Subject: Re: XSTRETCH and Library Lessons
Posted by JD Smith on Tue, 25 Apr 2006 22:29:50 GMT
View Forum Message <> Reply to Message

On Tue, 25 Apr 2006 15:05:45 +0000, Michael A. Miller wrote:

```
>>>> >> "JD" == JD Smith <jdsmith@as.arizona.edu> writes:
>
>>  On Mon, 24 Apr 2006 19:36:29 +0000, Michael A. Miller
>>  wrote:
>
>>>  For our local libraries, I've had to define release tags
>>>  for them.  Then, every "application," which means "every
>>>  thing that we expect to work the same way each time, gets
>>>  started from a script that includes setting the IDL_PATH
>>>  to include the proper release.
>
>>  This is a very heavy handed approach, because it requires
>>  your colleagues to use only your package in a given IDL
>>  session, and not mix and match.  It is, alas, an approach
>>  many people take.
>
>  I don't see how that is heavy handed.  The IDL_PATH can handle
>  more than one directory at once and users are they are welcome to
>  add anything they like to their IDL_PATH.  We regularly use
>  applications that use out libraries, Fannings, mpfit, textoidl
>  and the Juelich libraries, all in one applications.
>
>  The whole point of the IDL_PATH is to allow flexible loading, so
>  multiple "packages" can be used.  You are right that they get
>  only one version of a particular routine, but that is the whole
>  point.  If parts of a library don't change, then it doesn't
>  matter which version they use.  If parts do change, then they
>  pick which behavior they want.  This is no different with IDL
>  than with any other system, be it put together with a linker or
>  an interpreted language.  The same sort issue comes up all the
>  time for libc, for python, for java, for <insert system here>,
>  especially around major release increment time.
```

I think we are talking about different problems. You seem to be worrying about name space conflicts among multiple versions of your own library lying around on disk. I'm talking about name collisions between libraries (either because they have a real, distinct routine with coincidentally the same file name as another's routine, or they copied a possibly now incompatible version of that routine into their distribution).

The problem as I see it is making the assumption that simply shuffling

the IDL_PATH to put *your* library's path(s) up front is a solution.
Yes, it allows your library to run correctly. However, if the reason
you shuffled the path in the first place was to avoid unfortunate name
space conflicts with other user-installed packages, you have just
shifted the breakage from yourself onto them. Ideally, the ordering
of a package on IDL_PATH wouldn't matter, especially since this
ordering is alphabetical in recursive additions to the path.

I guess you could just name your library AAAAAAAAAAAAAAAAAAAAAAGOODLIB
to ensure it is sorted first ;).

>> Assuming library coders kept a (quasi-)fixed calling
>> interface and backward-compatible behavior for their
>> routines (which is mostly true of most of the big
>> libraries), the best approach would be if:
>
> What I was presenting is how I handle the case where backward
> compatability is broken (sometimes willfully, sometimes
> unintentionally).

You seem to be attacking the "intentional multiple installed versions
of the same library" issue, which is something related but different.

>> 1. External libraries are mentioned, by version number
>> required, and the user or site has the responsibility to
>> install them.
>
> I think that is exactly what we do here. Would you elaborate on
> what you mean by "install," if it doesn't mean, make sure IDL can
> find them by setting the appropriate the IDL_PATH?

I.e. instead of just bundling another library directly as part of your
package, just ask the user to go to the source and install it from there.
More work for you and them, yes, but far less likely to blow-up on you.

>> 2. Everyone uses likely-to-be-unique names for their
>> routines... object programming helps here (since it's not
>> weird seeming to hide everything behind a long unique class
>> name).
>
> Absolutely required - only gets hard-ish as multiple incompatible
> releases get promulgated, which is the case that I was talking
> about, as was the original poster (who, now that I look back, was
> you :-)

Not actually. I guess there's a bit more subtlety involved. There's
the pro-active "my sysadmin put 3 different versions of AstroLib"

issue, which is solvable locally. Then there's the "I took the parts
of AstroLib which I wanted and put them together with my library and
distribute that to users" issue, which isn't.

>> 3. Nobody messes with IDL_PATH via shell scripts or IDL
>> scripts. Your package should work no matter where it is on
>> the path, and should not make specific assumptions about
>> where it is in the heirarchy.
>
> How would IDL find the code then? If I don't mess with (= add
> the neccessary directories to) the path, my package cannot work,
> becuase IDL can't find it. If there are multiple versions of a
> routine, or even just one, there must be some way for IDL to find
> the code. Whether that is handled by using the built in IDL_PATH
> mechanism, or some new feature that is invented to replace it, it
> seems unavoidable. I must be missing something here - would you
> elaborate?

The *user*, and only the user, sets IDL_PATH. He may set it to
+~/idl/, or to something way more fancy, but code never gets to monkey
with it. I have seen plenty of packages that, on starting up (with a
required startup batch or via a shell script), first directly modify
!PATH to ensure they are in front. Essentially the equivalent of
cutting in line.

> Actually, I'll bet I'd always get the "blah" that I specified,
> regardless of what I wanted! If I specified the wrong version,
> I'd still get the wrong version ;-)

But at least then it would be your fault, not the fault of a library
packager ;).

>> Note this works as well for normal procedural programming
>> as object-oriented programming. It also makes it trivial to
>> "fork" a version of a library, and re-distribute. So you
>> might have to change "Package AstroLib" to "Package
>> AstroLib-FooBar" and reference that package in your code
>> instead. Our only equivalent would be to go through and
>> change all the routine definitions and calls to routines in
>> the library from routine to foobar_routine. Not exactly
>> maintainable.
>
>> Sadly, IDL doesn't really offer any help like this, so it's
>> up to the community to approximate, by convention, a system
>> with some of these properties.
>
> I see it a bit differently - IDL offers a simple method to
> specify which fork I want. If I want AstroLib, I put a !path =

> '/dir/AstroLib'+!path in my code. If I want AstroLib-FooBar, I
> put a !path = '/dir/AstroLib-FooBar'+!path in my code. Both
> AstroLib and AstroLib-FooBar contain do_this.pro and do_that.pro,
> so I continue to call them without changing my code. This is
> easy to do for any IDL code if I know where the codes are
> installed. I don't know how to do it if I adhere to your point
> 3.

That's a solution for end users, but it doesn't solve the problem generally, and doesn't help at all people who are distributing their own packages, since it's completely non-portable. What if you want to distribute a package which uses AstroLib-FooBar? Will you know where it is on your user's machine, so you can modify the !PATH directly like this? Will you know what else will already exist on their !PATH? The beauty of a real package system like I mention is it's portable.

If I, the end user, know about all the name space conflicts in my installed pile of IDL libs, I can hand tune my IDL_PATH to ensure that, despite the name collisions which occur, the version I want to be called first is actually getting called. But this is an enormous amount of work, and users just won't do it. If you end up downloading another package FOO which bundles a few outdated routines from AstroLib, and puts itself at the head of !PATH, I think you'd quickly realize what I mean ;). And, despite having written a tool that helps identify routine shadowing, I still get bitten by it far too often (e.g. in the original XSTRETCH example).

Another way of thinking about it... how would you recommend fixing the originally proposed XSTRETCH problem using !PATH alone? The solution has to allow me to run XSTRETCH, and the other random code which contain shadowing routines (by now quite obsolete and incompatible), all in the same session.

JD