Should a build system allow multiple versions of the same component?

Gene Michael Stover

created Monday, 2006 June 26
updated Monday, 2006 June 26

Contents

1 What is this?

My friend, Pavel, author of at least two in-house build systems, asked me whether I thought a build system should allow multiple versions of a single component in a single build.

After discussion with Pavel & thought on my own, here's my answer.

2 What was the question again?

Here's a better description of the puzzle posed by Pavel.

Your application consists of multiple programs, scripts, libraries, data files, Corba/COM/whatever components, & maybe other things. I'll refer to any of these as *parts*. A program is a part. A statically linked library is a part. A shared library is a part. A script is a part. A . . . You get the idea.

For each part, your build system allows you to specify other parts on which your part depends. You also specify the version on which your part depends.

For example, let's say that your application contains two parts: *Server 1.0* & *Client 2.3*. You tell that to the build system. You also tell it that *Server 1.0* is built from. . .

- Protocol version 3
- Splay-Heap version 2
- Cache version 1

Furthermore & elsewhere, you tell the build system that *Client 2.3* is built from. . .

- Protocol version 3
- Splay-Heap version 2
- Cache version 2

Notice that both Server & Client depend on Protocol version 3. That's fine. They both depend on Splay-Heap version 2; again, that's fine.

Both Server & Client depend on Cache, but they depend on different versions of Cache. Should the build system flag this as an error & refuse to build anything, or should it emit a warning & continue?

In case it matters, Server & Client might both have to run on the same computer. Just because I called them Server & Client does not mean that Server runs on a single computer & Client runs on multiple computers, most of which are located many kilometers from Server. I wish I could have imagined two appropriate part names which did not carry these same implications, but I couldn't.

I definitely do mean to imply that Server & Client communicate with each other.

3 My answer

My answer is "The build system should flag the Cache version mis-match as an error & refuse to build the application".

The reason it should be an error is that, if you allow multiple versions, it increases the likelihood that you will allow an increasing number of multiple versions until some situation forces you to update the dependent parts. The longer it takes to find that situation, the more versions of a part that exist in the system, the more likely that the update will require much wailing & gnashing of teeth.

For example let's think about the mythical Cache part, above, & let's say that you allow multiple versions. By definition, that means you have at least two versions of Cache. Call them 1 & 2. Later, you create Cache 3. Because your project is in a hurry, someone is lazy, or everyone agrees that it won't hurt if you don't upgrade all parts to use Cache 3, you allow Cache versions 1, 2, & 3 to co-exist. By the same process, you later add Cache versions 4 & 5. Now you have a system in which five versions of Cache co-exist.

Then an upgrade to, say, part A, which uses Cache 1, requires a feature that's already implemented, debugged, & tested in Cache 5. Ideally, you'd upgrade part A to use Cache 5, but apparently such an upgrade isn't trivial or you would have done it earlier, when you created Cache 5. In fact, since Cache 1 is sooooo old in comparison to Cache 5, the upgrade might be painful. What's more, what happens to parts which depend on part A if you upgrade A to use Cache 5? Whether it'll break anything or have no effect, can you predict what will happen if you upgrade part A to use Cache 5? Can you say "Oh jeez, why didn't we upgrade part A to Cache 5 as soon as Cache 5 was ready?"

Okay, so part A, which uses Cache 1, has a requirement for a new feature in Cache. The feature is already in Cache 5, so you assign the task to the new guy. He's a good programmer with a lot of experience even if he's only been on the project a couple of weeks.

It turns out that the new guy doesn't even know there's a Cache version 5. The new feature is easy to implement, so he does implement it... in Cache version 1. This creates Cache version 1.1. Now the application contains six versions of Cache: 1, 1.1, 2, 3, 4, & 5. Version 5 descends from 4 descends from 3 descends from 2 descends from 1. Version 1.1 descends from 1 but is unrelated, parallel, to the others. Oh boy. Godalmighty jeezus marry & yoseph. We have parallel versions of a single component. This is bad. Just shoot me now, please.

"But just because you allow multiple versions doesn't mean this will happen." Yes yes yes, that's right, but if you don't allow multiple versions, these things are much less likely to happen.

Notice that each & every one of these situations I've described ends in pain for the development team. It's always the kind of pain where biting the bullet & fixing it early would have been easier than fixing it later. So fix it early, & don't allow multiple versions.

4 Sometimes multiple versions don't make sense at all

Assuming that Server & Client communicate via a Protocol part, it would make little sense for them to use different versions of Protocol. Even if the later version of Protocol was backward-compatible with the earlier version, there's little point in building a new version of the application if only one side of the communication channel used the new version of the Protocol.

Notice that I'm not saying that later versions should not be backward-compatible with earlier versions. Neither am I saying that it's reasonable & possible to wipe all earlier versions from the field of installations. Backward-compatibility is definitely a good feature, mainly because older installations exist & there's little we can do about it.

I'm talking about compiling new versions of the whole application. In that case, with some parts, multiple versions simply don't make sense.

5 But multiple versions do exist on the same computer

Yes, on the same computer, multiple versions of a program, statically linked library, shared library, or other type of part can exist. Sometimes they must exist because separate, unrelated applications might make use of differing versions of the same library (or program or whatever).

That's true, & it's excusable, & it can also be pretty bad. (Think "DLL hell".) It's also different from using different versions of the same part *in the same application.*

Anyway, this situation of multiple versions of a part are used by different applications could easily be necessary when the part is a library made by an independent company. Which brings me to an exception.

6 An exception to my answer

After (hopefully) convincing you that multiple versions are bad bad bad, here's a possible exception.

Let's assume the same situation: We have an application consisting of multiple parts, & many of those parts use a part called Cache. The question is whether or not we should allow the various parts to use different versions of Cache.

I've said No, but that was assuming that Cache was developed by the same organization which develops & maintains the application as a whole.

What if Cache was developed by another organization? Like what if Cache was a product of an entirely separate company, & we are a customer of that company?

I could imagine multiple versions of Cache working in this case because, if at some time an old version (such as Cache version 1) was just too old & feeble to be worth using, & the application's team was requesting a new feature to that old version, the company which made Cache would say "Sorry, Jacques, but Cache version 1 is just too old. You must upgrade. So sorry." Basically, when the cost of maintaining multiple versions rose too much, they would axe some of the versions. The other company would balance the number of extant versions with the cost of maintaining them.

I could imagine this working, but it assumes that Cache's maker has the balls to say something like that. While saying things like that makes business sense when it makes business sense, I've worked at a company or two[1] which were notorious for never, ever having the balls to say such a thing. They would argue that it never makes business sense to say No to a customer. I say they are wimps, but if they are right, then I retract my suggestion that "third party manufacturer" would be an exception to "Do not ever allow multiple versions".

7 But it can be done!

Just because it's technically possible doesn't mean it's a good idea.

- It's technically possible to write self-modifying code, but the only good reason to do it is to discover why it's a bad idea. (And I have done it in Lisp, `/bin/sh`, SQL, & two types of assembly.)
- It's technically possible to write graphical user interfaces in FORTRAN, but it's not a good idea.
- It's technically possible to write a Prolog interpreter for a BASIC interpreter which is a compiled COBOL program, but that doesn't mean it's a good idea. (I did it with a friend in 1985 to prove a point, so I should know.)

Seriously, though...

If two separate programs of the application statically link with different versions of the same component, that would technically work. For example, maybe the Server part could link with Splay-Heap version 1 & the Client part linked with Splay-Heap version 2. It still runs the increased risk of parallel versions, & that alone is enough to preclude multiple versions.

Multiple versions of shared libraries can co-exist if you are careful to use a unique file name for each version. What happens if one program links to two shared libraries which use different versions of a third shared library, & the different versions of that same shared library define the same symbols? I don't know the answer, & I wouldn't be surprised if it depended on the operating system or the load-time linker, but the fact that it's a legitimate question which requires nearly guru-level knowledge of the specific operating system, knowledge which many experienced programmers will not have, is enough to discourage the

[1] I won't name names... Metapath. Marconi.

practice. Combined with other arguments I've made, it should be enough to preclude the practice.

Imagine reading through a debug log. Most of your functions include their names, source file names, & line numbers in each message. Do they also include the version numbers of the parts in which they exist? If they don't, those debug messages contain some ambiguity which might require at least a little work to dispel. What if the messages are from functions with the same name but different implementations because they are in different versions of the same shared library linked into one program? It's exactly the kind of situation which might not be much trouble most of the time, but when it is trouble, it'll cost weeks (literally) & have people rending their clothes & shouting "Why hath god forsaken us?"

8 Conclusion

All this is why...

1. multiple versions of parts is basically a bad idea, but
2. there are some special cases in which it *might* be okay, but
3. those special cases are narrow in scope, so
4. multiple versions are a bad idea.

A Other File Formats

- This document is available in multi-file HTML format at http://cybertiggyr.com/gene/bus/.
- This document is available in Pointless Document Format (PDF) at http://cybertiggyr.com/gene/bus/bus