the above as a Rosetta stone and just pretend that you don't mind reading C# notation), you should be up and running in minutes. Good luck, and drop us a note if you have any problems at all!

Ken Birman

ken@cs.cornell.edu; http://www.cs.cornell.edu/ken

http://Vsync.codeplex.com